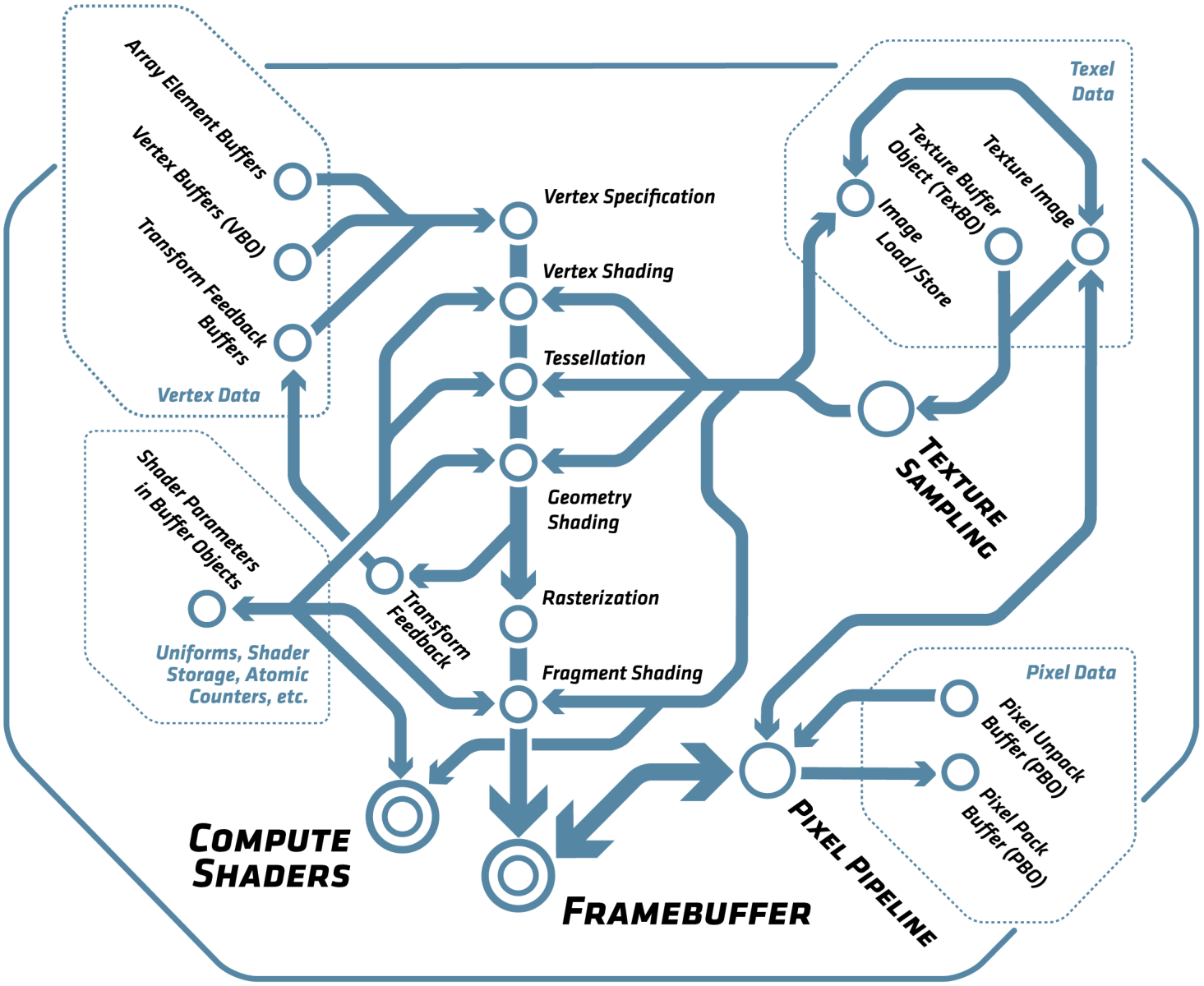
OpenGL 4.6

Compatibility Profile



The OpenGL® Graphics System: A Specification (Version 4.6 (Compatibility Profile) - May 5, 2022)

Mark Segal
Kurt Akeley

Editor (version 1.1): Chris Frazier
Editor (versions 1.2-4.6): Jon Leech
Editor (version 2.0): Pat Brown

Copyright © 2006-2022 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to Khronos. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos.

This specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at www.khronos.org/files/member_agreement.pdf. Khronos grants a conditional copyright license to use and reproduce the unmodified specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms. Parties desiring to implement the specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos IP Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this specification; see https://www.khronos.org/adopters.

Khronos makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation: merchantability, fitness for a particular purpose, non-infringement of any intellectual property, correctness, accuracy, completeness, timeliness, and reliability. Under no circumstances will Khronos, or any of its Promoters, Contributors or Members, or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Vulkan and Khronos are registered trademarks, and WebGL and EGL are trademarks of The Khronos Group Inc. ASTC is a trademark of ARM Holdings PLC; OpenCL is a trademark of Apple Inc.; and OpenGL is a registered trademark, and the OpenGL ES and OpenGL SC logos are trademarks of Hewlett Packard Enterprise, all used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Contents

1 Introduction **1**

1.1 Formatting of the OpenGL Specification 1

1.1.1 Formatting of the Compatibility Profile 1

1.1.2 Formatting of Optional Features 1

1.2 What is the OpenGL Graphics System? 2

1.2.1 Programmer's View of OpenGL 2

1.2.2 Implementor's View of OpenGL 2

1.2.3 Our View 3

1.2.4 Fixed-function Hardware and the Compatibility Profile . . 3

1.2.5 The Deprecation Model 3

1.3 Related APIs 4

1.3.1 OpenGL Shading Language 4

1.3.2 OpenGL ES 4

1.3.3 OpenGL ES Shading Language 5

1.3.4 SPIR-V 5

1.3.5 WebGL 6

1.3.6 Window System Bindings 6

1.3.7 OpenCL 7

1.4 Filing Bug Reports 7

2 OpenGL Fundamentals **8**

2.1 Execution Model 8

2.2 Command Syntax 10

2.2.1 Data Conversion For State-Setting Commands 12

2.2.2 Data Conversions For State Query Commands 14

2.3 Command Execution 15

2.3.1 Errors 16

2.3.2 Graphics Reset Recovery 19

2.3.3 Flush and Finish 21

2.3.4 Numeric Representation and Computation 21
2.3.5 Fixed-Point Data Conversions 25
2.4 Rendering Commands 27
2.5 Context State 27
2.5.1 Generic Context State Queries 28
2.6 Objects and the Object Model 28
2.6.1 Object Management 29
2.6.2 Buffer Objects 30
2.6.3 Shader Objects 30
2.6.4 Program Objects 30
2.6.5 Program Pipeline Objects 31
2.6.6 Texture Objects 31
2.6.7 Sampler Objects 31
2.6.8 Renderbuffer Objects 32
2.6.9 Framebuffer Objects 32
2.6.10 Vertex Array Objects 32
2.6.11 Transform Feedback Objects 32
2.6.12 Query Objects 33
2.6.13 Sync Objects 33
2.6.14 Display Lists 33

3 Dataflow Model **34**

4 Event Model **37**
4.1 Sync Objects and Fences 37
4.1.1 Waiting for Sync Objects 39
4.1.2 Signaling 41
4.1.3 Sync Object Queries 42
4.2 Query Objects and Asynchronous Queries 43
4.2.1 Query Object Types and Targets 43
4.2.2 Query Object Creation and Activation 44
4.2.3 Query Object Queries 48
4.3 Time Queries 51

5 Shared Objects and Multiple Contexts **53**
5.1 Object Deletion Behavior 54
5.1.1 Side Effects of Shared Context Destruction 54
5.1.2 Automatic Unbinding of Deleted Objects 54
5.1.3 Deleted Object and Object Name Lifetimes 54
5.2 Sync Objects and Multiple Contexts 55

5.3 Propagating Changes to Objects 55
5.3.1 Determining Completion of Changes to an object 56
5.3.2 Definitions 57
5.3.3 Rules 57

6 Buffer Objects **59**
6.1 Creating and Binding Buffer Objects 60
6.1.1 Binding Buffer Objects to Indexed Targets 62
6.2 Creating and Modifying Buffer Object Data Stores 65
6.2.1 Clearing Buffer Object Data Stores 71
6.3 Mapping and Unmapping Buffer Data 73
6.3.1 Unmapping Buffers 78
6.3.2 Effects of Mapping Buffers on Other GL Commands . . . 79
6.4 Effects of Accessing Outside Buffer Bounds 79
6.5 Invalidating Buffer Data 79
6.6 Copying Between Buffers 80
6.7 Buffer Object Queries 81
6.7.1 Indexed Buffer Object Limits and Binding Queries 83
6.8 Buffer Object State 85

7 Programs and Shaders **86**
7.1 Shader Objects 87
7.2 Shader Binaries 90
7.2.1 Shader Specialization 92
7.3 Program Objects 93
7.3.1 Program Interfaces 101
7.4 Program Pipeline Objects 120
7.4.1 Shader Interface Matching 124
7.4.2 SPIR-V Shader Interface Matching 128
7.4.3 Program Pipeline Object State 129
7.5 Program Binaries 129
7.6 Uniform Variables 132
7.6.1 Loading Uniform Variables In The Default Uniform Block 139
7.6.2 Uniform Blocks 143
7.6.3 Uniform Buffer Object Bindings 147
7.7 Atomic Counter Buffers 148
7.7.1 Atomic Counter Buffer Object Storage 149
7.7.2 Atomic Counter Buffer Bindings 149
7.8 Shader Buffer Variables and Shader Storage Blocks 150
7.9 Invocation Groups 152

7.10 Subroutine Uniform Variables . . . 153
7.11 Samplers . . . 156
7.12 Images . . . 157
7.13 Shader Memory Access . . . 158
7.13.1 Shader Memory Access Ordering . . . 158
7.13.2 Shader Memory Access Synchronization . . . 160
7.14 Shader, Program, and Program Pipeline Queries . . . 165
7.15 Required State . . . 175

8 Textures and Samplers **177**
8.1 Texture Objects . . . 180
8.2 Sampler Objects . . . 185
8.3 Sampler Object Queries . . . 189
8.4 Pixel Rectangles . . . 190
8.4.1 Pixel Storage Modes and Pixel Buffer Objects . . . 190
8.4.2 The Imaging Subset . . . 192
8.4.3 Pixel Transfer Modes . . . 193
8.4.4 Transfer of Pixel Rectangles . . . 213
8.4.5 Pixel Transfer Operations . . . 226
8.5 Texture Image Specification . . . 236
8.5.1 Required Texture Formats . . . 240
8.5.2 Encoding of Special Internal Formats . . . 240
8.5.3 Texture Image Structure . . . 245
8.6 Alternate Texture Image Specification Commands . . . 253
8.6.1 Texture Copying Feedback Loops . . . 260
8.7 Compressed Texture Images . . . 260
8.8 Multisample Textures . . . 268
8.9 Buffer Textures . . . 270
8.10 Texture Parameters . . . 273
8.11 Texture Queries . . . 278
8.11.1 Active Texture . . . 278
8.11.2 Texture Parameter Queries . . . 278
8.11.3 Texture Level Parameter Queries . . . 279
8.11.4 Texture Image Queries . . . 282
8.12 Depth Component Textures . . . 289
8.13 Cube Map Texture Selection . . . 289
8.13.1 Seamless Cube Map Filtering . . . 290
8.14 Texture Minification . . . 291
8.14.1 Scale Factor and Level-of-Detail . . . 293
8.14.2 Coordinate Wrapping and Texel Selection . . . 296

8.14.3 Mipmapping . . . 301
8.14.4 Manual Mipmap Generation . . . 303
8.14.5 Automatic Mipmap Generation . . . 304
8.15 Texture Magnification . . . 305
8.16 Combined Depth/Stencil Textures . . . 305
8.17 Texture Completeness . . . 306
8.17.1 Effects of Sampler Objects on Texture Completeness . . . 307
8.17.2 Effects of Completeness on Texture Application . . . 307
8.17.3 Effects of Completeness on Texture Image Specification . . . 307
8.18 Texture Views . . . 308
8.19 Immutable-Format Texture Images . . . 311
8.19.1 Behavior of Immutable-Format Texture Images . . . 318
8.20 Invalidating Texture Image Data . . . 318
8.21 Clearing Texture Image Data . . . 319
8.22 Texture State and Proxy State . . . 321
8.23 Texture Comparison Modes . . . 324
8.23.1 Depth Texture Comparison Mode . . . 324
8.24 sRGB Texture Color Conversion . . . 326
8.25 Shared Exponent Texture Color Conversion . . . 327
8.26 Texture Image Loads and Stores . . . 327
8.26.1 Image Unit Queries . . . 336

9 Framebuffers and Framebuffer Objects 337
9.1 Framebuffer Overview . . . 337
9.2 Binding and Managing Framebuffer Objects . . . 339
9.2.1 Framebuffer Object Parameters . . . 343
9.2.2 Attaching Images to Framebuffer Objects . . . 344
9.2.3 Framebuffer Object Queries . . . 345
9.2.4 Renderbuffer Objects . . . 351
9.2.5 Required Renderbuffer Formats . . . 354
9.2.6 Renderbuffer Object Queries . . . 355
9.2.7 Attaching Renderbuffer Images to a Framebuffer . . . 356
9.2.8 Attaching Texture Images to a Framebuffer . . . 358
9.3 Feedback Loops Between Textures and the Framebuffer . . . 363
9.3.1 Rendering Feedback Loops . . . 364
9.3.2 Texture Copying Feedback Loops . . . 365
9.4 Framebuffer Completeness . . . 365
9.4.1 Framebuffer Attachment Completeness . . . 366
9.4.2 Whole Framebuffer Completeness . . . 367
9.4.3 Required Framebuffer Formats . . . 370

9.4.4 Effects of Framebuffer Completeness on Framebuffer Operations . . . 371
9.4.5 Effects of Framebuffer State on Framebuffer Dependent Values . . . 371
9.5 Mapping between Pixel and Element in Attached Image . . . 372
9.6 Conversion to Framebuffer-Attachable Image Components . . . 373
9.7 Conversion to RGBA Values . . . 373
9.8 Layered Framebuffers . . . 373

10 Vertex Specification and Drawing Commands 376
10.1 Primitive Types . . . 380
10.1.1 Points . . . 380
10.1.2 Line Strips . . . 381
10.1.3 Line Loops . . . 381
10.1.4 Separate Lines . . . 381
10.1.5 Polygons . . . 381
10.1.6 Triangle Strips . . . 382
10.1.7 Triangle Fans . . . 382
10.1.8 Separate Triangles . . . 383
10.1.9 Quadrilateral (quad) strips . . . 383
10.1.10 Separate Quadrilaterals . . . 383
10.1.11 Lines with Adjacency . . . 384
10.1.12 Line Strips with Adjacency . . . 385
10.1.13 Triangles with Adjacency . . . 385
10.1.14 Triangle Strips with Adjacency . . . 386
10.1.15 Separate Patches . . . 387
10.1.16 General Considerations For Polygon Primitives . . . 388
10.1.17 Polygon Edges . . . 389
10.2 Current Vertex Attribute Values . . . 389
10.2.1 Current Generic Attributes . . . 389
10.2.2 Current Conventional Attributes . . . 392
10.2.3 Vertex Attribute Queries . . . 395
10.2.4 Required State . . . 395
10.3 Vertex Arrays . . . 395
10.3.1 Vertex Array Objects . . . 396
10.3.2 Specifying Arrays for Generic Vertex Attributes . . . 398
10.3.3 Specifying Arrays for Fixed-Function Attributes . . . 405
10.3.4 Vertex Attribute Divisors . . . 408
10.3.5 Transferring Array Elements . . . 409
10.3.6 Primitive Restart . . . 409

10.3.7 Robust Buffer Access . . . 410
10.3.8 Packed Vertex Data Formats . . . 410
10.3.9 Vertex Arrays in Buffer Objects . . . 411
10.3.10 Array Indices in Buffer Objects . . . 412
10.3.11 Indirect Commands in Buffer Objects . . . 413
10.4 Drawing Commands Using Vertex Arrays . . . 413
10.4.1 Interleaved Arrays . . . 426
10.5 Vertex Array and Vertex Array Object Queries . . . 428
10.6 Required State . . . 431
10.7 Drawing Commands Using **Begin** and **End** . . . 432
10.7.1 Transferring Vertices With **Vertex** Commands . . . 433
10.7.2 Transferring Vertices With Vertex Attribute Zero . . . 434
10.7.3 Bundling Attributes With **Vertex** Commands . . . 434
10.7.4 Transferring Vertices With **ArrayElement** . . . 434
10.7.5 Commands Allowed Between **Begin** and **End** . . . 436
10.8 Rectangles . . . 437
10.9 Conditional Rendering . . . 437
10.10 Submission Queries . . . 439

11 Programmable Vertex Processing **441**
11.1 Vertex Shaders . . . 441
11.1.1 Vertex Attributes . . . 441
11.1.2 Vertex Shader Variables . . . 447
11.1.3 Shader Execution . . . 453
11.1.4 Vertex Shader Queries . . . 467
11.2 Tessellation . . . 467
11.2.1 Tessellation Control Shaders . . . 469
11.2.2 Tessellation Primitive Generation . . . 474
11.2.3 Tessellation Evaluation Shaders . . . 482
11.2.4 Tessellation Shader Queries . . . 487
11.3 Geometry Shaders . . . 488
11.3.1 Geometry Shader Input Primitives . . . 489
11.3.2 Geometry Shader Output Primitives . . . 490
11.3.3 Geometry Shader Variables . . . 491
11.3.4 Geometry Shader Execution Environment . . . 491
11.3.5 Geometry Shader Queries . . . 498

12 Fixed-Function Vertex Processing **499**
12.1 Fixed-Function Vertex Transformations 499
12.1.1 Matrices 500
12.1.2 Normal Transformation 506
12.1.3 Generating Texture Coordinates 507
12.2 Fixed-Function Vertex Lighting and Coloring 510
12.2.1 Lighting 510
12.2.2 Lighting Parameter Specification 516
12.2.3 **ColorMaterial** 519
12.2.4 Lighting Parameter Queries 519
12.2.5 Lighting State 521
12.2.6 Color Index Lighting 521

13 Fixed-Function Vertex Post-Processing **523**
13.1 Clamping or Masking 524
13.2 The Last Vertex Processing Stage 524
13.3 Transform Feedback 524
13.3.1 Transform Feedback Objects 525
13.3.2 Transform Feedback Primitive Capture 527
13.3.3 Transform Feedback Draw Operations 533
13.4 Primitive Queries 534
13.5 Transform Feedback Overflow Queries 535
13.6 Flatshading 535
13.7 Primitive Clipping 537
13.7.1 Color and Associated Data Clipping 541
13.7.2 Clip Plane Queries 541
13.7.3 Primitive Clipping Queries 542
13.8 Coordinate Transformations 543
13.8.1 Controlling the Viewport 543
13.9 Final Color Processing 547

14 Fixed-Function Primitive Assembly and Rasterization **548**
14.1 Discarding Primitives Before Rasterization 550
14.2 Invariance 550
14.3 Antialiasing 550
14.3.1 Multisampling 552
14.4 Points 555
14.4.1 Basic Point Rasterization 557
14.4.2 Point Rasterization State 561
14.4.3 Point Multisample Rasterization 561

14.5 Line Segments . . . 562
14.5.1 Basic Line Segment Rasterization . . . 562
14.5.2 Other Line Segment Features . . . 565
14.5.3 Line Rasterization State . . . 568
14.5.4 Line Multisample Rasterization . . . 568
14.6 Polygons . . . 569
14.6.1 Basic Polygon Rasterization . . . 569
14.6.2 Stippling . . . 572
14.6.3 Antialiasing . . . 573
14.6.4 Options Controlling Polygon Rasterization . . . 573
14.6.5 Depth Offset . . . 573
14.6.6 Polygon Multisample Rasterization . . . 575
14.6.7 Polygon Rasterization State . . . 576
14.7 Current Raster Position . . . 576
14.8 Bitmaps . . . 579
14.9 Early Per-Fragment Tests . . . 581
14.9.1 Pixel Ownership Test . . . 582
14.9.2 Scissor Test . . . 582
14.9.3 Multisample Fragment Operations . . . 584
14.9.4 The Early Fragment Test Qualifier . . . 585

15 Programmable Fragment Processing **587**
15.1 Fragment Shader Variables . . . 587
15.2 Shader Execution . . . 589
15.2.1 Texture Access . . . 589
15.2.2 Shader Inputs . . . 589
15.2.3 Shader Outputs . . . 592
15.2.4 Early Fragment Tests . . . 597
15.3 Fragment Shader Queries . . . 597

16 Fixed-Function Fragment Processing **598**
16.1 Texture Environments and Texture Functions . . . 598
16.1.1 Texture Environment Queries . . . 604
16.2 Texture Application . . . 605
16.3 Color Sum . . . 606
16.4 Fog . . . 608

17 Writing Fragments and Samples to the Framebuffer **611**
17.1 Antialiasing Application . . . 611
17.2 Multisample Point Fade . . . 612

17.3 Per-Fragment Operations 612
17.3.1 Alpha To Coverage 612
17.3.2 Alpha Test 614
17.3.3 Stencil Test 615
17.3.4 Depth Buffer Test 616
17.3.5 Occlusion Queries 617
17.3.6 Blending 618
17.3.7 sRGB Conversion 625
17.3.8 Dithering 626
17.3.9 Logical Operation 626
17.3.10 Additional Multisample Fragment Operations 628
17.4 Whole Framebuffer Operations 629
17.4.1 Selecting Buffers for Writing 629
17.4.2 Fine Control of Buffer Updates 634
17.4.3 Clearing the Buffers 636
17.4.4 Invalidating Framebuffer Contents 640
17.4.5 The Accumulation Buffer 641

18 Drawing, Reading, and Copying Pixels 644
18.1 Drawing Pixels 644
18.1.1 Final Conversion 645
18.1.2 Conversion to Fragments 646
18.1.3 Pixel Rectangle Multisample Rasterization 647
18.1.4 Writing to the Stencil or Depth/Stencil Buffers 647
18.2 Reading Pixels 648
18.2.1 Selecting Buffers for Reading 648
18.2.2 ReadPixels 650
18.2.3 Obtaining Pixels from the Framebuffer 652
18.2.4 Conversion of RGBA values 653
18.2.5 Conversion of Depth values 654
18.2.6 Pixel Transfer Operations 654
18.2.7 Conversion to L 654
18.2.8 Final Conversion 654
18.2.9 Placement in Pixel Pack Buffer or Client Memory 655
18.3 Copying Pixels 657
18.3.1 Copying Pixel Rectangles 658
18.3.2 Blitting Pixel Rectangles 660
18.3.3 Copying Between Images 664
18.4 Pixel Draw and Read State 667

19 Compute Shaders **668**
19.1 Compute Shader Variables 670
19.2 Compute Shader Queries 670

20 Debug Output **671**
20.1 Debug Messages 672
20.2 Debug Message Callback 674
20.3 Debug Message Log 675
20.4 Controlling Debug Messages 675
20.5 Externally Generated Messages 677
20.6 Debug Groups 677
20.7 Debug Labels 679
20.8 Asynchronous and Synchronous Debug Output 680
20.9 Debug Output Queries 681

21 Special Functions **684**
21.1 Evaluators 684
21.1.1 Evaluator Queries 691
21.2 Selection 692
21.3 Feedback 695
21.4 Display Lists 697
21.4.1 Commands Not Usable In Display Lists 701
21.5 Hints 703
21.6 Saving and Restoring State 704

22 Context State Queries **708**
22.1 Simple Queries 708
22.2 Pointer, String, and Related Context Queries 710
22.3 Internal Format Queries 713
22.3.1 Supported Operation Queries 715
22.3.2 Other Internal Format Queries 718
22.4 Transform Feedback State Queries 724
22.5 Indexed Binding State Queries 725

23 State Tables **727**

A Invariance **816**
A.1 Repeatability 816
A.2 Multi-pass Algorithms 817
A.3 Invariance Rules 817

A.4 Tessellation Invariance 820
A.5 Atomic Counter Invariance 822
A.6 What All This Means 822

B Corollaries **824**

C The OpenGL SPIR-V Execution Environment **827**
C.1 Required Versions and Formats 827
C.2 Valid SPIR-V Built-In Variable Decorations 827
C.3 Valid SPIR-V Capabilities 827
C.4 Validation Rules 828
C.5 Precision and Operation of SPIR-V Instructions 830
C.6 Precision of GLSL.std.450 Instructions 831

D Compressed Texture Image Formats **835**
D.1 RGTC Compressed Texture Image Formats 835
D.2 BPTC Compressed Texture Image Formats 836
D.3 ETC Compressed Texture Image Formats 836

E Profiles and the Deprecation Model **838**
E.1 Core and Compatibility Profiles 839
E.2 Deprecated and Removed Features 839
E.2.1 Deprecated But Still Supported Features 839
E.2.2 Removed Features 840

F Version 4.2 **845**
F.1 New Features 845
F.2 Deprecation Model 846
F.3 Changed Tokens 846
F.4 Change Log for Released Specifications 847
F.5 Credits and Acknowledgements 849

G Version 4.3 **852**
G.1 Restructuring 852
G.2 New Features 853
G.3 Deprecation Model 854
G.4 Changed Tokens 854
G.5 Change Log for Released Specifications 855
G.6 Credits 862
G.7 Acknowledgements 864

H Version 4.4 **865**
H.1 New Features . . . 865
H.2 Deprecation Model . . . 866
H.3 Change Log for Released Specifications . . . 866
H.4 Credits . . . 877
H.5 Acknowledgements . . . 878

I Version 4.5 **880**
I.1 New Features . . . 880
I.2 Deprecation Model . . . 881
I.3 Change Log for Released Specifications . . . 881
I.4 Credits . . . 893
I.5 Acknowledgements . . . 895

J Version 4.6 **896**
J.1 New Features . . . 896
J.2 Deprecation Model . . . 897
J.3 Change Log Descriptions . . . 897
J.4 Change Log for Released Specifications . . . 897
J.5 Credits . . . 906
J.6 Acknowledgements . . . 907

K OpenGL Registry, Header Files, and ARB Extensions **908**
K.1 OpenGL Registry . . . 908
K.2 Header Files . . . 908
K.3 ARB and Khronos Extensions . . . 909
K.3.1 Naming Conventions . . . 910
K.3.2 Promoting Extensions to Core Features . . . 910
K.3.3 Extension Summaries . . . 910

Index **942**

List of Figures

3.1 Block diagram of the GL pipeline. 35

8.1 Transfer of pixel rectangles. 213
8.2 Selecting a subimage from an image 218
8.3 A texture image and the coordinates used to access it. 252
8.4 Example of the components returned for `textureGather`. 298

10.1 Vertex processing and primitive assembly. 376
10.2 Creation of a processed vertex from a transformed vertex and current values. 378
10.3 Primitive assembly and processing. 378
10.4 Triangle strips, fans, and independent triangles. 382
10.5 Quadrilateral strips and independent quadrilaterals. 383
10.6 Lines with adjacency. 384
10.7 Triangles with adjacency. 385
10.8 Triangle strips with adjacency. 386

11.1 Domain parameterization for tessellation. 474
11.2 Inner triangle tessellation. 478
11.3 Inner quad tessellation. 480
11.4 Isoline tessellation. 482

12.1 Vertex transformation sequence. 499
12.2 Processing of RGBA colors. 510
12.3 Processing of color indices. 510
12.4 ColorMaterial operation. 519

14.1 Rasterization. 548
14.2 Rasterization of non-antialiased wide points. 557
14.3 Rasterization of antialiased wide points. 557

14.4 Visualization of Bresenham's algorithm. 563
14.5 Rasterization of non-antialiased wide lines. 566
14.6 The region used in rasterizing an antialiased line segment. 567
14.7 Current raster position. 577
14.8 A bitmap and its associated parameters. 580

16.1 Multitexture pipeline. 606

17.1 Per-fragment operations. 612

18.1 Operation of **ReadPixels**. 648
18.2 Operation of **CopyPixels**. 658

21.1 Map Evaluation. 686
21.2 Feedback syntax. 698

List of Tables

1.1 OpenGL ES to OpenGL version relationship. 5

2.1 GL command suffixes 12
2.2 GL data types 13
2.3 Summary of GL errors 18

4.1 Initial properties of a sync object created with **FenceSync**. 38

6.1 Buffer object binding targets. 61
6.2 Buffer object parameters and their values. 61
6.3 Buffer object state. 67
6.4 Buffer object state set by **MapBufferRange** and **MapNamedBufferRange**. 75
6.5 Indexed buffer object limits and binding queries 84

7.1 **CreateShader** *type* values and the corresponding shader stages. . 88
7.2 **GetProgramResourceiv** properties and supported interfaces . . . 111
7.3 OpenGL Shading Language type tokens 118
7.4 Query targets for default uniform block storage, in components. . 133
7.5 Query targets for combined uniform block storage, in components. 133
7.6 **GetProgramResourceiv** properties used by **GetActiveUniformsiv**. 137
7.7 **GetProgramResourceiv** properties used by **GetActiveUniformBlockiv**. 138
7.8 **GetProgramResourceiv** properties used by **GetActiveAtomicCounterBufferiv**. 139
7.9 Interfaces for active subroutines 154
7.10 Interfaces for active subroutine uniforms 154

8.1 **PixelStore*** parameters. 191
8.2 **PixelTransfer** parameters. 193

8.3 **PixelMap** parameters. 194
8.4 Color table names. 197
8.5 Pixel data formats accepted for the imaging queries. 200
8.6 Pixel data types accepted for the imaging queries. 201
8.7 Pixel data types. 215
8.8 Pixel data formats. 217
8.9 Swap Bytes bit ordering. 218
8.10 Packed pixel formats. 220
8.11 `UNSIGNED_BYTE` formats. Bit numbers are indicated for each component. 221
8.12 `UNSIGNED_SHORT` formats 222
8.13 `UNSIGNED_INT` formats 223
8.14 `FLOAT_UNSIGNED_INT` formats 224
8.15 Packed pixel field assignments. 225
8.16 Color table lookup. 229
8.17 Computation of filtered color components. 230
8.18 Conversion from RGBA, depth, and stencil pixel components to internal texture, table, or filter components. 238
8.19 Sized internal color formats. 243
8.20 Sized internal luminance and intensity formats. 244
8.21 Sized internal depth and stencil formats. 245
8.22 Generic and specific compressed internal formats. 246
8.23 Valid texture *target* parameters 256
8.24 Internal formats for buffer textures 273
8.25 Texture parameters and their values. 276
8.26 Texture, table and filter return values. 285
8.27 Selection of cube map images. 290
8.28 Texel location wrap mode application. 297
8.29 Legal texture targets for **TextureView**. 309
8.30 Compatible internal formats for **TextureView** 310
8.31 Depth texture comparison functions. 325
8.32 sRGB texture internal formats. 326
8.33 Mapping of image load, store, and atomic texel coordinate components to texel numbers. 331
8.34 Supported image unit formats, with equivalent format `layout` qualifiers. 334
8.35 Texel sizes, compatibility classes, and pixel format/type combinations for each image format. 336

9.1 Buffer selection for default framebuffer attachment queries 347

9.2 Framebuffer attachment points. 357
9.3 Layer numbers for cube map texture faces. 374

10.1 Triangles generated by triangle strips with adjacency. 387
10.2 Alternative ordering to triangles generated by triangle strips with adjacency (see table 10.1). 388
10.3 Vertex array sizes (values per vertex) and data types for generic vertex attributes 399
10.4 Fixed-function vertex array sizes and data types 406
10.5 Packed component layout for non-BGRA formats. 411
10.6 Packed component layout for BGRA format. 411
10.7 Packed component layout for UNSIGNED_INT_10F_11F_11F_REV format. 411
10.8 Indirect commands and corresponding indirect buffer targets. . . . 413
10.9 Variables that direct the execution of **InterleavedArrays**. 427

11.1 Generic attribute components accessed by attribute variables. . . . 442
11.2 Generic attributes and vector types used by column vectors of matrix variables bound to generic attribute index i. 443
11.3 Scalar and vector vertex attribute types 443

12.1 Summary of lighting parameters. 513
12.2 Correspondence of lighting parameter symbols to names. 517

13.1 Transform feedback modes 529
13.2 Provoking vertex selection. 536

16.1 Correspondence of filtered texture components to texture base components. 600
16.2 Texture functions REPLACE, MODULATE, and DECAL 601
16.3 Texture functions BLEND and ADD. 601
16.4 COMBINE texture functions. 602
16.5 Arguments for COMBINE_RGB functions. 603
16.6 Arguments for COMBINE_ALPHA functions. 603

17.1 RGB and alpha blend equations. 621
17.2 Blending functions. 623
17.3 Logical operations 627
17.4 Buffer selection for the default framebuffer 631
17.5 Buffer selection for a framebuffer object 631
17.6 **DrawBuffers** buffer selection for the default framebuffer 632

18.1 **PixelStore** parameters. 651
18.2 **ReadPixels** GL data types and reversed component conversion formulas. 656
18.3 **ReadPixels** index masks. 657
18.4 Effective **ReadPixels** format for `DEPTH_STENCIL` **CopyPixels** operation. 660
18.5 Compatible internal formats for copying 666

20.1 Sources of debug output messages 672
20.2 Types of debug output messages 673
20.3 Severity levels of messages 673
20.4 Object namespace identifiers 679

21.1 Values specified by the *target* to **Map1**. 685
21.2 Correspondence of feedback type to number of values per vertex. . 696
21.3 Hint targets and descriptions 704
21.4 Attribute groups 706

22.1 Context profile bits 712
22.2 Internal format targets 714

23.1 State Variable Types 729
23.2 GL Internal State (inaccessible) 730
23.3 Current Values and Associated Data 731
23.4 Vertex Array Object State 732
23.5 Vertex Array Object State (cont.) 733
23.6 Vertex Array Object State (cont.) 734
23.7 Vertex Array Object State (cont.) 735
23.8 Vertex Array Data (not in Vertex Array objects) 736
23.9 Buffer Object State 737
23.10 Transformation state 738
23.11 Coloring 739
23.12 Lighting 740
23.13 Lighting (cont.) 741
23.14 Rasterization 742
23.15 Rasterization (cont.) 743
23.16 Multisampling 744
23.17 Textures (state per texture unit) 745
23.18 Textures (state per texture unit) (cont.) 746
23.19 Textures (state per texture object) 747
23.20 Textures (state per texture object) (cont.) 748

23.21 Textures (state per texture image) . . . 749
23.22 Textures (state per texture image) (cont.) . . . 750
23.23 Textures (state per sampler object) . . . 751
23.24 Texture Environment and Generation . . . 752
23.25 Texture Environment and Generation (cont.) . . . 753
23.26 Pixel Operations . . . 754
23.27 Pixel Operations (cont.) . . . 755
23.28 Framebuffer Control . . . 756
23.29 Framebuffer (state per target binding point) . . . 757
23.30 Framebuffer (state per framebuffer object) . . . 758
23.31 Framebuffer (state per attachment point) . . . 759
23.32 Renderbuffer (state per target and binding point) . . . 760
23.33 Renderbuffer (state per renderbuffer object) . . . 761
23.34 Pixels . . . 762
23.35 Pixels (cont.) . . . 763
23.36 Pixels (cont.) . . . 764
23.37 Pixels (cont.) . . . 765
23.38 Pixels (cont.) . . . 766
23.39 Pixels (cont.) . . . 767
23.40 Evaluators . . . 768
23.41 Shader Object State . . . 769
23.42 Program Pipeline Object State . . . 770
23.43 Program Object State . . . 771
23.44 Program Object State (cont.) . . . 772
23.45 Program Object State (cont.) . . . 773
23.46 Program Object State (cont.) . . . 774
23.47 Program Object State (cont.) . . . 775
23.48 Program Object State (cont.) . . . 776
23.49 Program Object State (cont.) . . . 777
23.50 Program Object State (cont.) . . . 778
23.51 Program Interface State . . . 779
23.52 Program Object Resource State . . . 780
23.53 Program Object Resource State (cont.) . . . 781
23.54 Vertex and Geometry Shader State . . . 782
23.55 Query Object State . . . 783
23.56 Image State (state per image unit) . . . 784
23.57 Atomic Counter Buffer Binding State . . . 785
23.58 Shader Storage Buffer Binding State . . . 786
23.59 Transform Feedback Object State . . . 787
23.60 Uniform Buffer Binding State . . . 788

23.61 Sync Object State . . . 789
23.62 Hints . . . 790
23.63 Compute Dispatch State . . . 791
23.64 Implementation Dependent Values . . . 792
23.65 Implementation Dependent Values (cont.) . . . 793
23.66 Implementation Dependent Values (cont.) . . . 794
23.67 Implementation Dependent Values (cont.) . . . 795
23.68 Implementation Dependent Version and Extension Support . . . 796
23.69 Implementation Dependent Vertex Shader Limits . . . 797
23.70 Implementation Dependent Tessellation Shader Limits . . . 798
23.71 Implementation Dependent Tessellation Shader Limits (cont.) . . . 799
23.72 Implementation Dependent Geometry Shader Limits . . . 800
23.73 Implementation Dependent Fragment Shader Limits . . . 801
23.74 Implementation Dependent Compute Shader Limits . . . 802
23.75 Implementation Dependent Aggregate Shader Limits . . . 803
23.76 Implementation Dependent Aggregate Shader Limits (cont.) . . . 804
23.77 Implementation Dependent Aggregate Shader Limits (cont.) . . . 805
23.78 Implementation Dependent Aggregate Shader Limits (cont.) . . . 806
23.79 Debug Output State . . . 807
23.80 Implementation Dependent Debug Output State . . . 808
23.81 Implementation Dependent Values (cont.)
† These queries return the maximum no. of samples for all internal formats required to support multisampled rendering. . . . 809
23.82 Implementation Dependent Values (cont.)
† "ULP" means "Units in the Last Place" or "Unit of Least Precision". . . . 810
23.83 Internal Format Dependent Values . . . 811
23.84 Implementation Dependent Transform Feedback Limits . . . 812
23.85 Framebuffer Dependent Values . . . 813
23.86 Framebuffer Dependent Values (cont.) . . . 814
23.87 Miscellaneous . . . 815

C.1 Built-in Variable Decorations . . . 832
C.2 Valid SPIR-V Capabilities . . . 833
C.3 32-Bit Precision . . . 834
C.4 GLSL.std.450 Precision . . . 834

D.1 Mapping of OpenGL RGTC formats to descriptions. . . . 836
D.2 Mapping of OpenGL BPTC formats to descriptions. . . . 836
D.3 Mapping of OpenGL ETC formats to descriptions. . . . 837

F.1 New token names 847

G.1 New token names 855

Chapter 1

Introduction

This document, referred to as the "OpenGL Specification" or just "Specification" hereafter, describes the OpenGL graphics system: what it is, how it acts, and what is required to implement it. We assume that the reader has at least a rudimentary understanding of computer graphics. This means familiarity with the essentials of computer graphics algorithms and terminology as well as with modern GPUs (Graphic Processing Units).

The canonical version of the Specification is available in the official *OpenGL Registry*, located at URL

http://www.opengl.org/registry/

1.1 Formatting of the OpenGL Specification

Starting with version 4.3, the OpenGL Specification has undergone major restructuring to focus on programmable shading, and to describe important concepts and objects in the context of the entire API before describing details of their use in the graphics pipeline.

1.1.1 Formatting of the Compatibility Profile

Material which is present only in the compatibility profile Specification and not in the core Specification (see appendix E) is typeset in orange, like this paragraph.

1.1.2 Formatting of Optional Features

Some features in the Specification are considered optional; an OpenGL implementation may or may not choose to provide them (see section 8.4.2).

Portions of the Specification which are optional are so described where the optional features are first defined (see section 8.4.2). State table entries which are optional are typeset against a gray background.

1.2 What is the OpenGL Graphics System?

OpenGL (for "Open Graphics Library") is an *API* (Application Programming Interface) to graphics hardware. The API consists of a set of several hundred procedures and functions that allow a programmer to specify the shader programs, objects, and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

Most of OpenGL requires that the graphics hardware contain a framebuffer. Many OpenGL calls control drawing geometric objects such as points, lines, and polygons, but the way that some of this drawing occurs (such as when antialiasing or multisampling is in use) relies on the existence of a framebuffer and its properties. Some commands explicitly manage the framebuffer.

1.2.1 Programmer's View of OpenGL

To the programmer, OpenGL is a set of commands that allow the specification of *shader programs* or *shaders*, data used by shaders, and state controlling aspects of OpenGL outside the scope of shaders. Typically the data represent geometry in two or three dimensions and texture images, while the shaders control the geometric processing, rasterization of geometry and the lighting and shading of *fragments* generated by rasterization, resulting in rendering geometry into the framebuffer.

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate an OpenGL *context* and associate it with the window. Once a context is allocated, OpenGL commands to define shaders, geometry, and textures are made, followed by commands which draw geometry by transferring specified portions of the geometry to the shaders. Drawing commands specify simple geometric objects such as points, line segments, and polygons, which can be further manipulated by shaders. There are also commands which directly control the framebuffer by reading and writing pixels.

1.2.2 Implementor's View of OpenGL

To the implementor, OpenGL is a set of commands that control the operation of the GPU. Modern GPUs accelerate almost all OpenGL operations, storing data and framebuffer images in GPU memory and executing shaders in dedicated GPU

processors. However, OpenGL may be implemented on less capable GPUs, or even without a GPU, by moving some or all operations into the host CPU.

The implementor's task is to provide a software library on the CPU which implements the OpenGL API, while dividing the work for each OpenGL command between the CPU and the graphics hardware as appropriate for the capabilities of the GPU.

OpenGL contains a considerable amount of information including many types of objects representing programmable shaders and the data they consume and generate, as well as other *context state* controlling non-programmable aspects of OpenGL. Most of these objects and state are available to the programmer, who can set, manipulate, and query their values through OpenGL commands. Some of it, however, is *derived state* visible only by the effect it has on how OpenGL operates. One of the main goals of this Specification is to describe OpenGL objects and context state explicitly, to elucidate how they change in response to OpenGL commands, and to indicate what their effects are.

1.2.3 Our View

We view OpenGL as a pipeline having some programmable stages and some state-driven *fixed-function* stages that are invoked by a set of specific drawing operations. This model should engender a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. An implementation must produce results conforming to those produced by the specified methods, but there may be ways to carry out a particular computation that are more efficient than the one specified.

1.2.4 Fixed-function Hardware and the Compatibility Profile

Older generations of graphics hardware were not programmable using shaders, although they were configurable by setting state controlling specific details of their operation. The compatibility profile of OpenGL continues to support the legacy OpenGL commands developed for such fixed-function hardware, although they are typically implemented by writing shaders which reproduce the operation of such hardware. Fixed-function OpenGL commands and operations are described as alternative interfaces following descriptions of the corresponding shader stages.

1.2.5 The Deprecation Model

Features marked as *deprecated* in one version of the Specification are expected to be removed in a future version, allowing applications time to transition away from

use of deprecated features. The deprecation model is described in more detail, together with a summary of the commands and state deprecated from this version of the API, in appendix E.

1.3 Related APIs

Other APIs and related specifications related to OpenGL are described below. Most of the specifications for these APIs are available on the Khronos Group websites, although some vendor-specific APIs are documented on that vendor's developer website.

1.3.1 OpenGL Shading Language

The OpenGL Specification should be read together with a companion document titled *The OpenGL Shading Language*. The latter document (referred to as the OpenGL Shading Language Specification hereafter) defines the syntax and semantics of the programming language used to write shaders (see chapter 7). Descriptions of shaders later in this document may include references to concepts and terms (such as shading language variable types) defined in the OpenGL Shading Language Specification.

OpenGL 4.6 implementations are guaranteed to support version 4.60 of the OpenGL Shading Language. All references to sections of that specification refer to that version. The latest supported version of the shading language may be queried as described in section 22.2.

The compatibility profile of OpenGL 4.6 is also guaranteed to support all previous versions of the OpenGL Shading Language back to version 1.10. The `#version` strings for all supported versions of the OpenGL Shading Language may be queried as described in section 22.2.

The OpenGL Shading Language Specification is available in the OpenGL Registry.

1.3.2 OpenGL ES

OpenGL ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems such as mobile phones, game consoles, and vehicles. It consists of well-defined subsets of OpenGL. Each version of OpenGL ES implements a subset of a corresponding OpenGL version as shown in table 1.1.

OpenGL ES versions also include some additional functionality taken from later OpenGL versions or specific to OpenGL ES. It is straightforward to port code written for OpenGL ES to corresponding versions of OpenGL.

OpenGL ES Version	OpenGL Version it subsets
OpenGL ES 1.1	OpenGL 1.5
OpenGL ES 2.0	OpenGL 2.0
OpenGL ES 3.0	OpenGL 3.3
OpenGL ES 3.1	OpenGL 4.3

Table 1.1: OpenGL ES to OpenGL version relationship.

OpenGL and OpenGL ES are developed in parallel within the Khronos Group, which controls both standards.

OpenGL 4.3 and 4.5 include additional functionality initially defined in OpenGL ES 3.0 and OpenGL ES 3.1, respectively, for increased compatibility between OpenGL and OpenGL ES implementations.

The OpenGL ES Specifications are available in the *Khronos API Registry* at URL

http://www.khronos.org/registry/

1.3.3 OpenGL ES Shading Language

The Specification should also be read together with companion documents titled *The OpenGL ES Shading Language*. Versions 1.00, 3.00, and 3.10 should be read. These documents define versions of the OpenGL Shading Language designed for implementations of OpenGL ES 2.0, 3.0, and 3.1 respectively, but also supported by OpenGL implementations. References to the OpenGL Shading Language Specification hereafter include both OpenGL and OpenGL ES versions of the Shading Language; references to specific sections are to those sections in version 4.60 of the OpenGL Shading Language Specification.

OpenGL 4.6 implementations are guaranteed to support versions 1.00, 3.00, and 3.10 of the OpenGL ES Shading Language.

The `#version` strings for all supported versions of the OpenGL Shading Language may be queried as described in section 22.2.

The OpenGL ES Shading Language Specifications are available in the Khronos API Registry.

1.3.4 SPIR-V

SPIR-V is a binary intermediate language for representing graphical-shader stages and compute kernels for multiple Khronos APIs, such as OpenCL, OpenGL, and Vulkan.

The SPIR-V Specification, and the related SPIR-V Extended Instructions for the OpenGL Shading Language Specification, are available in the Khronos API Registry.

1.3.5 WebGL

WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES. Developers familiar with OpenGL ES will recognize WebGL as a shader-based API using the OpenGL Shading Language, with constructs that are semantically similar to those of the underlying OpenGL ES API. It stays very close to the OpenGL ES specification, with some concessions made for what developers expect out of memory-managed languages such as JavaScript.

The WebGL Specification and related documentation are available in the Khronos API Registry.

1.3.6 Window System Bindings

OpenGL requires a companion API to create and manage graphics contexts, windows to render into, and other resources beyond the scope of this Specification. There are several such APIs supporting different operating and window systems.

1.3.6.1 GLX - X Window System Bindings

OpenGL Graphics with the X Window System, referred to as the GLX Specification hereafter, describes the GLX API for use of OpenGL in the X Window System. It is primarily directed at Linux and Unix systems, but GLX implementations also exist for Microsoft Windows, MacOS X, and some other platforms where X is available. The GLX Specification is available in the OpenGL Registry.

1.3.6.2 WGL - Microsoft Windows Bindings

The WGL API supports use of OpenGL with Microsoft Windows. WGL is documented in Microsoft's MSDN system, although no full specification exists.

1.3.6.3 MacOS X Window System Bindings

Several APIs exist supporting use of OpenGL with Quartz, the MacOS X window system, including CGL, AGL, and NSOpenGLView. These APIs are documented on Apple's developer website.

1.3.6.4 EGL - Mobile and Embedded Device Bindings

The *Khronos Native Platform Graphics Interface* or "EGL Specification" describes the EGL API for use of OpenGL ES on mobile and embedded devices. EGL implementations supporting OpenGL may be available on some desktop platforms as well. The EGL Specification is available in the Khronos API Registry.

1.3.7 OpenCL

OpenCL is an open, royalty-free standard for cross-platform, general-purpose parallel programming of processors found in personal computers, servers, and mobile devices, including GPUs. OpenCL defines *interop* methods to share OpenCL memory and image objects with corresponding OpenGL buffer and texture objects, and to coordinate control of and transfer of data between OpenCL and OpenGL. This allows applications to split processing of data between OpenCL and OpenGL; for example, by using OpenCL to implement a physics model and then rendering and interacting with the resulting dynamic geometry using OpenGL.

The OpenCL Specification is available in the Khronos API Registry.

1.4 Filing Bug Reports

Bug reports on the OpenGL API Specification should be filed in the Khronos OpenGL-API Github repository, located at URL

https://github.com/KhronosGroup/OpenGL-API

Bug reports on the OpenGL Shading Language Specification should be filed in the Khronos OpenGL-GLSL Github repository, located at URL

https://github.com/KhronosGroup/OpenGL-GLSL

It is best to file bugs against the most recently released versions, since older versions are usually not updated for bugfixes.

Chapter 2

OpenGL Fundamentals

This chapter introduces fundamental concepts including the OpenGL execution model, API syntax, contexts and threads, numeric representation, context state and state queries, and the different types of objects and shaders. It provides a framework for interpreting more specific descriptions of commands and behavior in the remainder of the Specification.

2.1 Execution Model

OpenGL (henceforth, "the GL") is concerned only with processing data in GPU memory, including rendering into a framebuffer and reading values stored in that framebuffer. There is no support for other input or output devices. Programmers must rely on other mechanisms to obtain user input.

The GL draws *primitives* processed by a variety of shader programs and fixed-function processing units controlled by context state. Each primitive is a point, line segment, patch, pixel rectangle, or polygon. Context state may be changed independently; the setting of one piece of state does not affect the settings of others (although state and shaders all interact to determine what eventually ends up in the framebuffer). State is set, primitives drawn, and other GL operations described by sending *commands* in the form of function or procedure calls.

Primitives are defined by a group of one or more *vertices*. A vertex defines a point, an endpoint of a line segment, or a corner of a polygon where two edges meet. Data such as positional coordinates, colors, normals, texture coordinates, etc. are associated with a vertex and each vertex is processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be *clipped* so that the indicated primitive fits within a specified region; in this case vertex data may be modified and new vertices created. The type of clipping

depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before the effects of a command are realized. This means, for example, that one primitive must be drawn completely before any subsequent one can affect the framebuffer. It also means that queries and pixel read operations return state consistent with complete execution of all previously invoked GL commands, except where explicitly specified otherwise. In general, the effects of a GL command on either GL state or the framebuffer must be complete before any subsequent command can have any such effects.

Data binding occurs on call. This means that data passed to a GL command are interpreted when that command is received. Even if the command requires a pointer to data, those data are interpreted when the call is made, and any subsequent changes to the data have no effect on the GL (unless the same pointer is used in a subsequent command).

The GL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of parameters of application-defined shader programs performing transformation, lighting, texturing, and shading operations, as well as built-in functionality such as antialiasing and texture filtering. It does not provide a means for describing or modeling complex geometric objects, although shaders can be written to generate such objects. In other words, OpenGL provides mechanisms to describe how complex geometric objects are to be rendered, rather than mechanisms to describe the complex objects themselves.

The model for interpretation of GL commands is client-server. That is, a program (the client) issues commands, and these commands are interpreted and processed by the GL (the server). The server may or may not operate on the same computer or in the same address space as the client. In this sense, the GL is *network transparent*. A server may maintain a number of GL contexts, each of which is an encapsulation of current GL state and objects. A client may choose to be made *current* to any one of these contexts.

Issuing GL commands when the program does not have a current context results in undefined behavior, up to and including program termination.

There are two classes of framebuffers: a window system-provided framebuffer associated with a context when the context is made current, and application-created framebuffers. The window system-provided framebuffer is referred to as the *default framebuffer*. Application-created framebuffers, referred to as *framebuffer objects*, may be created as desired, A context may be associated with two framebuffers, one for each of reading and drawing operations. The default framebuffer and framebuffer objects are distinguished primarily by the interfaces for configuring and managing their state.

The effects of GL commands on the default framebuffer are ultimately con-

trolled by the window system, which allocates framebuffer resources, determines which portions of the default framebuffer the GL may access at any given time, and communicates to the GL how those portions are structured. Therefore, there are no GL commands to initialize a GL context or configure the default framebuffer. Similarly, display of framebuffer contents on a physical display device (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by the GL.

Allocation and configuration of the default framebuffer occurs outside of the GL in conjunction with the window system, using companion APIs described in section 1.3.6.

Allocation and initialization of GL contexts is also done using these companion APIs. GL contexts can be associated with different default framebuffers, and some context state is determined at the time this association is performed.

It is possible to use a GL context *without* a default framebuffer, in which case a framebuffer object must be used to perform all rendering. This is useful for applications needing to perform *offscreen rendering*.

OpenGL is designed to be run on a range of platforms with varying capabilities, memory, and performance. To accommodate this variety, we specify ideal behavior instead of actual behavior for certain GL operations. In cases where deviation from the ideal is allowed, we also specify the rules that an implementation must obey if it is to approximate the ideal behavior usefully. This allowed variation in GL behavior implies that two distinct GL implementations may not agree pixel for pixel when presented with the same input, even when run on identical framebuffer configurations.

Finally, command names, constants, and types are prefixed in the `C` language binding to OpenGL (by **gl**, `GL_`, and `GL`, respectively), to reduce name clashes with other packages. The prefixes are omitted in this document for clarity.

2.2 Command Syntax

The Specification describes OpenGL commands as functions or procedures using ANSI `C` syntax. Languages such as `C++` and `Javascript` which allow passing of argument type information permit language bindings with simpler declarations and fewer entry points.

Various groups of GL commands perform the same operation but differ in how arguments are supplied to them. To conveniently accommodate this variation, we adopt a notation for describing commands and their arguments.

GL commands are formed from a *name* which may be followed, depending on the particular command, by a sequence of characters describing a parameter to the

command. If present, a digit indicates the required length (number of values) of the indicated type. Next, a string of characters making up one of the *type descriptors* from table 2.1 indicates the specific size and data type of parameter values. A final **v** character, if present, indicates that the command takes a pointer to an array (a vector) of values rather than a series of individual arguments. Two specific examples are:

```
void Uniform4f( int location, float v0, float v1,
   float v2, float v3 );
```

and

```
void GetFloatv( enum pname, float *data );
```

In general, a command declaration has the form

rtype **Name**{ϵ**1234**}{ϵ **b s i i64 f d ub us ui ui64**}{ϵ**v**}
(*[args ,] T arg1, . . ., T argN [, args]*) ;

rtype is the return type of the function. The braces ({}) enclose a series of type descriptors (see table 2.1), of which one is selected. ϵ indicates no type descriptor. The arguments enclosed in brackets (*[args ,]* and *[, args]*) may or may not be present. The N arguments *arg1* through *argN* have type *T*, which corresponds to one of the type descriptors indicated in table 2.1 (if there are no letters, then the arguments' type is given explicitly). If the final character is not **v**, then N is given by the digit **1**, **2**, **3**, or **4** (if there is no digit, then the number of arguments is fixed). If the final character is **v**, then only *arg1* is present and it is an array of N values of the indicated type.

For example,

```
void Uniform{1234}{if}( int location, T value );
```

indicates the eight declarations

```
void Uniform1i( int location, int value );
void Uniform1f( int location, float value );
void Uniform2i( int location, int v0, int v1 );
void Uniform2f( int location, float v0, float v1 );
void Uniform3i( int location, int v0, int v1, int v2 );
void Uniform3f( int location, float v0, float v1,
   float v3 );
```

Type Descriptor	Corresponding GL Type
b	`byte`
s	`short`
i	`int`
i64	`int64`
f	`float`
d	`double`
ub	`ubyte`
us	`ushort`
ui	`uint`
ui64	`uint64`

Table 2.1: Correspondence of command suffix type descriptors to GL argument types. Refer to table 2.2 for definitions of the GL types.

```
void Uniform4i( int location, int v0, int v1, int v2,
   int v3 );
void Uniform4f( int location, float v0, float v1,
   float v2, float v3 );
```

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of the GL data types summarized in table 2.2, or pointers to one of these types. Since many GL operations represent bitfields within these types, transfer blocks of data in these types to graphics hardware which uses the same data types, or otherwise requires these sizes, it is not possible to implement the GL API on an architecture which cannot satisfy the exact bit width requirements in table 2.2.

The types `clampf` and `clampd` are no longer used, replaced by `float` and `double` respectively together with specification language requiring parameter clamping[1].

2.2.1 Data Conversion For State-Setting Commands

Many GL commands specify a value or values to which GL state of a specific type (boolean, enum, integer, or floating-point) is to be set. When multiple versions of such a command exist, using the type descriptor syntax described above, any such version may be used to set the state value. When state values are specified using

[1]These changes are backwards-compatible at the compilation and linking levels, and are being propagated to man pages and header files as well.

GL Type	Bit Width	Description
`boolean`	8	Boolean
`byte`	8	Signed two's complement binary integer
`ubyte`	8	Unsigned binary integer
`char`	8	Characters making up strings
`short`	16	Signed two's complement binary integer
`ushort`	16	Unsigned binary integer
`int`	32	Signed two's complement binary integer
`uint`	32	Unsigned binary integer
`fixed`	32	Signed two's complement 16.16 scaled integer
`int64`	64	Signed two's complement binary integer
`uint64`	64	Unsigned binary integer
`sizei`	32	Non-negative binary integer size
`enum`	32	Enumerated binary integer value
`intptr`	*ptrbits*	Signed twos complement binary integer
`sizeiptr`	*ptrbits*	Non-negative binary integer size
`sync`	*ptrbits*	Sync object handle (see section 4.1)
`bitfield`	32	Bit field
`half`	16	Half-precision floating-point value encoded in an unsigned scalar
`float`	32	Floating-point value
`clampf`	32	Floating-point value clamped to $[0, 1]$
`double`	64	Floating-point value
`clampd`	64	Floating-point value clamped to $[0, 1]$

Table 2.2: GL data types. GL types are not `C` types. Thus, for example, GL type `int` is referred to as `GLint` outside this document, and is not necessarily equivalent to the `C` type `int`. An implementation must use exactly the number of bits indicated in the table to represent a GL type.
ptrbits is the number of bits required to represent a pointer type; in other words, types `intptr`, `sizeiptr`, and `sync` must be large enough to store any CPU address. `sync` is defined as an anonymous struct pointer in the `C` language bindings while `intptr` and `sizeiptr` are defined as integer types large enough to hold a pointer.

a different parameter type than the actual type of that state, data conversions are performed as follows:

- When the type of internal state is boolean, zero integer or floating-point values are converted to `FALSE` and non-zero values are converted to `TRUE`.
- When the type of internal state is integer or enum, boolean values of `FALSE` and `TRUE` are converted to 0 and 1, respectively. Floating-point values are rounded to the nearest integer. If the resulting value is so large in magnitude that it cannot be represented by the internal state variable, the internal state value is undefined.
- When the type of internal state is floating-point, boolean values of `FALSE` and `TRUE` are converted to 0.0 and 1.0, respectively. Integer values are converted to floating-point, with or without normalization as described for specific commands.

For commands taking arrays of the specified type, these conversions are performed for each element of the passed array.

Each command following these conversion rules refers back to this section. Some commands have additional conversion rules specific to certain state values and data types, which are described following the reference.

Validation of values performed by state-setting commands is performed after conversion, unless specified otherwise for a specific command.

2.2.2 Data Conversions For State Query Commands

Query commands (commands whose name begins with **Get**) return a value or values to which GL state has been set. Some of these commands exist in multiple versions returning different data types. When a query command is issued that returns data types different from the actual type of that state, data conversions are performed as follows. If more than one step is applicable, all relevant steps are applied in the following order:

- If a command returning boolean data is called, such as **GetBooleanv**, a floating-point or integer value converts to `FALSE` if and only if it is zero. Otherwise it converts to `TRUE`.
- If a command returning unsigned integer data is called, such as **GetSamplerParameterIuiv**, negative values are clamped to zero.

- If a command returning signed or unsigned integer data is called, such as **GetIntegerv** or **GetInteger64v**, a boolean value of `TRUE` or `FALSE` is interpreted as one or zero, respectively. A floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a normal coordinate, a **DepthRange** value, or a depth buffer clear value. In these cases, the query command converts the floating-point value to an integer according to the `INT` entry of table 18.2; a value not in $[-1, 1]$ converts to an undefined value.

- If a command returning floating-point data is called, such as **GetFloatv** or **GetDoublev**, a boolean value of `TRUE` or `FALSE` is interpreted as 1.0 or 0.0, respectively. An integer value is coerced to floating-point. Single- and double-precision floating-point values are converted as necessary.

Following these steps, if a value is so large in magnitude that it cannot be represented by the returned data type, then the nearest value representable using that type is returned.

When querying bitmasks (such as `SAMPLE_MASK_VALUE` or `STENCIL_WRITEMASK`) with **GetIntegerv**, the mask value is treated as a signed integer, so that mask values with the high bit set will not be clamped when returned as signed integers.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two **DepthRange** parameters are returned in the order *n* followed by *f*. Similarly, points for evaluator maps are returned in the order that they appeared when passed to **Map1**. **Map2** returns $\mathbf{R}_{ij}$ in the $[(uorder)i + j]$th block of values (see page 686 for i, j, $uorder$, and $\mathbf{R}_{ij}$).

2.3 Command Execution

Most of the Specification discusses the behavior of a single context bound to a single *CPU thread*. It is also possible for multiple contexts to share GL objects and for each such context to be bound to a different thread. This section introduces concepts related to GL command execution including error reporting, command queue flushing, and synchronization between command streams. Using these tools can increase performance and utilization of the GPU by separating loosely related tasks into different contexts.

Methods to create, manage, and destroy CPU threads are defined by the host CPU operating system and are not described in the Specification. Binding of GL

contexts to CPU threads is controlled through a window system binding layer such as those described in section 1.3.6.

2.3.1 Errors

The GL detects only a subset of those conditions that could be considered errors. This is because in many cases error checking would adversely impact the performance of an error-free program.

The command

```
enum GetError( void );
```

is used to obtain error information. Each detectable error is assigned a numeric code. When an error is detected, a flag is set and the code is recorded. Further errors, if they occur, do not affect this recorded code. When **GetError** is called, the code is returned and the flag is cleared, so that a further error will again record its code. If a call to **GetError** returns NO_ERROR, then there has been no detectable error since the last call to **GetError** (or since the GL was initialized).

To allow for distributed implementations, there may be several flag-code pairs. In this case, after a call to **GetError** returns a value other than NO_ERROR each subsequent call returns the non-zero code of a distinct flag-code pair (in unspecified order), until all non-NO_ERROR codes have been returned. When there are no more non-NO_ERROR error codes, all flags are reset. This scheme requires some positive number of pairs of a flag bit and an integer. The initial state of all flags is cleared and the initial value of all codes is NO_ERROR.

Table 2.3 summarizes GL errors. Currently, when an error flag is set, results of GL operation are undefined only if an OUT_OF_MEMORY error has occurred. In other cases, there are no side effects unless otherwise noted; the command which *generates* the error is ignored so that it has no effect on GL state or framebuffer contents. Except as otherwise noted, if the generating command returns a value, it returns zero. If the generating command modifies values through a pointer argument, no change is made to these values.

These error semantics apply only to GL errors, not to system errors such as memory access errors. This behavior is the current behavior; the action of the GL in the presence of errors is subject to change, and extensions to OpenGL may define behavior currently considered as an error.

Several error generation conditions are implicit in the description of every GL command[2].

[2]For historical reasons, some older errors do not follow the patterns described below, and cannot easily be changed. The explicit error language with each command always controls behavior when in conflict with this section.

- If the GL context has been reset as a result of previous GL command, or if the context is reset as a side effect of execution of a command, a CONTEXT_LOST error is generated.
- If a command that requires an enumerated value is passed a symbolic constant that is never allowable for that command, an INVALID_ENUM error is generated. This is the case even if the argument is a pointer to a symbolic constant, if the value or values pointed to are not allowable for the given command. In some cases, a symbolic constant is allowable for a command, but is forbidden in combination with current GL state and/or a value passed for another parameter of that command. These cases are documented explicitly, and an INVALID_OPERATION error is generated instead.
- If a negative number is provided where an argument of type sizei or sizeiptr is specified, an INVALID_VALUE error is generated.
- If memory is exhausted as a side effect of the execution of a command, an OUT_OF_MEMORY error may be generated.

The Specification attempts to explicitly describe these implicit error conditions (with the exception of CONTEXT_LOST[3] and OUT_OF_MEMORY[4] wherever they apply. However, they apply even if not explicitly described, unless a specific command describes different behavior. For example, certain commands use a sizei parameter to indicate the length of a string, and also use negative values of the parameter to indicate a null-terminated string. These commands do not generate an INVALID_VALUE error, because they explicitly describe different behavior.

Otherwise, errors are generated only for conditions that are explicitly described in the Specification.

When a command could potentially generate several different errors (for example, when it is passed separate enum and numeric parameters which are both out of range), the GL implementation may choose to generate any of the applicable errors.

Errors based solely on one or more argument values to a command must be detected before any processing based on current state[5].

[3] CONTEXT_LOST is not described because it can potentially be generated by almost all GL commands, and occurs for reasons not directly related to the affected commands.

[4] OUT_OF_MEMORY is not described because it can potentially be generated by any GL command, even those which do not explicitly allocate GPU memory.

[5] This ensures consistent behavior for commands including language which ignores certain parameters under some conditions, such as **glBlitFramebuffer** treatment of *mask* and *filter*.

Error	Description	Offending command ignored?
CONTEXT_LOST	Context has been lost and reset by the driver	Except as noted for specific commands
INVALID_ENUM	enum argument out of range	Yes
INVALID_VALUE	Numeric argument out of range	Yes
INVALID_OPERATION	Operation illegal in current state	Yes
INVALID_FRAMEBUFFER_OPERATION	Framebuffer object is not complete	Yes
OUT_OF_MEMORY	Not enough memory left to execute command	Unknown
STACK_OVERFLOW	Command would cause a stack overflow	Yes
STACK_UNDERFLOW	Command would cause a stack underflow	Yes
TABLE_TOO_LARGE	The specified table is too large	Yes

Table 2.3: Summary of GL errors

When an error is generated, the GL may also generate a debug output message describing its cause (see chapter 20). The message has *source* DEBUG_SOURCE_API, *type* DEBUG_TYPE_ERROR, and an implementation-dependent ID.

Most commands include a complete summary of errors at the end of their description, including even the implicit errors described above.

Such error summaries are set in a distinct style, like this sentence.

In some cases, however, errors may be generated for a single command for reasons not directly related to that command. One such example is that deferred processing for shader programs may result in link errors detected only when attempting to draw primitives using vertex specification commands. In such cases, errors generated by a command may be described elsewhere in the specification than the command itself.

2.3.1.1 No Error Mode

If the GL context was created with the no error mode enabled, then any place where the driver would have generated an error instead has undefined behavior. This could include application termination. All calls to **GetError** will return NO_ERROR or OUT_OF_MEMORY. OUT_OF_MEMORY errors are a special case because they already allow for undefined behavior and are more difficult for application developers to predict than other errors. OUT_OF_MEMORY errors may be delayed, which can be useful for optimizing multithreaded drivers, but eventually the OUT_OF_MEMORY error will be reported if an implementation would have reported this error. Since behavior of OUT_OF_MEMORY errors are undefined there is some implementation flexibility here. However, this behavior may provide useful information on some implementations that do report OUT_OF_MEMORY without crashing. [6]

2.3.2 Graphics Reset Recovery

Certain events can result in a reset of the GL context. After such an event, it is referred to as a *lost context* and is unusable for almost all purposes. Recovery requires creating a new context and recreating all relevant state from the lost context. The current status of the graphics reset state is returned by

```
enum GetGraphicsResetStatus( void );
```

[6] Jon - "should" be reported?

The value returned indicates if the GL context has been in a reset state at any point since the last call to **GetGraphicsResetStatus**:

- NO_ERROR indicates that the GL context has not been in a reset state since the last call.
- GUILTY_CONTEXT_RESET indicates that a reset has been detected that is attributable to the current GL context.
- INNOCENT_CONTEXT_RESET indicates a reset has been detected that is not attributable to the current GL context.
- UNKNOWN_CONTEXT_RESET indicates a detected graphics reset whose cause is unknown.

If a reset status other than NO_ERROR is returned and subsequent calls return NO_ERROR, the context reset was encountered and completed. If a reset status is repeatedly returned, the context may be in the process of resetting.

Reset notification behavior is determined at context creation time, and may be queried by calling **GetIntegerv** with *pname* RESET_NOTIFICATION_STRATEGY.

If the reset notification behavior is NO_RESET_NOTIFICATION, then the implementation will never deliver notification of reset events, and **GetGraphicsResetStatus** will always return NO_ERROR[7].

If the behavior is LOSE_CONTEXT_ON_RESET, a graphics reset will result in the loss of all context state, requiring the recreation of all associated objects. In this case **GetGraphicsResetStatus** may return any of the values described above.

If a graphics reset notification occurs in a context, a notification must also occur in all other contexts which share objects with that context[8].

After a graphics reset has occurred on a context, subsequent GL commands on that context (or any context which shares with that context) will generate a CONTEXT_LOST error. Such commands will not have side effects (in particular, they will not modify memory passed by pointer for query results), and may not block indefinitely or cause termination of the application. Exceptions to this behavior include:

- **GetError** and **GetGraphicsResetStatus** behave normally following a graphics reset, so that the application can determine a reset has occurred, and when it is safe to destroy and re-create the context.

[7] In this case, it is recommended that implementations should not allow loss of context state no matter what events occur. However, this is only a recommendation, and cannot be relied upon by applications.

[8] The values returned by **GetGraphicsResetStatus** in the different contexts may differ.

- Any commands which might cause a polling application to block indefinitely will generate a `CONTEXT_LOST` error, but will also return a value indicating completion to the application. Such commands include:
  - **GetSynciv** with *pname* `SYNC_STATUS` ignores the other parameters and returns `SIGNALED` in *values*.
  - **GetQueryObjectuiv** with *pname* `QUERY_RESULT_AVAILABLE` ignores the other parameters and returns `TRUE` in *params*.

2.3.3 Flush and Finish

Implementations may buffer multiple commands in a *command queue* before sending them to the GL server for execution. This may happen in places such as the network stack (for network transparent implementations), CPU code executing as part of the GL client or the GL server, or internally to the GPU hardware. Coarse control over command queues is available using the command

```
void Flush( void );
```

which causes all previously issued GL commands to complete in finite time[9] (although such commands may still be executing when **Flush** returns).

The command

```
void Finish( void );
```

forces all previously issued GL commands to complete. **Finish** does not return until all effects from such commands on GL client and server state and the framebuffer are fully realized.

Finer control over command execution can be expressed using fence commands and sync objects, as discussed in section 4.1.

2.3.4 Numeric Representation and Computation

The GL must perform a number of floating-point operations during the course of its operation.

[9] Historically, use of **Flush** has had negative performance implications for some implementations, and subsequently it does not universally operate as described here. One exception to this is when the default framebuffer is bound, and it is single-buffered; in this case, flush behaves as expected. Other references to the flush operation in the specification, such as that in section 4.1.2, will behave as expected. Waiting on a fence sync object with `SYNC_FLUSH_COMMANDS_BIT` is thus recommended as a way to perform a guaranteed flush.

Implementations normally perform computations in floating-point, and must meet the range and precision requirements defined under **"Floating-Point Computation"** below.

These requirements only apply to computations performed in GL operations outside of shader execution, such as texture image specification and sampling, and per-fragment operations. Range and precision requirements during shader execution differ and are specified by the OpenGL Shading Language Specification.

In some cases, the representation and/or precision of operations is implicitly limited by the specified format of vertex, texture, or renderbuffer data consumed by the GL. Specific floating-point formats are described later in this section.

2.3.4.1 Floating-Point Computation

We do not specify how floating-point numbers are to be represented, or the details of how operations on them are performed.

We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5. The maximum representable magnitude for all floating-point values must be at least 2^{32}. $x \cdot 0 = 0 \cdot x = 0$ for any non-infinite and non-NaN x. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

The special values Inf and $-Inf$ encode values with magnitudes too large to be represented; the special value NaN encodes "Not A Number" values resulting from undefined arithmetic operations such as $\frac{0}{0}$. Implementations are permitted, but not required, to support Infs and NaNs in their floating-point computations.

Any representable floating-point value is legal as input to a GL command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to GL interruption or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing a NaN or an infinity yields unspecified results.

2.3.4.2 16-Bit Floating-Point Numbers

A 16-bit floating-point number has a 1-bit sign (S), a 5-bit exponent (E), and a 10-bit mantissa (M). The value V of a 16-bit floating-point number is determined by the following:

$$V = \begin{cases} (-1)^S \times 0.0, & E = 0, M = 0 \\ (-1)^S \times 2^{-14} \times \frac{M}{2^{10}}, & E = 0, M \neq 0 \\ (-1)^S \times 2^{E-15} \times \left(1 + \frac{M}{2^{10}}\right), & 0 < E < 31 \\ (-1)^S \times Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

If the floating-point number is interpreted as an unsigned 16-bit integer N, then

$$S = \left\lfloor \frac{N \bmod 65536}{32768} \right\rfloor$$
$$E = \left\lfloor \frac{N \bmod 32768}{1024} \right\rfloor$$
$$M = N \bmod 1024.$$

Any representable 16-bit floating-point value is legal as input to a GL command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.

2.3.4.3 Unsigned 11-Bit Floating-Point Numbers

An unsigned 11-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 6-bit mantissa (M). The value V of an unsigned 11-bit floating-point number is determined by the following:

$$V = \begin{cases} 0.0, & E = 0, M = 0 \\ 2^{-14} \times \frac{M}{64}, & E = 0, M \neq 0 \\ 2^{E-15} \times \left(1 + \frac{M}{64}\right), & 0 < E < 31 \\ Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

If the floating-point number is interpreted as an unsigned 11-bit integer N, then

$$E = \left\lfloor \frac{N}{64} \right\rfloor$$
$$M = N \bmod 64.$$

When a floating-point value is converted to an unsigned 11-bit floating-point representation, finite values are rounded to the closest representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 65024 (the maximum finite representable unsigned 11-bit floating-point value) are converted to 65024. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 11-bit floating-point value is legal as input to a GL command that accepts 11-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number to GL must yield predictable results.

2.3.4.4 Unsigned 10-Bit Floating-Point Numbers

An unsigned 10-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 5-bit mantissa (M). The value V of an unsigned 10-bit floating-point number is determined by the following:

$$V = \begin{cases} 0.0, & E = 0, M = 0 \\ 2^{-14} \times \frac{M}{32}, & E = 0, M \neq 0 \\ 2^{E-15} \times \left(1 + \frac{M}{32}\right), & 0 < E < 31 \\ Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

If the floating-point number is interpreted as an unsigned 10-bit integer N, then

$$E = \left\lfloor \frac{N}{32} \right\rfloor$$
$$M = N \bmod 32.$$

When a floating-point value is converted to an unsigned 10-bit floating-point representation, finite values are rounded to the closest representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 64512 (the maximum finite representable unsigned 10-bit floating-point value) are converted to 64512. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 10-bit floating-point value is legal as input to a GL command that accepts 10-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number to GL must yield predictable results.

2.3.4.5 Fixed-Point Computation

Vertex attributes may be specified using a 32-bit two's-complement signed representation with 16 bits to the right of the binary point (fraction bits).

2.3.4.6 General Requirements

Some calculations require division. In such cases (including implied divisions required by vector normalizations), a division by zero produces an unspecified result but must not lead to GL interruption or termination.

2.3.5 Fixed-Point Data Conversions

When generic vertex attributes and pixel color or depth components are represented as integers, they are often (but not always) considered to be *normalized*. Normalized integer values are treated specially when being converted to and from floating-point values, and are usually referred to as *normalized fixed-point*. Such values are always either *signed* or *unsigned*.

In the remainder of this section, b denotes the bit width of the fixed-point integer representation. When the integer is one of the types defined in table 2.2, b is the required bit width of that type. When the integer is a texture or renderbuffer color or depth component (see section 8.5), b is the number of bits allocated to that component in the internal format of the texture or renderbuffer. When the integer is a framebuffer color or depth component (see section 9), b is the number of bits allocated to that component in the framebuffer. For framebuffer and renderbuffer alpha components, b must be at least 2 if the buffer does not contain an alpha component, or if there is only one bit of alpha in the buffer.

The signed and unsigned fixed-point representations are assumed to be b-bit binary twos-complement integers and binary unsigned integers, respectively.

2.3.5.1 Conversion from Normalized Fixed-Point to Floating-Point

Unsigned normalized fixed-point integers represent numbers in the range $[0, 1]$. The conversion from an unsigned normalized fixed-point value c to the correspond-

ing floating-point value f is defined as

$$f = \frac{c}{2^b - 1}. \tag{2.1}$$

Signed normalized fixed-point integers represent numbers in the range $[-1, 1]$. The conversion from a signed normalized fixed-point value c to the corresponding floating-point value f is performed using

$$f = \max\left\{\frac{c}{2^{b-1} - 1}, -1.0\right\}. \tag{2.2}$$

Only the range $[-2^{b-1} + 1, 2^{b-1} - 1]$ is used to represent signed fixed-point values in the range $[-1, 1]$. For example, if $b = 8$, then the integer value -127 corresponds to -1.0 and the value 127 corresponds to 1.0. Note that while zero can be exactly expressed in this representation, one value (-128 in the example) is outside the representable range, and must be clamped before use. This equation is used everywhere that signed normalized fixed-point values are converted to floating-point, including for all signed normalized fixed-point parameters in GL commands, such as vertex attribute values[10], as well as for texture sampling or framebuffer values used for blending.

2.3.5.2 Conversion from Floating-Point to Normalized Fixed-Point

The conversion from a floating-point value f to the corresponding unsigned normalized fixed-point value c is defined by first clamping f to the range $[0, 1]$, then computing

$$f' = convert_float_uint(f \times (2^b - 1), b) \tag{2.3}$$

where $convert_float_uint(r, b)$ returns one of the two unsigned binary integer values with exactly b bits which are closest to the floating-point value r (where rounding to nearest is preferred).

The conversion from a floating-point value f to the corresponding signed normalized fixed-point value c is performed by clamping f to the range $[-1, 1]$, then computing

$$f' = convert_float_int(f \times (2^{b-1} - 1), b) \tag{2.4}$$

[10] This is a behavior change in OpenGL 4.2. In previous versions, a different conversion for signed normalized values was used in which -128 mapped to -1.0, 127 mapped to 1.0, and 0.0 was not exactly representable.

where $convert_float_int(r, b)$ returns one of the two signed two's-complement binary integer values with exactly b bits which are closest to the floating-point value r (where rounding to nearest is preferred).

This equation is used everywhere that floating-point values are converted to signed normalized fixed-point, including when querying floating-point state (see section 2.2.2) and returning integers[11], as well as for specifying signed normalized texture or framebuffer values using floating-point.

2.4 Rendering Commands

GL commands performing rendering into a framebuffer are sometimes treated specially by other GL operations such as conditional rendering (see section 10.9). Such commands are called *rendering commands*, and include the *drawing commands* ***Draw*** (see section 10.4) and **Begin / End** (see section 10.7), as well as these additional commands:

- **Accum** (see section 17.4.5)
- **Bitmap** (see section 14.8)
- **BlitFramebuffer** (see section 18.3.2)
- **Clear** (see section 17.4.3)
- **ClearBuffer*** (see section 17.4.3.1)
- **CopyPixels** (see section 18.3)
- **DispatchCompute*** (see section 19)
- **DrawPixels** (see section 18.1)
- **EvalMesh1** and **EvalMesh2** (see section 21.1)

2.5 Context State

Context state is state that belongs to the GL context as a whole, rather than to instances of the different object types described in section 2.6. Context state controls fixed-function stages of the GPU, such as clipping, primitive rasterization, and

[11] This is a behavior change in OpenGL 4.2. In previous versions, a different conversion for signed normalized values was used in which -1.0 mapped to -128, 1.0 mapped to 127, and 0.0 was not exactly representable.

framebuffer clears, and also specifies *bindings* of objects to the context specifying which objects are used during command execution.

Context state also controls the fixed-function stages described as legacy alternatives to shaders for performing transformation, lighting, and texturing.

The Specification describes all visible context state variables and describes how each one can be changed. State variables are grouped somewhat arbitrarily by their function. Although we describe operations that the GL performs on the framebuffer, the framebuffer is not a part of GL state.

There are two types of context state. *Server state* resides in the GL server; the majority of GL state falls into this category. *Client state* resides in the GL client. Unless otherwise specified, all state is server state; client state is specifically identified. Each instance of a context includes a complete set of server state; each connection from a client to a server also includes a complete set of client state.

While an implementation of OpenGL may be hardware dependent, the Specification is independent of any specific hardware on which it is implemented. We are concerned with the state of graphics hardware only when it corresponds precisely to GL state.

2.5.1 Generic Context State Queries

Context state queries are described in detail in chapter 22.

2.6 Objects and the Object Model

Many types of *objects* are defined in the remainder of the Specification. Applications may create, modify, query, and destroy many *instances* of each of these object types, limited in most cases only by available graphics memory. Specific instances of different object types are *bound* to a context. The set of bound objects define the shaders which are invoked by GL drawing operations; specify the buffer data, texture image, and framebuffer memory that is accessed by shaders and directly by GL commands; and contain the state used by other operations such as fence synchronization and timer queries.

Each object type corresponds to a distinct set of commands which manage objects of that type. However, there is an object model describing how most types of objects are managed, described below. Exceptions to the object model for specific object types are described later in the Specification together with those object types.

Following the description of the object model, each type of object is briefly described below, together with forward references to full descriptions of that ob-

ject type in later chapters of the Specification. Objects are described in an order corresponding to the structure of the remainder of the Specification.

2.6.1 Object Management

2.6.1.1 Name Spaces, Name Generation, and Object Creation

Each object type has a corresponding *name space*. Names of objects are represented by unsigned integers of type `uint`. The name zero is reserved by the GL; for some object types, zero names a *default object* of that type, and in others zero will never correspond to an actual instance of that object type.

Names of most types of objects are created by *generating* unused names using commands starting with **Gen** followed by the object type. For example, the command **GenBuffers** returns one or more previously unused buffer object names.

Generated names are marked by the GL as used, for the purpose of name generation only. Object names marked in this fashion will not be returned by additional calls to generate names of the same type until the names are marked unused again by deleting them (see below).

Generated names do not initially correspond to an instance of an object. Objects with generated names are created by binding a generated name to the context. For example, a buffer object is created by calling the command **BindBuffer** with a name returned by **GenBuffers**, which allocates resources for the buffer object and its state, and associate the name with that object. Sampler objects may also be created by commands in addition to **BindSampler**, as described in section 8.2.

Objects may also be created directly by functions that return a new name or names representing a freshly initialized object. Some functions return a single object name directly whereas others are able to create a large number of new objects, returning their names in an array. Examples of the former are **CreateProgram** for program objects and **FenceSync** for fence sync objects. Examples of the latter are **CreateBuffers**, **CreateTextures** and **CreateVertexArrays** for buffers, textures and vertex arrays, respectively.

2.6.1.2 Name Deletion and Object Deletion

Objects are deleted by calling deletion commands specific to that object type. For example, the command **DeleteBuffers** is passed an array of buffer object names to delete. After an object is deleted it has no contents, and its name is once again marked unused for the purpose of name generation. If names are deleted that do not correspond to an object, but have been marked for the purpose of name generation, such names are marked as unused again. If unused and unmarked names are deleted they are silently ignored, as is the name zero.

If an object is deleted while it is currently in use by a GL context, its name is immediately marked as unused, and some types of objects are automatically unbound from binding points in the current context, as described in section 5.1.2. However, the actual underlying object is not deleted until it is no longer in use. This situation is discussed in more detail in section 5.1.3.

2.6.1.3 Shared Object State

It is possible for groups of contexts to share some server state. Enabling such sharing between contexts is done through window system binding APIs such as those described in section 1.3.6. These APIs are responsible for creation and management of contexts, and are not discussed further here. More detailed discussion of the behavior of shared objects is included in chapter 5. Except as defined below for specific object types, all state in a context is specific to that context only.

2.6.2 Buffer Objects

The GL uses many types of data supplied by the client. Some of this data must be stored in server memory, and it is desirable to store other types of frequently used client data, such as vertex array and pixel data, in server memory for performance reasons, even if the option to store it in client memory exists.

Buffer objects contain a *data store* holding a fixed-sized allocation of server memory, and provide a mechanism to allocate, initialize, read from, and write to such memory. Under certain circumstances, the data store of a buffer object may be shared between the client and server and accessed simultaneously by both.

Buffer objects may be shared. They are described in detail in chapter 6.

2.6.3 Shader Objects

The source and/or binary code representing part or all of a shader program that is executed by one of the programmable stages defined by the GL (such as a vertex or fragment shader) is encapsulated in one or more *shader objects*.

Shader objects may be shared. They are described in detail in chapter 7.

2.6.4 Program Objects

Shader objects that are to be used by one or more of the programmable stages of the GL are linked together to form a *program object*. The shader programs that are executed by these programmable stages are called *executables*. All information necessary for defining each executable is encapsulated in a program object.

Program objects may be shared. They are described in detail in chapter 7.

2.6.5 Program Pipeline Objects

Program pipeline objects contain a separate program object binding point for each programmable stage. They allow a primitive to be processed by independent programs in each programmable stage, instead of requiring a single program object for each combination of shader operations. They allow greater flexibility when combining different shaders in various ways, without requiring a program object for each such combination.

Program pipeline objects are *container objects* including references to program objects, and are not shared. They are described in detail in chapter 7.

2.6.6 Texture Objects

Texture objects or *textures* include a data store containing a collection of *texture images* built from arrays of image elements. The image elements are referred to as *texels*. There are many types of texture objects varying by dimensionality and structure; the different texture types are described in detail in the introduction to chapter 8.

Texture objects also include state describing the image parameters of the texture images, and state describing how sampling is performed when a shader accesses a texture.

Shaders may *sample* a texture at a location indicated by specified *texture coordinates*, with details of sampling determined by the sampler state of the texture. The resulting texture samples are typically used to modify a fragment's color, in order to map an image onto a geometric primitive being drawn, but may be used for any purpose in a shader.

Texture objects may be shared. They are described in detail in chapter 8.

2.6.7 Sampler Objects

Sampler objects contain the subset of texture object state controlling how sampling is performed when a shader accesses a texture. Sampler and texture objects may be bound together so that the sampler object state is used by shaders when sampling the texture, overriding equivalent state in the texture object. Separating texture image data from the method of sampling that data allows reuse of the same sampler state with many different textures without needing to set the sampler state in each texture.

Sampler objects may be shared. They are described in detail in chapter 8.

2.6.8 Renderbuffer Objects

Renderbuffer objects contain a single image in a format which can be rendered to. Renderbuffer objects are attached to framebuffer objects (see below) when performing *off-screen rendering*.

Renderbuffer objects may be shared. They are described in detail in chapter 9.

2.6.9 Framebuffer Objects

Framebuffer objects encapsulate the state of a framebuffer, including a collection of color, depth, and stencil buffers. Each such buffer is represented by a renderbuffer object or texture object *attached* to the framebuffer object.

Framebuffer objects are container objects including references to renderbuffer and/or texture objects, and are not shared[12]. They are described in detail in chapter 9.

2.6.10 Vertex Array Objects

Vertex array objects represent a collection of sets of *vertex attribute*s. Each set is stored as an array in a buffer object data store, with each element of the array having a specified format and component count. The attributes of the currently bound vertex array object are used as inputs to the vertex shader when executing drawing commands.

Vertex array objects are container objects including references to buffer objects, and are not shared. They are described in detail in chapter 10.

2.6.11 Transform Feedback Objects

Transform feedback objects are used to capture attributes of the vertices of transformed primitives passed to the transform feedback stage when *transform feedback mode* is active. They include state required for transform feedback together with references to buffer objects in which attributes are captured.

Transform feedback objects are container objects including references to buffer objects, and are not shared. They are described in detail in section 13.3.1.

[12] Framebuffer objects created with the commands defined by the `GL_EXT_framebuffer_object` extension are defined to be shared, while FBOs created with commands defined by the OpenGL core or `GL_ARB_framebuffer_object` extension are defined to **not** be shared. Undefined behavior results when using FBOs created by EXT commands through non-EXT interfaces, or vice-versa.

2.6.12 Query Objects

Query objects return information about the processing of a sequence of GL commands, such as the number of primitives processed by drawing commands; the number of primitives written to transform feedback buffers; the number of samples that pass the depth test during fragment processing; and the amount of time required to process commands.

Query objects are not shared. They are described in detail in section 4.2.

2.6.13 Sync Objects

A *sync object* acts as a *synchronization primitive* – a representation of events whose completion status can be tested or waited upon. Sync objects may be used for synchronization with operations occurring in the GL state machine or in the graphics pipeline, and for synchronizing between multiple graphics contexts, among other purposes.

Sync objects may be shared. They are described in detail in section 4.1.

2.6.14 Display Lists

A *display list* is an object representing a group of GL commands and their parameters which have been stored in server memory for subsequent execution. This may be desirable for performance reasons when the GL client and server are separated by a network, or when the CPU to GPU interface is too slow to keep the GPU fully fed. Most of the benefits of display lists can be more easily achieved in modern graphics architectures by storing as much data as possible in GL objects in server memory, and display lists have been deprecated.

Display lists may be shared. They are described in detail in chapter 21.4.

Chapter 3

Dataflow Model

Figure 3.1 shows a block diagram of the GL. Some commands specify geometric objects to be drawn while others specify state controlling how objects are handled by the various stages, or specify data contained in textures and buffer objects. Most commands may be accumulated in a *display list* for processing by the GL at a later time. Otherwise, commands are effectively sent through a processing pipeline. Different stages of the pipeline use data contained in different types of buffer objects. Some pipeline stages may use data in client memory instead of, or in addition to data in buffer objects. Data in client memory are not shown in the block diagram, but are described later in the specification as part of the description of those pipeline stages.

The first stage assembles vertices to form geometric primitives such as points, line segments, and polygons. In the next stage vertices may be transformed and lit, followed by assembly into geometric primitives. Tessellation and geometry shaders may then generate multiple primitives from a single input primitive. Optionally, the results of these pipeline stages may be fed back into buffer objects using transform feedback.

The final resulting primitives are clipped to a clip volume in preparation for the next stage, rasterization. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon. Each *fragment* so produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colors with stored colors, as well as masking, stenciling, and other logical operations on fragment values.

There is a way to bypass the vertex processing portion of the pipeline to send a

block of fragments directly to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer, and pixels may also be read back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

Some additional stages are not shown in figure 3.1. One such stage computes vertices by evaluating polynomial functions of input values, providing an efficient means for approximating curve and surface geometry rather than specifying each vertex explicitly. Another operates on pixel data in the process of reading from or writing to the framebuffer or texture images.

Finally, compute shaders which may read from and write to buffer objects may be executed independently of the pipeline shown in figure 3.1.

This ordering is meant only as a tool for describing the GL, not as a strict rule of how the GL is implemented, and we present it only as a means to organize the various operations of the GL. Objects such as curved surfaces, for instance, may be transformed before they are converted to polygons.

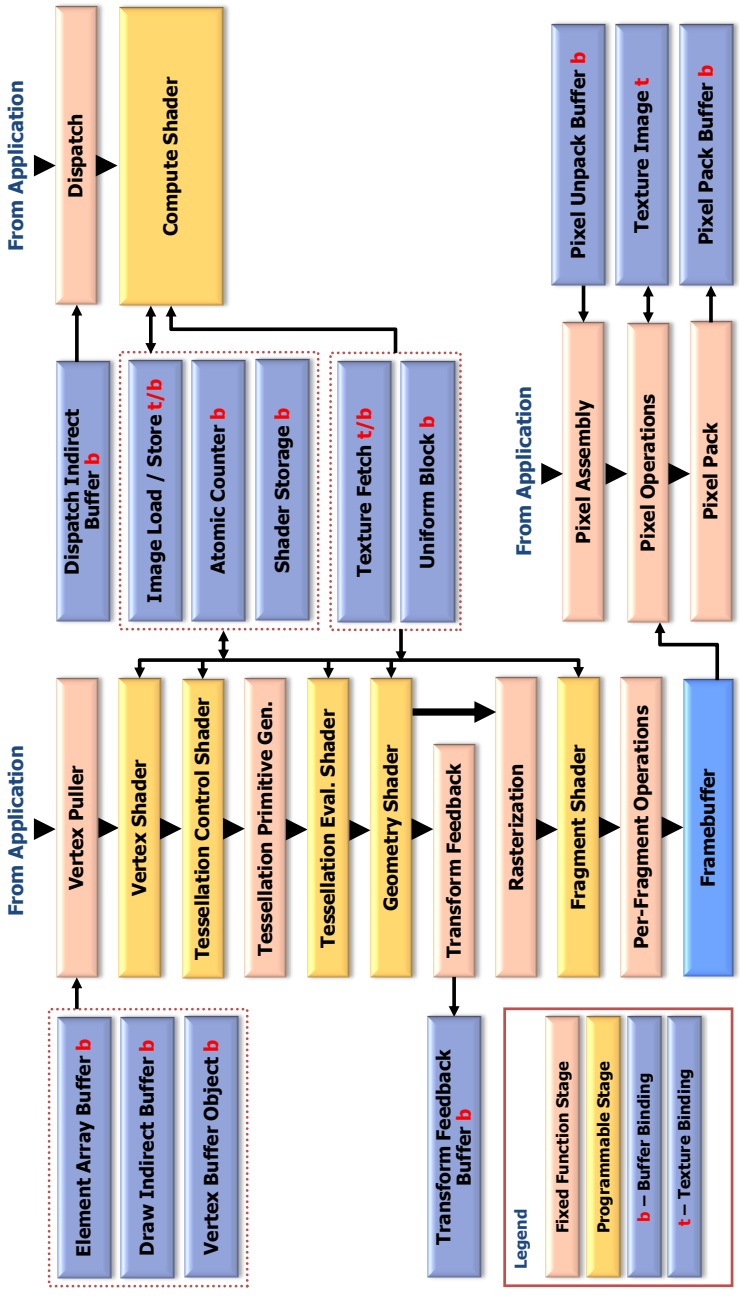


Figure 3.1. Block diagram of the GL pipeline.

Chapter 4

Event Model

4.1 Sync Objects and Fences

A sync object acts as a *synchronization primitive* – a representation of events whose completion status can be tested or waited upon. Sync objects may be used for synchronization with operations occurring in the GL state machine or in the graphics pipeline, and for synchronizing between multiple graphics contexts, among other purposes.

Sync objects have a status value with two possible states: *signaled* and *unsignaled*. Events are associated with a sync object. When a sync object is created, its status is set to unsignaled. When the associated event occurs, the sync object is signaled (its status is set to signaled). The GL may be asked to wait for a sync object to become signaled.

Initially, only one specific type of sync object is defined: the fence sync object, whose associated event is triggered by a fence command placed in the GL command stream. Fence sync objects are used to wait for partial completion of the GL command stream, as a more flexible form of **Finish**.

The command

`sync` **FenceSync**(`enum` *condition*, `bitfield` *flags*);

creates a new fence sync object, inserts a fence command in the GL command stream and associates it with that sync object, and returns a non-zero name corresponding to the sync object.

When the specified *condition* of the sync object is satisfied by the fence command, the sync object is signaled by the GL, causing any **ClientWaitSync** or **WaitSync** commands (see below) blocking on *sync* to *unblock*. No other state is affected by **FenceSync** or by execution of the associated fence command.

Property Name	Property Value
OBJECT_TYPE	SYNC_FENCE
SYNC_CONDITION	*condition*
SYNC_STATUS	UNSIGNALED
SYNC_FLAGS	*flags*

Table 4.1: Initial properties of a sync object created with **FenceSync**.

condition must be SYNC_GPU_COMMANDS_COMPLETE. This condition is satisfied by completion of the fence command corresponding to the sync object and all preceding commands in the same command stream. The sync object will not be signaled until all effects from these commands on GL client and server state and the framebuffer are fully realized. Note that completion of the fence command occurs once the state of the corresponding sync object has been changed, but commands waiting on that sync object may not be unblocked until some time after the fence command completes.

flags must be zero.

Each sync object contains a number of *properties* which determine the state of the object and the behavior of any commands associated with it. Each property has a *property name* and *property value*. The initial property values for a sync object created by **FenceSync** are shown in table 4.1.

Properties of a sync object may be queried with **GetSynciv** (see section 4.1.3). The SYNC_STATUS property will be changed to SIGNALED when *condition* is satisfied.

Errors

If **FenceSync** fails to create a sync object, zero will be returned and a GL error is generated.

An INVALID_ENUM error is generated if *condition* is not SYNC_GPU_COMMANDS_COMPLETE.

An INVALID_VALUE error is generated if *flags* is not zero.

A sync object can be deleted by passing its name to the command

`void` **DeleteSync**(`sync` *sync*);

If the fence command corresponding to the specified sync object has completed, or if no **ClientWaitSync** or **WaitSync** commands are blocking on *sync*, the object is deleted immediately. Otherwise, *sync* is flagged for deletion and will be

deleted when it is no longer associated with any fence command and is no longer blocking any **ClientWaitSync** or **WaitSync** command. In either case, after returning from **DeleteSync** the *sync* name is invalid and can no longer be used to refer to the sync object.

DeleteSync will silently ignore a *sync* value of zero.

Errors

An `INVALID_VALUE` error is generated if *sync* is neither zero nor the name of a sync object.

4.1.1 Waiting for Sync Objects

The command

```
enum ClientWaitSync( sync sync, bitfield flags,
    uint64 timeout );
```

causes the GL to block, and will not return until the sync object *sync* is signaled, or until the specified *timeout* period expires. *timeout* is in units of nanoseconds. *timeout* is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

If *sync* is signaled at the time **ClientWaitSync** is called, then **ClientWaitSync** returns immediately. If *sync* is unsignaled at the time **ClientWaitSync** is called, then **ClientWaitSync** will block and will wait up to *timeout* nanoseconds for *sync* to become signaled. *flags* controls command flushing behavior, and may be `SYNC_FLUSH_COMMANDS_BIT`, as discussed in section 4.1.2.

ClientWaitSync returns one of four status values. A return value of `ALREADY_SIGNALED` indicates that *sync* was signaled at the time **ClientWaitSync** was called. `ALREADY_SIGNALED` will always be returned if *sync* was signaled, even if the value of *timeout* is zero. A return value of `TIMEOUT_EXPIRED` indicates that the specified timeout period expired before *sync* was signaled. A return value of `CONDITION_SATISFIED` indicates that *sync* was signaled before the timeout expired. Finally, if an error occurs, in addition to generating a GL error as specified below, **ClientWaitSync** immediately returns `WAIT_FAILED` without blocking.

If the value of *timeout* is zero, then **ClientWaitSync** does not block, but simply tests the current state of *sync*. `TIMEOUT_EXPIRED` will be returned in this case if *sync* is not signaled, even though no actual wait was performed.

Errors

An INVALID_VALUE error is generated if *sync* is not the name of a sync object.

An INVALID_VALUE error is generated if *flags* contains any bits other than SYNC_FLUSH_COMMANDS_BIT.

The command

```
void WaitSync( sync sync, bitfield flags,
    uint64 timeout );
```

is similar to **ClientWaitSync**, but instead of blocking and not returning to the application until *sync* is signaled, **WaitSync** returns immediately, instead causing the GL server to block[1] until *sync* is signaled[2].

sync has the same meaning as for **ClientWaitSync**.

timeout must currently be the special value TIMEOUT_IGNORED, and is not used. Instead, **WaitSync** will always wait no longer than an implementation-dependent timeout. The duration of this timeout in nanoseconds may be queried by calling **GetInteger64v** with *pname* MAX_SERVER_WAIT_TIMEOUT. There is currently no way to determine whether **WaitSync** unblocked because the timeout expired or because the sync object being waited on was signaled.

flags must be zero.

If an error occurs, **WaitSync** generates a GL error as specified below, and does not cause the GL server to block.

Errors

An INVALID_VALUE error is generated if *sync* is not the name of a sync object.

An INVALID_VALUE error is generated if *timeout* is not TIMEOUT_IGNORED or *flags* is not zero[a].

[a] *flags* and *timeout* are placeholders for anticipated future extensions of sync object capabilities. They must have these reserved values in order that existing code calling **WaitSync** operate properly in the presence of such extensions.

[1] The GL server may choose to wait either in the CPU executing server-side code, or in the GPU hardware if it supports this operation.

[2] **WaitSync** allows applications to continue to queue commands from the client in anticipation of the sync being signaled, increasing client-server parallelism.

4.1.1.1 Multiple Waiters

It is possible for both the GL client to be blocked on a sync object in a **ClientWaitSync** command, the GL server to be blocked as the result of a previous **WaitSync** command, and for additional **WaitSync** commands to be queued in the GL server, all for a single sync object. When such a sync object is signaled in this situation, the client will be unblocked, the server will be unblocked, and all such queued **WaitSync** commands will continue immediately when they are reached.

See section 5.2 for more information about blocking on a sync object in multiple GL contexts.

4.1.2 Signaling

A fence sync object enters the signaled state only once the corresponding fence command has completed and signaled the sync object.

If the sync object being blocked upon will not be signaled in finite time (for example, by an associated fence command issued previously, but not yet flushed to the graphics pipeline), then **ClientWaitSync** may hang forever. To help prevent this behavior[3], if **ClientWaitSync** is called and all of the following are true:

- the `SYNC_FLUSH_COMMANDS_BIT` bit is set in *flags*,
- *sync* is unsignaled when **ClientWaitSync** is called,
- and the calls to **ClientWaitSync** and **FenceSync** were issued from the same context,

then the GL will behave as if the equivalent of **Flush** were inserted immediately after the creation of *sync*.

Additional constraints on the use of sync objects are discussed in chapter 5.

State must be maintained to indicate which sync object names are currently in use. The state required for each sync object in use is an integer for the specific type, an integer for the condition, and a bit indicating whether the object is signaled or unsignaled. The initial values of sync object state are defined as specified by **FenceSync**.

[3]The simple flushing behavior defined by `SYNC_FLUSH_COMMANDS_BIT` will not help when waiting for a fence command issued in another context's command stream to complete. Applications which block on a fence sync object must take additional steps to assure that the context from which the corresponding fence command was issued has flushed that command to the graphics pipeline.

4.1.3 Sync Object Queries

Properties of sync objects may be queried using the command

```
void GetSynciv( sync sync, enum pname, sizei count,
   sizei *length, int *values );
```

The value or values being queried are returned in the parameters *length* and *values*.

On success, **GetSynciv** replaces up to *count* integers in *values* with the corresponding property values of the object being queried. The actual number of integers replaced is returned in **length*. If *length* is NULL, no length is returned.

If *pname* is OBJECT_TYPE, a single value representing the specific type of the sync object is placed in *values*. The only type supported is SYNC_FENCE.

If *pname* is SYNC_STATUS, a single value representing the status of the sync object (SIGNALED or UNSIGNALED) is placed in *values*.

If *pname* is SYNC_CONDITION, a single value representing the condition of the sync object is placed in *values*. The only condition supported is SYNC_GPU_COMMANDS_COMPLETE.

If *pname* is SYNC_FLAGS, a single value representing the flags with which the sync object was created is placed in *values*. No flags are currently supported.

Errors

An INVALID_VALUE error is generated if *sync* is not the name of a sync object.

An INVALID_ENUM error is generated if *pname* is not one of the values described above.

An INVALID_VALUE error is generated if *count* is negative.

The command

```
boolean IsSync( sync sync );
```

returns TRUE if *sync* is the name of a sync object. If *sync* is not the name of a sync object, or if an error condition occurs, **IsSync** returns FALSE (note that zero is not the name of a sync object).

Sync object names immediately become invalid after calling **DeleteSync**, as discussed in sections 4.1 and 5.2, but the underlying sync object will not be deleted until it is no longer associated with any fence command and no longer blocking any ***WaitSync** command.

4.2 Query Objects and Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands.

4.2.1 Query Object Types and Targets

Query types supported by the GL include:

- Primitive queries with a *target* of PRIMITIVES_GENERATED (see section 13.4) return information on the number of primitives processed by the GL. There may be at most the value of MAX_VERTEX_STREAMS active queries of this type.
- Primitive queries with a *target* of TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN (see section 13.4) return information on the number of primitives written to one or more buffer objects. There may be at most the value of MAX_VERTEX_STREAMS active queries of this type.
- Transform feedback overflow queries with a *target* of TRANSFORM_FEEDBACK_OVERFLOW or TRANSFORM_FEEDBACK_STREAM_OVERFLOW return information on whether or not transform feedback overflow happened for one or more streams (see section 13.5).
- Occlusion queries with a *target* of SAMPLES_PASSED, ANY_SAMPLES_PASSED or ANY_SAMPLES_PASSED_CONSERVATIVE (see section 17.3.5) count the number of fragments or samples that pass the depth test, or set a boolean to true when any fragments or samples pass the depth test. There may be at most one active query of this type.
- Time elapsed queries with a *target* of TIME_ELAPSED (see section 4.3) record the amount of time needed to fully process a sequence of commands. There may be at most one active query of this type.
- Timer queries with a *target* of TIMESTAMP (see section 4.3) record the current time of the GL. There may be at most one active query of this type.
- Submission queries with a *target* of VERTICES_SUBMITTED and PRIMITIVES_SUBMITTED return information on the number of vertices and primitives transferred to the GL, respectively (see section 10.10).
- Vertex shader queries with a *target* of VERTEX_SHADER_INVOCATIONS return information on the number of times the vertex shader has been invoked (see section 11.1.4).

- Tessellation shader queries with a *target* of `TESS_CONTROL_SHADER_PATCHES` and `TESS_EVALUATION_SHADER_INVOCATIONS` return information on the number of patches processed by the tessellation control shader stage and the number of times the tessellation evaluation shader has been invoked, respectively (see section 11.2.4).
- Geometry shader queries with a *target* of `GEOMETRY_SHADER_INVOCATIONS` and `GEOMETRY_SHADER_PRIMITIVES_EMITTED` return information on the number of times the geometry shader has been invoked and the number of primitives it emitted (see section 11.3.5).
- Primitive clipping queries with a *target* of `CLIPPING_INPUT_PRIMITIVES` and `CLIPPING_OUTPUT_PRIMITIVES` return information on the number of primitives that were processed in the primitive clipping stage and the number of primitives that were output by the primitive clipping stage and are further processed by the rasterization stage, respectively (see section 13.7.3).
- Fragment shader queries with a *target* of `FRAGMENT_SHADER_INVOCATIONS` return information on the number of times the fragment shader has been invoked (see section 15.3).
- Compute shader queries with a *target* of `COMPUTE_SHADER_INVOCATIONS` return information on the number of times the compute shader has been invoked (see section 19.2).

4.2.2 Query Object Creation and Activation

The results of asynchronous queries are not returned by the GL immediately after the completion of the last command in the set; subsequent commands can be processed while the query results are not complete. When available, the query results are stored in an associated query object. The commands described in section 4.2.3 provide mechanisms to determine when query results are available and return the actual results of the query. The name space for query objects is the unsigned integers, with zero reserved by the GL.

The command

`void` **GenQueries**(`sizei` *n*, `uint` **ids*);

returns *n* previously unused query object names in *ids*. These names are marked as used, for the purposes of **GenQueries** only, but no object is associated with them until the first time they are used by **BeginQuery**, **BeginQueryIndexed**, or **QueryCounter** (see section 4.3).

Errors

An INVALID_VALUE error is generated if *n* is negative.

Query objects may also be created with the command

```
void CreateQueries( enum target, sizei n, uint *ids );
```

CreateQueries returns *n* previously unused query object names in *ids*, each representing a new query object with the specified *target*. *target* must be one of the query object targets described in section 4.2.1.

The initial state of the resulting query object is that the result is marked available (the value of QUERY_RESULT_AVAILABLE for the query object is TRUE) and the result value (the value of QUERY_RESULT) is zero.

Errors

An INVALID_ENUM error is generated if *target* is not one of the query object targets described in section 4.2.1.

An INVALID_VALUE error is generated if *n* is negative.

Query objects are deleted by calling

```
void DeleteQueries( sizei n, const uint *ids );
```

ids contains *n* names of query objects to be deleted. After a query object is deleted, its name is again unused. If an active query object is deleted its name immediately becomes unused, but the underlying object is not deleted until it is no longer active (see section 5.1). Unused names in *ids* that have been marked as used for the purposes of **GenQueries** are marked as unused again. Unused names in *ids* are silently ignored, as is the value zero.

Errors

An INVALID_VALUE error is generated if *n* is negative.

Each type of query, other than timer queries of type TIMESTAMP, supported by the GL has an active query object name for each of the possible active queries. If an active query object name is non-zero, the GL is currently tracking the corresponding information, and the query results will be written into that query object. If an active query object name is zero, no such information is being tracked.

A query object may be created and made active with the command

```
void BeginQueryIndexed( enum target, uint index,
  uint id );
```

target indicates the type of query to be performed. The valid values of *target* are discussed in more detail in subsequent sections.

If *id* is an unused query object name, the name is marked as used and associated with a new query object of the type specified by *target*. Otherwise *id* must be the name of an existing query object of that type. Note that occlusion query objects specified by either of the two *target*s ANY_SAMPLES_PASSED or ANY_SAMPLES_PASSED_CONSERVATIVE may be reused for either *target* in future queries. Objects specified with *target* SAMPLES_PASSED may only be reused for that *target*.

index is the index of the query, and must be between zero and a *target*-specific maximum. The state of the query object named *id*, whether newly created or not, is that the result is marked unavailable (the value of QUERY_RESULT_AVAILABLE for the query object is FALSE), and the result value (the value of QUERY_RESULT) is zero.

The active query object name for *target* and *index* is set to *id*.

Errors

An INVALID_ENUM error is generated if *target* is TIMESTAMP, or is not one of the query object targets described in section 4.2.1.

An INVALID_VALUE error is generated if *target* is PRIMITIVES_GENERATED, TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW, and *index* is not in the range zero to the value of MAX_VERTEX_STREAMS minus one.

An INVALID_VALUE error is generated if *target* is a valid target other than PRIMITIVES_GENERATED, TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW, and *index* is not zero.

An INVALID_OPERATION error is generated if *id* is any of:

- zero
- the name of an existing query object whose type does not match *target*
- an active query object name for any *target* and *index*
- the active query object for conditional rendering (see section 10.9).

An INVALID_OPERATION error is generated if the active query object name for *target* and *index* is non-zero.

The command

```
void BeginQuery( enum target, uint id );
```

is equivalent to

```
BeginQueryIndexed(target, 0, id);
```

The command

```
void EndQueryIndexed( enum target, uint index );
```

marks the end of the sequence of commands to be tracked for the active query specified by *target* and *index*. *target* and *index* have the same meaning as for **BeginQueryIndexed**.

The corresponding active query object is updated to indicate that query results are not available, and the active query object name for *target* and *index* is reset to zero. When the commands issued prior to **EndQueryIndexed** have completed and a final query result is available, the query object active when **EndQueryIndexed** was called is updated to contain the query result and to indicate that the query result is available.

Errors

An INVALID_ENUM error is generated if *target* is TIMESTAMP, or is not one of the query object targets described in section 4.2.1.

An INVALID_VALUE error is generated if *target* is PRIMITIVES_GENERATED TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW, and *index* is not in the range zero to the value of MAX_VERTEX_STREAMS minus one.

An INVALID_VALUE error is generated if *target* is a valid target other than PRIMITIVES_GENERATED, TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW, and *index* is not zero.

An INVALID_OPERATION error is generated if the active query object name for *target* and *index* is zero.

The command

```
void EndQuery( enum target );
```

is equivalent to

```
EndQueryIndexed(target, 0);
```

Query objects contain two pieces of state: a single bit indicating whether a query result is available, and an integer containing the query result value. The number of bits, n, used to represent the query result is implementation-dependent and may be determined as described in section 4.2.3. The initial state of a query object depends on whether it was created with **CreateQueries** or **BeginQueryIndexed**, as described above.

If the query result overflows (exceeds the value $2^n - 1$), its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at $2^n - 1$ and incrementing no further.

The necessary state for each possible active query *target* and *index* is an unsigned integer holding the active query object name (zero if no query object is active), and any state necessary to keep the current results of an asynchronous query in progress. Only a single type of occlusion query can be active at one time, so the required state for occlusion queries is shared.

4.2.3 Query Object Queries

The command

```
boolean IsQuery( uint id );
```

returns TRUE if *id* is the name of a query object. If *id* is zero, or if *id* is a non-zero value that is not the name of a query object, **IsQuery** returns FALSE.

Information about an active query object may be queried with the command

```
void GetQueryIndexediv( enum target, uint index,
   enum pname, int *params );
```

target and *index* specify the active query, and have the same meaning as for **BeginQueryIndexed**.

If *pname* is CURRENT_QUERY, the name of the currently active query object for *target* and *index*, or zero if no query is active, will be placed in *params*. If *target* is TIMESTAMP, zero is always returned.

If *pname* is QUERY_COUNTER_BITS, *index* is ignored and the implementation-dependent number of bits used to hold the query result for *target* will be placed in

params. The number of query counter bits may be zero, in which case the counter contains no useful information.

For primitive queries (PRIMITIVES_GENERATED and TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN) if the number of bits is non-zero, the minimum number of bits allowed is 32.

For transform feedback overflow queries (TRANSFORM_FEEDBACK_OVERFLOW and TRANSFORM_FEEDBACK_STREAM_OVERFLOW) if the number of bits is non-zero, the minimum number of bits is 1.

For occlusion queries with *target* ANY_SAMPLES_PASSED or ANY_SAMPLES_PASSED_CONSERVATIVE, if the number of bits is non-zero, the minimum number of bits is 1. For occlusion queries with *target* SAMPLES_PASSED, if the number of bits is non-zero, the minimum number of bits allowed is 32.

For timer queries (*target* TIME_ELAPSED and TIMESTAMP), if the number of bits is non-zero, the minimum number of bits allowed is 30. This will allow at least one second of timing.

For pipeline statistics queries (*target* VERTICES_SUBMITTED, PRIMITIVES_SUBMITTED, VERTEX_SHADER_INVOCATIONS, TESS_CONTROL_SHADER_PATCHES, TESS_EVALUATION_SHADER_INVOCATIONS, GEOMETRY_SHADER_INVOCATIONS, FRAGMENT_SHADER_INVOCATIONS, COMPUTE_SHADER_INVOCATIONS, GEOMETRY_SHADER_PRIMITIVES_EMITTED, CLIPPING_INPUT_PRIMITIVES, and CLIPPING_OUTPUT_PRIMITIVES), if the number of bits is non-zero, the minimum number of bits allowed is 32.

Errors

An INVALID_ENUM error is generated if *target* is not one of the query object targets described in section 4.2.1.

An INVALID_VALUE error is generated if *target* is PRIMITIVES_GENERATED, TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW, and *index* is not in the range zero to the value of MAX_VERTEX_STREAMS minus one.

An INVALID_VALUE error is generated if *target* is a valid target other than PRIMITIVES_GENERATED, TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW, and *index* is not zero.

An INVALID_ENUM error is generated if *pname* is not CURRENT_QUERY or QUERY_COUNTER_BITS.

The command

```
void GetQueryiv( enum target, enum pname, int *params );
```

is equivalent to

```
GetQueryIndexediv(target, 0, pname, params);
```

The state of a query object may be queried with the commands

```
void GetQueryObjectiv( uint id, enum pname,
   int *params );
void GetQueryObjectuiv( uint id, enum pname,
   uint *params );
void GetQueryObjecti64v( uint id, enum pname,
   int64 *params );
void GetQueryObjectui64v( uint id, enum pname,
   uint64 *params );
void GetQueryBufferObjectiv( uint id, uint buffer,
   enum pname, intptr offset );
void GetQueryBufferObjectuiv( uint id, uint buffer,
   enum pname, intptr offset );
void GetQueryBufferObjecti64v( uint id, uint buffer,
   enum pname, intptr offset );
void GetQueryBufferObjectui64v( uint id, uint buffer,
   enum pname, intptr offset );
```

id is the name of a query object.

For **GetQueryBufferObject***, *buffer* is the name of a buffer object and *offset* is an offset into *buffer* at which the queried value is written.

For **GetQueryObject***, the queried value may be returned either in client memory or in a buffer object. If zero is bound to the current query result buffer binding point (see QUERY_RESULT in section 6.1), then *params* is treated as a pointer into client memory at which the queried value is written. Otherwise, *params* is treated as an offset into the query result buffer object at which the queried value is written.

There may be an indeterminate delay before a query object's result value is available. If *pname* is QUERY_RESULT_AVAILABLE, FALSE is returned if such a delay would be required; otherwise TRUE is returned. It must always be true that if any query object returns a result available of TRUE, all queries of the same type issued prior to that query must also return TRUE. Repeatedly querying QUERY_RESULT_AVAILABLE for any given query object is guaranteed to return TRUE eventually.

If *pname* is QUERY_TARGET, then the target of the query object is returned as a single integer.

If *pname* is QUERY_RESULT, then the query object's result value is returned as a single integer. If the value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned. If the number of query counter bits for *target* is zero, then the result is returned as a single integer with the value zero. Querying QUERY_RESULT for any given query object forces that query to complete within a finite amount of time.

If *pname* is QUERY_RESULT_NO_WAIT, then the query object's result value is returned as a single integer if the result is available at the time of the state query. If the result is not available then the query return value is not written.

If multiple queries are issued using the same object name prior to calling these query commands, the result and availability information returned will always be from the last query issued. The results from any queries before the last one will be lost if they are not retrieved before starting a new query on the same *target* and *id*.

Errors

An INVALID_OPERATION error is generated if *id* is not the name of a query object, or if the query object named by *id* is currently active.

An INVALID_OPERATION error is generated by **GetQueryBufferObject*** if *buffer* is not the name of an existing buffer object.

An INVALID_ENUM error is generated if *pname* is not QUERY_RESULT, QUERY_RESULT_AVAILABLE, QUERY_RESULT_NO_WAIT, or QUERY_TARGET.

An INVALID_OPERATION error is generated if the query writes to a buffer object, and the specified buffer offset would cause data to be written beyond the bounds of that buffer object.

An INVALID_VALUE error is generated by **GetQueryBufferObject*** if *offset* is negative.

4.3 Time Queries

Query objects may also be used to track the amount of time needed to fully complete a set of GL commands (a *time elapsed query*), or to determine the current time of the GL (a *timer query*).

When **BeginQuery** and **EndQuery** are called with a *target* of TIME_ELAPSED, the GL prepares to start and stop the timer used for time elapsed queries. The timer

is started or stopped when the effects from all previous commands on the GL client and server state and the framebuffer have been fully realized. The **BeginQuery** and **EndQuery** commands may return before the timer is actually started or stopped. When the time elapsed query timer is finally stopped, the elapsed time (in nanoseconds) is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

A timer query object is created with the command

`void` **QueryCounter**(`uint` *id*, `enum` *target*);

target must be `TIMESTAMP`. If *id* is an unused query object name, the name is marked as used and associated with a new query object of type `TIMESTAMP`. Otherwise *id* must be the name of an existing query object of that type.

Alternatively, `TIMESTAMP` query objects can be created by calling **CreateQueries** with *target* set to `TIMESTAMP`.

When **QueryCounter** is called, the GL records the current time into the corresponding query object. The time is recorded after all previous commands on the GL client and server state and the framebuffer have been fully realized. When the time is recorded, the query result for that object is marked available. Timer queries can be used within a **BeginQuery / EndQuery** block where the *target* is `TIME_ELAPSED`, and it does not affect the result of that query object.

The current time of the GL may be queried by calling **GetIntegerv** or **GetInteger64v** with the symbolic constant `TIMESTAMP`. This will return the GL time after all previous commands have reached the GL server but have not yet necessarily executed. By using a combination of this synchronous get command and the asynchronous timestamp query object target, applications can measure the latency between when commands reach the GL server and when they are realized in the framebuffer.

Errors

An `INVALID_ENUM` error is generated if *target* is not `TIMESTAMP`.

An `INVALID_OPERATION` error is generated if *id* is the name of an existing query object whose type is not `TIMESTAMP`.

Chapter 5

Shared Objects and Multiple Contexts

This chapter describes special considerations for objects shared between multiple OpenGL contexts, including deletion behavior and how changes to shared objects are propagated between contexts.

Objects that may be shared between contexts include buffer objects, display lists, program and shader objects, renderbuffer objects, sampler objects, sync objects, and texture objects (except for the texture objects named zero).

Some of these objects may contain *views* (alternate interpretations) of part or all of the data store of another object. Examples are texture buffer objects, which contain a view of a buffer object's data store, and texture views, which contain a view of another texture object's data store. Views act as references on the object whose data store is viewed.

Objects which contain references to other objects include framebuffer, program pipeline, transform feedback, and vertex array objects. Such objects are called *container objects* and are not shared.

Implementations may allow sharing between contexts implementing different OpenGL versions or different profiles of the same OpenGL version (see appendix E). However, implementation-dependent behavior may result when aspects and/or behaviors of such shared objects do not apply to, and/or are not described by more than one version or profile.

5.1 Object Deletion Behavior

5.1.1 Side Effects of Shared Context Destruction

The *share list* is the group of all contexts which share objects. If a shared object is not explicitly deleted, then destruction of any individual context has no effect on that object unless it is the only remaining context in the share list. Once the last context on the share list is destroyed, all shared objects, and all other resources allocated for that context or share list, will be deleted and reclaimed by the implementation as soon as possible.

5.1.2 Automatic Unbinding of Deleted Objects

When a buffer, texture, or renderbuffer object is deleted, it is unbound from any bind points it is bound to in the current context, and detached from any attachments of container objects that are bound to the current context, as described for **DeleteBuffers**, **DeleteTextures**, and **DeleteRenderbuffers**. If the object binding was established with other related state (such as a buffer range in **BindBufferRange** or selected level and layer information in **FramebufferTexture** or **BindImageTexture**), all such related state are restored to default values by the automatic unbind. Bind points in other contexts are not affected. Attachments to unbound container objects, such as deletion of a buffer attached to a vertex array object which is not bound to the context, are not affected and continue to act as references on the deleted object, as described in the following section.

5.1.3 Deleted Object and Object Name Lifetimes

When a buffer, texture, sampler, renderbuffer, query, or sync object is deleted, its name immediately becomes invalid (e.g. is marked unused), but the underlying object will not be deleted until it is no longer *in use*.

A buffer, texture, sampler, or renderbuffer object is in use if any of the following conditions are satisfied:

- the object is attached to any container object
- the object is bound to a context bind point in any context
- any other object contains a view of the data store of the object.

A sync object is in use while there is a corresponding fence command which has not yet completed and signaled the sync object, or while there are any GL

clients and/or servers blocked on the sync object as a result of **ClientWaitSync** or **WaitSync** commands.

Query objects are in use so long as they are active, as described in section 4.2.

When a shader object or program object is deleted, it is flagged for deletion, but its name remains valid until the underlying object can be deleted because it is no longer in use. A shader object is in use while it is attached to any program object. A program object is in use while it is attached to any program pipeline object or is a current program in any context.

Caution should be taken when deleting an object while it is in use (as defined above). Following its deletion, the object's name may be used by any context to create a new object or returned by **Gen*** or **Create*** commands. The underlying object state and data for such a deleted, but still in use object may still be read and written by the GL, but cannot be accessed by name. The underlying storage backing a deleted object will not be reclaimed by the GL until all references to the object from container object attachment points, context binding points, views, fence commands, active queries, etc. are removed. Since the name is marked unused, binding the name will create a new object with the same name, and attaching the name will generate an error.

5.2 Sync Objects and Multiple Contexts

When multiple GL clients and/or servers are blocked on a single sync object and that sync object is signaled, all such blocks are released. The order in which blocks are released is implementation-dependent.

5.3 Propagating Changes to Objects

GL objects contain two types of information, *data* and *state*. Collectively these are referred to below as the *contents* of an object. For the purposes of propagating changes to object contents as described below, data and state are treated consistently.

Data is information the GL implementation does not have to inspect, and does not have an operational effect. Currently, data consists of:

- Pixels in the framebuffer.
- The contents of the data stores of buffer objects, renderbuffers, and textures.

State determines the configuration of the rendering pipeline, and the GL implementation does have to inspect it.

In hardware-accelerated GL implementations, state typically lives in GPU registers, while data typically lives in GPU memory.

When the contents of an object *T* are changed, such changes are not always immediately visible, and do not always immediately affect GL operations involving that object. Changes may occur via any of the following means:

- State-setting commands, such as **TexParameter**.
- Data-setting commands, such as **TexSubImage*** or **BufferSubData**.
- Data-setting through rendering to renderbuffers or textures attached to a framebuffer object.
- Data-setting through transform feedback operations followed by an **EndTransformFeedback** command.
- Commands that affect both state and data, such as **TexImage*** and **BufferData**.
- Changes to mapped buffer data followed by a command such as **UnmapBuffer** or **FlushMappedBufferRange**.
- Rendering commands that trigger shader invocations, where the shader performs image or buffer variable stores or atomic operations, or built-in atomic counter functions.

When *T* is a texture, the contents of *T* are construed to include the contents of the data store of *T*, even if *T*'s data store was modified via a different view of the data store.

5.3.1 Determining Completion of Changes to an object

The contents of an object *T* are considered to have been changed once a command such as described in section 5.3 has completed. Completion of a command [1] may be determined either by calling **Finish**, or by calling **FenceSync** and executing a **WaitSync** command on the associated sync object. The second method does not require a round trip to the GL server and may be more efficient, particularly when changes to *T* in one context must be known to have completed before executing commands dependent on those changes in another context. In cases where a feedback loop has been established (see sections 8.6.1, 8.14.2.1, and 9.3, as well as the

[1] The GL already specifies that a single context processes commands in the order they are received. This means that a change to an object in a context at time *t* must be completed by the time a command issued in the same context at time $t + 1$ uses the result of that change.

discussion of rule 1 below in section 5.3.3) the resulting contents of an object may be undefined.

5.3.2 Definitions

In the remainder of this section, the following terminology is used:

- An object *T* is *directly attached* to the current context if it has been bound to one of the context binding points. Examples include but are not limited to bound textures, bound framebuffers, bound vertex arrays, and current programs.
- *T* is *indirectly attached* to the current context if it is attached to another object *C*, referred to as a *container object*, and *C* is itself directly or indirectly attached. Examples include but are not limited to renderbuffers or textures attached to framebuffers; buffers attached to vertex arrays; and shaders attached to programs.
- An object *T* which is directly attached to the current context may be *re-attached* by re-binding *T* at the same bind point. An object *T* which is indirectly attached to the current context may be re-attached by re-attaching the container object *C* to which *T* is attached.

  Corollary: re-binding *C* to the current context re-attaches *C* and its hierarchy of contained objects.

5.3.3 Rules

The following rules must be obeyed by all GL implementations:

Rule 1 *If the contents of an object* T *are changed in the current context while* T *is directly or indirectly attached, then all operations on* T *will use the new contents in the current context.*

Note: *The intent of this rule is to address changes in a single context only. The multi-context case is handled by the other rules.*

Note: *"Updates" via rendering or transform feedback are treated consistently with updates via GL commands. Once* **EndTransformFeedback** *has been issued, any subsequent command in the same context that uses the results of the transform feedback operation will see the results. If a feedback loop is setup between rendering and transform feedback (see section 13.3.3), results will be undefined.*

Rule 2 *While a container object* C *is bound, any changes made to the contents of* C*'s attachments in the current context are guaranteed to be seen. To guarantee seeing changes made in another context to objects attached to* C*, such changes must be completed in that other context (see section 5.3.1) prior to* C *being bound. Changes made in another context but not determined to have completed as described in section 5.3.1, or after* C *is bound in the current context, are not guaranteed to be seen.*

Rule 3 *Changes to the contents of shared objects are not automatically propagated between contexts. If the contents of a shared object* T *are changed in a context other than the current context, and* T *is already directly or indirectly attached to the current context, any operations on the current context involving* T *via those attachments are not guaranteed to use its new contents.*

Rule 4 *If the contents of an object* T *are changed in a context other than the current context,* T *must be attached or re-attached to at least one binding point in the current context, or at least one attachment point of a currently bound container object* C*, in order to guarantee that the new contents of* T *are visible in the current context.*

Note: "Attached or re-attached" *means either attaching an object to a binding point it wasn't already attached to, or attaching an object again to a binding point it was already attached.*

Example: *If a texture image is bound to multiple texture bind points and the texture is changed in another context, re-binding the texture at any one of the texture bind points is sufficient to cause the changes to be visible at all texture bind points.*

Chapter 6

Buffer Objects

Buffer objects contain a data store holding a fixed-sized allocation of server memory. This chapter specifies commands to create, manage, and destroy buffer objects. Specific types of buffer objects and their uses are briefly described together with references to their full specification.

The name space for buffer objects is the unsigned integers, with zero reserved by the GL.

The command

`void` **GenBuffers**(`sizei` *n*, `uint` **buffers*);

returns *n* previously unused buffer object names in *buffers*. These names are marked as used, for the purposes of **GenBuffers** only, but they acquire buffer state only when they are first bound with **BindBuffer** (see below), just as if they were unused.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

In addition to generating an unused name and then binding it to a target with **BindBuffer**, a buffer object may also be created with the command

`void` **CreateBuffers**(`sizei` *n*, `uint` **buffers*);

CreateBuffers returns *n* previously unused buffer names in *buffers*, each representing a new buffer object initialized as if it had been bound to an unspecified target.

Errors

An INVALID_VALUE error is generated if *n* is negative.

Buffer objects are deleted by calling

`void` **DeleteBuffers**(`sizei` *n*, `const uint` **buffers*);

buffers contains *n* names of buffer objects to be deleted. After a buffer object is deleted it has no contents, and its name is again unused. If any portion of a buffer object being deleted is mapped in the current context or any context current to another thread, it is as though **UnmapBuffer** (see section 6.3.1) is executed in each such context prior to deleting the data store of the buffer.

Unused names in *buffers* that have been marked as used for the purposes of **GenBuffers** are marked as unused again. Unused names in *buffers* are silently ignored, as is the value zero.

Errors

An INVALID_VALUE error is generated if *n* is negative.

The command

`boolean` **IsBuffer**(`uint` *buffer*);

returns TRUE if *buffer* is the name of a buffer object. If *buffer* is zero, or if *buffer* is a non-zero value that is not the name of a buffer object, **IsBuffer** returns FALSE.

6.1 Creating and Binding Buffer Objects

A buffer object is created by binding an unused name to a buffer target. The binding is effected by calling

`void` **BindBuffer**(`enum` *target*, `uint` *buffer*);

target must be one of the targets listed in table 6.1. If the buffer object named *buffer* has not been previously bound, or has been deleted since the last binding, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising all the state and with the same initial values listed in table 6.2.

Buffer objects created by binding an unused name to any of the valid *target*s are formally equivalent.

Target name	Purpose	Described in section(s)
ARRAY_BUFFER	Vertex attributes	10.3.9
ATOMIC_COUNTER_BUFFER	Atomic counter storage	7.7
COPY_READ_BUFFER	Buffer copy source	6.6
COPY_WRITE_BUFFER	Buffer copy destination	6.6
DISPATCH_INDIRECT_BUFFER	Indirect compute dispatch commands	19
DRAW_INDIRECT_BUFFER	Indirect command arguments	10.3.11
ELEMENT_ARRAY_BUFFER	Vertex array indices	10.3.10
PARAMETER_BUFFER	Draw parameters	10.4
PIXEL_PACK_BUFFER	Pixel read target	18.2, 22
PIXEL_UNPACK_BUFFER	Texture data source	8.4
QUERY_BUFFER	Query result buffer	4.2.3
SHADER_STORAGE_BUFFER	Read-write storage for shaders	7.8
TEXTURE_BUFFER	Texture data buffer	8.9
TRANSFORM_FEEDBACK_BUFFER	Transform feedback buffer	13.3
UNIFORM_BUFFER	Uniform block storage	7.6.2

Table 6.1: Buffer object binding targets.

Name	Type	Initial Value	Legal Values
BUFFER_SIZE	int64	0	any non-negative integer
BUFFER_USAGE	enum	STATIC_DRAW	STREAM_DRAW, STREAM_READ, STREAM_COPY, STATIC_DRAW, STATIC_READ, STATIC_COPY, DYNAMIC_DRAW, DYNAMIC_READ, DYNAMIC_COPY
BUFFER_ACCESS	enum	READ_WRITE	READ_ONLY, WRITE_ONLY, READ_WRITE
BUFFER_ACCESS_FLAGS	int	0	See section 6.3
BUFFER_IMMUTABLE_STORAGE	boolean	FALSE	TRUE, FALSE
BUFFER_MAPPED	boolean	FALSE	TRUE, FALSE
BUFFER_MAP_POINTER	void*	NULL	address
BUFFER_MAP_OFFSET	int64	0	any non-negative integer
BUFFER_MAP_LENGTH	int64	0	any non-negative integer
BUFFER_STORAGE_FLAGS	int	0	See section 6.2

Table 6.2: Buffer object parameters and their values.

BindBuffer may also be used to bind an existing buffer object. If the bind is successful no change is made to the state of the newly bound buffer object, and any previous binding to *target* is broken.

While a buffer object is bound, GL operations on the target to which it is bound affect the bound buffer object, and queries of the target to which a buffer object is bound return state from the bound object. Operations on the target also affect any other bindings of that object.

If a buffer object is deleted while it is bound, all bindings to that object in the current context (i.e. in the thread that called **DeleteBuffers**) are reset to zero. Bindings to that buffer in other contexts are not affected, and the deleted buffer may continue to be used at any places it remains bound or attached, as described in section 5.1.

Initially, each buffer object target is bound to zero.

Errors

An `INVALID_ENUM` error is generated if *target* is not one of the targets listed in table 6.1.

An `INVALID_OPERATION` error is generated by client attempts to modify or query buffer object state for a target bound to zero, since there is no buffer object corresponding to the name zero,

6.1.1 Binding Buffer Objects to Indexed Targets

Buffer objects may be created and bound to *indexed targets* by calling one of the commands

```
void BindBufferRange( enum target, uint index,
    uint buffer, intptr offset, sizeiptr size );
void BindBufferBase( enum target, uint index, uint buffer );
```

target must be one of `ATOMIC_COUNTER_BUFFER`, `SHADER_STORAGE_BUFFER`, `TRANSFORM_FEEDBACK_BUFFER` or `UNIFORM_BUFFER`. Additional language specific to each target is included in sections referred to for each target in table 6.1.

Each *target* represents an indexed array of buffer object binding points, as well as a single general binding point that can be used by other buffer object manipulation functions, such as **BindBuffer** or **MapBuffer**. Both commands bind the buffer object named by *buffer* to both the general binding point, and to the binding point in the array given by *index*. If the binds are successful no change is made

to the state of the bound buffer object, and any previous bindings to the general binding point or to the binding point in the array are broken.

If the buffer object named *buffer* has not been previously bound, or has been deleted since the last binding, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising all the state and with the same initial values listed in table 6.2.

For **BindBufferRange**, *offset* specifies a starting offset into the buffer object *buffer*, and *size* specifies the amount of data that can be read from or written to the buffer object while used as an indexed target. Both *offset* and *size* are in basic machine units.

BindBufferBase binds the entire buffer, even when the size of the buffer is changed after the binding is established. The starting offset is zero, and the amount of data that can be read from or written to the buffer is determined by the size of the bound buffer at the time the binding is used.

Regardless of the *size* specified with **BindBufferRange**, the GL will never read or write beyond the end of a bound buffer. In some cases this constraint may result in visibly different behavior when a buffer overflow would otherwise result, such as described for transform feedback operations in section 13.3.2.

Errors

An `INVALID_ENUM` error is generated if *target* is not one of the targets listed above.

An `INVALID_VALUE` error is generated if *index* is greater than or equal to the number of *target*-specific indexed binding points, as described in section 6.7.1.

An `INVALID_VALUE` error is generated by **BindBufferRange** if *offset* is negative.

An `INVALID_VALUE` error is generated by **BindBufferRange** if *buffer* is non-zero and *size* is less than or equal to zero.

An `INVALID_VALUE` error is generated by **BindBufferRange** if *buffer* is non-zero and *offset* or *size* do not respectively satisfy the constraints described for those parameters for the specified *target*, as described in section 6.7.1.

The commands

```
void BindBuffersBase( enum target, uint first, sizei count,
    const uint *buffers );
void BindBuffersRange( enum target, uint first,
    sizei count, const uint *buffers, const
    intptr *offsets, const sizeiptr *sizes );
```

bind *count* existing buffer objects to bindings numbered *first* through $first + count - 1$ in the array of buffer binding points corresponding to *target*. If *buffers* is not NULL, it specifies an array of *count* values, each of which must be zero or the name of an existing buffer object. For **BindBuffersRange**, *offsets* and *sizes* specify arrays of *count* values indicating the range of each buffer to bind. If *buffers* is NULL, all bindings from *first* to $first + count - 1$ are reset to their unbound (zero) state. In this case, the offsets and sizes associated with the binding points are set to default values, ignoring *offsets* and *sizes*.

BindBuffersBase is equivalent (assuming no errors are generated) to:

```
for (i = 0; i < count; i++) {
    if (buffers == NULL) {
        BindBufferBase(target, first + i, 0);
    } else {
        BindBufferBase(target, first + i, buffers[i]);
    }
}
```

except that the single general buffer binding corresponding to *target* is unmodified, and that buffers will not be created if they do not exist.

BindBuffersRange is equivalent (assuming no errors are generated) to:

```
for (i = 0; i < count; i++) {
    if (buffers == NULL) {
        BindBufferRange(target, first + i, 0, 0, 0);
    } else {
        BindBufferRange(target, first + i, buffers[i],
            offsets[i], sizes[i]);
    }
}
```

except that the single general buffer binding corresponding to *target* is unmodified, and that buffers will not be created if they do not exist.

The values specified in *buffers*, *offsets*, and *sizes* will be checked separately for each binding point. When values for a specific binding point are invalid, the state for that binding point will be unchanged and an error will be generated. When such an error occurs, state for other binding points will still be changed if their corresponding values are valid.

Errors

An INVALID_ENUM error is generated if *target* is not one of the targets listed above.

An INVALID_OPERATION error is generated if $first + count$ is greater than the number of target-specific indexed binding points, as described in section 6.7.1.

An INVALID_VALUE error is generated if *count* is negative.

An INVALID_OPERATION error is generated if any value in *buffers* is not zero or the name of an existing buffer object.

An INVALID_VALUE error is generated by **BindBuffersRange** if any value in *offsets* is less than zero (per binding).

An INVALID_VALUE error is generated by **BindBuffersRange** if any value in *sizes* is less than or equal to zero (per binding).

An INVALID_VALUE error is generated by **BindBuffersRange** if any pair of values in *offsets* and *sizes* does not respectively satisfy the constraints described for those parameters for the specified target, as described in section 6.7.1 (per binding).

6.2 Creating and Modifying Buffer Object Data Stores

The data store of a buffer object is created by calling one of

```
void BufferStorage( enum target, sizeiptr size, const
    void *data, bitfield flags );
void NamedBufferStorage( uint buffer, sizeiptr size,
    const void *data, bitfield flags );
```

For **BufferStorage**, the buffer object is that bound to *target*, which must be one of the values listed in table 6.1. For **NamedBufferStorage**, *buffer* is the name of the buffer object. *size* is the size of the data store in basic machine units, and *flags* containing a bitfield describing the intended usage of the data store.

The data store of the buffer object is allocated as a result of these commands, and cannot be de-allocated until the buffer is deleted with a call to **DeleteBuffers**. Such a store may not be re-allocated through further calls to ***BufferStorage** or **BufferData**.

data specifies the address in client memory of the data that should be used to initialize the buffer object's data store. If *data* is NULL, the data store of the buffer object is created, but contains undefined data. Otherwise, *data* should point to an array of at least *size* basic machine units.

flags is the bitwise OR of flags describing the intended usage of the buffer object's data store by the application. Valid flags and their meanings are as follows:

`DYNAMIC_STORAGE_BIT` The contents of the data store may be updated after creation through calls to **BufferSubData**. If this bit is not set, the buffer content may not be directly updated by the client. The *data* argument may be used to specify the initial content of the buffer's data store regardless of the presence of the `DYNAMIC_STORAGE_BIT`. Regardless of the presence of this bit, buffers may always be updated with server-side calls such as **CopyBufferSubData** and **ClearBufferSubData**.

`MAP_READ_BIT` The data store may be mapped by the client for read access and a pointer in the client's address space obtained that may be read from.

`MAP_WRITE_BIT` The data store may be mapped by the client for write access and a pointer in the client's address space obtained that may be written to.

`MAP_PERSISTENT_BIT` The client may request that the server read from or write to the buffer while it is mapped. The client's pointer to the data store remains valid so long as the data store is mapped, even during execution of drawing or dispatch commands.

`MAP_COHERENT_BIT` Shared access to buffers that are simultaneously mapped for client access and are used by the server will be coherent, so long as that mapping is performed using **MapBufferRange** or **MapNamedBufferRange**. That is, data written to the store by either the client or server will be visible to any subsequently issued GL commands with no further action taken by the application. In particular,

- If `MAP_COHERENT_BIT` is not set and the client performs a write followed by a call to one of the **FlushMapped*BufferRange** commands with a range including the written range, then in subsequent commands the server will see the writes.
- If `MAP_COHERENT_BIT` is set and the client performs a write, then in subsequent commands the server will see the writes.
- If `MAP_COHERENT_BIT` is not set and the server performs a write, the application must call **MemoryBarrier** with the `CLIENT_MAPPED_BUFFER_BARRIER_BIT` set and then call **FenceSync** with `SYNC_GPU_COMMANDS_COMPLETE` (or **Finish**). Then the CPU will see the writes after the sync is complete.

Name	Value for **BufferData**	Value for ***BufferStorage**
BUFFER_SIZE	*size*	*size*
BUFFER_USAGE	*usage*	DYNAMIC_DRAW
BUFFER_ACCESS	READ_WRITE	READ_WRITE
BUFFER_ACCESS_FLAGS	0	0
BUFFER_IMMUTABLE_STORAGE	FALSE	TRUE
BUFFER_MAPPED	FALSE	FALSE
BUFFER_MAP_POINTER	NULL	NULL
BUFFER_MAP_OFFSET	0	0
BUFFER_MAP_LENGTH	0	0
BUFFER_STORAGE_FLAGS	MAP_READ_BIT | MAP_WRITE_BIT | DYNAMIC_STORAGE_BIT	*flags*

Table 6.3: Buffer object state after calling **BufferData**, **BufferStorage**, or **NamedBufferStorage**.

- If MAP_COHERENT_BIT is set and the server does a write, the application must call **FenceSync** with SYNC_GPU_COMMANDS_COMPLETE (or **Finish**). Then the CPU will see the writes after the sync is complete.

CLIENT_STORAGE_BIT When all other criteria for the buffer storage allocation are met, this bit may be used by an implementation to determine whether to use storage that is local to the server or to the client to serve as the backing store for the buffer.

If *flags* contains MAP_PERSISTENT_BIT, it must also contain at least one of MAP_READ_BIT or MAP_WRITE_BIT.

It is an error to specify MAP_COHERENT_BIT without also specifying MAP_PERSISTENT_BIT.

BufferStorage and **NamedBufferStorage** delete any existing data store, and set the values of the buffer object's state variables as shown in table 6.3.

If any portion of the buffer object is mapped in the current context or any context current to another thread, it is as though **UnmapBuffer** (see section 6.3.1) is executed in each such context prior to deleting the existing data store.

Errors

An INVALID_OPERATION error is generated by **BufferStorage** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **NamedBufferStorage** if *buffer* is not the name of an existing buffer object.

An INVALID_VALUE error is generated if *size* is less than or equal to zero.

An INVALID_VALUE error is generated if *flags* has any bits set other than those defined above.

An INVALID_VALUE error is generated if *flags* contains MAP_PERSISTENT_BIT but does not contain at least one of MAP_READ_BIT or MAP_WRITE_BIT.

An INVALID_VALUE error is generated if *flags* contains MAP_COHERENT_BIT, but does not also contain MAP_PERSISTENT_BIT.

An INVALID_OPERATION error is generated if the BUFFER_IMMUTABLE_STORAGE flag of the buffer bound to *target* is TRUE.

A mutable data store may be allocated for a buffer object with the commands

```
void BufferData( enum target, sizeiptr size, const
    void *data, enum usage );
void NamedBufferData( uint buffer, sizeiptr size, const
    void *data, enum usage );
```

For **BufferData**, the buffer object is that bound to *target*, which must be one of the targets listed in table 6.1. For **NamedBufferData**, *buffer* is the name of the buffer object.

size is the size of the data store in basic machine units, *data* points to the source data in client memory, and *usage* indicates the expected application usage pattern of the data store.

If *data* is non-NULL, then the source data is copied to the buffer object's data store. If *data* is NULL, then the contents of the buffer object's data store are undefined.

usage is specified as one of nine enumerated values. In the following descriptions, a buffer's data store is *sourced* when if is read from as a result of GL commands which specify images, or invoke shaders accessing buffer data as a result of drawing commands or compute shader dispatch.

The values are:

STREAM_DRAW The data store contents will be specified once by the application, and sourced at most a few times.

`STREAM_READ` The data store contents will be specified once by reading data from the GL, and queried at most a few times by the application.

`STREAM_COPY` The data store contents will be specified once by reading data from the GL, and sourced at most a few times

`STATIC_DRAW` The data store contents will be specified once by the application, and sourced many times.

`STATIC_READ` The data store contents will be specified once by reading data from the GL, and queried many times by the application.

`STATIC_COPY` The data store contents will be specified once by reading data from the GL, and sourced many times.

`DYNAMIC_DRAW` The data store contents will be respecified repeatedly by the application, and sourced many times.

`DYNAMIC_READ` The data store contents will be respecified repeatedly by reading data from the GL, and queried many times by the application.

`DYNAMIC_COPY` The data store contents will be respecified repeatedly by reading data from the GL, and sourced many times.

usage is provided as a performance hint only. The specified usage value does not constrain the actual usage pattern of the data store.

BufferData and **NamedBufferData** delete any existing data store, and set the values of the buffer object's state variables as shown in table 6.3.

If any portion of the buffer object is mapped in the current context or any context current to another thread, it is as though **UnmapBuffer** (see section 6.3.1) is executed in each such context prior to deleting the existing data store.

Clients must align data elements consistent with the requirements of the client platform, with an additional base-level requirement that an offset within a buffer to a datum comprising N basic machine units be a multiple of N.

Calling ***BufferData** is equivalent to calling **BufferStorage** with *target*, *size* and *data* as specified, and *flags* set to the logical OR of `DYNAMIC_STORAGE_BIT`, `MAP_READ_BIT` and `MAP_WRITE_BIT`. The GL will use the value of the *usage* parameter to ***BufferData** as a hint to further determine the intended use of the buffer. However, **BufferStorage** allocates immutable storage whereas ***BufferData** allocates mutable storage. Thus, when a buffer's data store is allocated through a call to ***BufferData**, the buffer's `BUFFER_IMMUTABLE_STORAGE` flag is set to `FALSE`.

Errors

An INVALID_OPERATION error is generated by **BufferData** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **NamedBufferData** if *buffer* is not the name of an existing buffer object.

An INVALID_VALUE error is generated if *size* is negative.

An INVALID_ENUM error is generated by **BufferData** if *target* is not one of the targets listed in table 6.1.

An INVALID_OPERATION error is generated if the BUFFER_IMMUTABLE_STORAGE flag of the buffer object is TRUE.

An INVALID_ENUM error is generated if *usage* is not one of the nine usages described above.

To modify some or all of the data contained in a buffer object's data store, the client may use the commands

```
void BufferSubData( enum target, intptr offset,
   sizeiptr size, const void *data );
void NamedBufferSubData( uint buffer, intptr offset,
   sizeiptr size, const void *data );
```

For **BufferSubData**, *target* specifies the target to which the buffer object is bound, and must be one of the values listed in table 6.1. For **NamedBufferSubData**, *buffer* is the name of the buffer object.

offset and *size* indicate the range of data in the buffer object that is to be replaced, in terms of basic machine units. *data* specifies a region of client memory *size* basic machine units in length, containing the data that replace the specified buffer range.

Errors

An INVALID_OPERATION error is generated by **BufferSubData** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **NamedBufferSubData** if *buffer* is not the name of an existing buffer object.

An INVALID_ENUM error is generated by **BufferSubData** if *target* is not one of the targets listed in table 6.1.

An INVALID_VALUE error is generated if *offset* or *size* is negative, or if $offset + size$ is greater than the value of BUFFER_SIZE for the buffer object.

An INVALID_OPERATION error is generated if any part of the specified buffer range is mapped with **MapBufferRange** or **MapBuffer** (see section 6.3), unless it was mapped with MAP_PERSISTENT_BIT set in the **MapBufferRange** *access* flags.

An INVALID_OPERATION error is generated if the BUFFER_IMMUTABLE_STORAGE flag of the buffer object is TRUE and the value of BUFFER_STORAGE_FLAGS for the buffer does not have the DYNAMIC_STORAGE_BIT set.

6.2.1 Clearing Buffer Object Data Stores

To fill all or part of a buffer object's data store with constant values, use the commands

```
void ClearBufferSubData( enum target, enum internalformat,
    intptr offset, sizeiptr size, enum format, enum type,
    const void *data );
void ClearNamedBufferSubData( uint buffer,
    enum internalformat, intptr offset, sizeiptr size,
    enum format, enum type, const void *data );
```

For **ClearBufferSubData**, the buffer object is that bound to *target*, which must be one of the values listed in table 6.1. For **ClearNamedBufferSubData**, *buffer* is the name of the buffer object.

internalformat must be set to one of the format tokens listed in table 8.24. *format* and *type* specify the format and type of the source data and are interpreted as described in section 8.4.4. Unlike the related **ClearTexImageSubData** command described in section 8.21, all defined conversions between the clear value's *format* and *type*, and the specified *internalformat* are allowed, including conversions between floating-point and integer components.

offset is the offset, measured in basic machine units, into the buffer object's data store from which to begin filling, and *size* is the size, also in basic machine units, of the range to fill.

data is a pointer to an array of between one and four components containing the data to be used as the source of the constant fill value. The elements of *data* are converted by the GL into the format specified by *internalformat* in the manner described in section 2.2.1, and then used to fill the specified range of the destination buffer. If *data* is NULL, then the pointer is ignored and the sub-range of the buffer is filled with zeros.

Errors

An INVALID_ENUM error is generated by **ClearBufferSubData** if *target* is not one of the targets listed in table 6.1.

An INVALID_VALUE error is generated by **ClearBufferSubData** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **ClearNamedBufferData** if *buffer* is not the name of an existing buffer object.

An INVALID_ENUM error is generated if *internalformat* is not one of the format tokens listed in table 8.24.

An INVALID_VALUE error is generated if *offset* or *size* are not multiples of the number of basic machine units for the internal format specified by *internalformat*. This value may be computed by multiplying the number of components for *internalformat* from table 8.24 by the size of the base type from that table.

An INVALID_VALUE error is generated if *offset* or *size* is negative, or if *offset* + *size* is greater than the value of BUFFER_SIZE for the buffer object.

An INVALID_OPERATION error is generated if any part of the specified buffer range is mapped with **MapBufferRange** or **MapBuffer** (see section 6.3), unless it was mapped with MAP_PERSISTENT_BIT set in the **MapBufferRange** *access* flags.

An INVALID_VALUE error is generated if *type* is not one of the types in table 8.7.

An INVALID_VALUE error is generated if *format* is not one of the formats in table 8.8.

The commands

```
void ClearBufferData( enum target, enum internalformat,
    enum format, enum type, const void *data );
void ClearNamedBufferData( uint buffer,
    enum internalformat, enum format, enum type, const
    void *data );
```

are respectively equivalent to

ClearBufferSubData (*target*, *internalformat*, 0, size, *format*, *type*, *data*);

and

ClearNamedBufferSubData (*buffer*, *internalformat*, 0, size, *format*, *type*, *data*);

where size is the value of BUFFER_SIZE for the destination buffer object.

6.3 Mapping and Unmapping Buffer Data

All or part of the data store of a buffer object may be mapped into the client's address space with the commands

```
void *MapBufferRange( enum target, intptr offset,
    sizeiptr length, bitfield acesss );
void *MapNamedBufferRange( uint buffer, intptr offset,
    sizeiptr length, bitfield access );
```

For **MapBufferRange**, the buffer object is that bound to *target*, which must be one of the values listed in table 6.1. For **MapNamedBufferRange**, *buffer* is the name of the buffer object.

offset and *length* indicate the range of data in the buffer object that is to be mapped, in terms of basic machine units. *access* is a bitfield containing flags which describe the requested mapping. These flags are described below.

If no error occurs, a pointer to the beginning of the mapped range is returned once all pending operations on that buffer have completed, and may be used to modify and/or query the corresponding range of the buffer, according to the following flag bits set in *access*:

- `MAP_READ_BIT` indicates that the returned pointer may be used to read buffer object data. No GL error is generated if the pointer is used to query a mapping which excludes this flag, but the result is undefined and system errors (possibly including program termination) may occur.

- `MAP_WRITE_BIT` indicates that the returned pointer may be used to modify buffer object data. No GL error is generated if the pointer is used to modify a mapping which excludes this flag, but the result is undefined and system errors (possibly including program termination) may occur.

- `MAP_PERSISTENT_BIT` indicates that it is not an error for the GL to read data from or write data to the buffer while it is mapped (see section 6.3.2). If this bit is set, the value of `BUFFER_STORAGE_FLAGS` for the buffer being mapped must include `MAP_PERSISTENT_BIT`.

- `MAP_COHERENT_BIT` indicates that the mapping should be performed coherently. That is, such a mapping follows the rules set forth in section 6.2. If this bit is set, the value of `BUFFER_STORAGE_FLAGS` for the buffer being mapped must include `MAP_COHERENT_BIT`.

If no error occurs, the pointer values returned by **Map*BufferRange** must reflect an allocation aligned to the value of MIN_MAP_BUFFER_ALIGNMENT basic machine units. Subtracting *offset* basic machine units from the returned pointer will always produce a multiple of the value of MIN_MAP_BUFFER_ALIGNMENT.

The returned pointer values may not be passed as parameter values to GL commands. For example, they may not be used to specify array pointers, or to specify or query pixel or texture image data; such actions produce undefined results, although implementations may not check for such behavior for performance reasons.

Mappings to the data stores of buffer objects may have nonstandard performance characteristics. For example, such mappings may be marked as uncacheable regions of memory, and in such cases reading from them may be very slow. To ensure optimal performance, the client should use the mapping in a fashion consistent with the values of BUFFER_USAGE and *access*. Using a mapping in a fashion inconsistent with these values is liable to be multiple orders of magnitude slower than using normal memory.

The following optional flag bits in *access* may be used to modify the mapping:

- MAP_INVALIDATE_RANGE_BIT indicates that the previous contents of the specified range may be discarded. Data within this range are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with MAP_READ_BIT.

- MAP_INVALIDATE_BUFFER_BIT indicates that the previous contents of the entire buffer may be discarded. Data within the entire buffer are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with MAP_READ_BIT.

- MAP_FLUSH_EXPLICIT_BIT indicates that one or more discrete subranges of the mapping may be modified. When this flag is set, modifications to each subrange must be explicitly flushed by calling **FlushMappedBufferRange**. No GL error is set if a subrange of the mapping is modified and not flushed, but data within the corresponding subrange of the buffer are undefined. This flag may only be used in conjunction with MAP_WRITE_BIT. When this option is selected, flushing is strictly limited to regions that are explicitly indicated with calls to **FlushMappedBufferRange** prior to unmap; if this option is not selected **UnmapBuffer** will automatically flush the entire mapped range when called.

Name	Value
BUFFER_ACCESS	Depends on *access*[1]
BUFFER_ACCESS_FLAGS	*access*
BUFFER_MAPPED	TRUE
BUFFER_MAP_POINTER	pointer to the data store
BUFFER_MAP_OFFSET	*offset*
BUFFER_MAP_LENGTH	*length*

Table 6.4: Buffer object state set by **MapBufferRange** and **MapNamedBufferRange**.
[1] BUFFER_ACCESS is set to READ_ONLY, WRITE_ONLY, or READ_WRITE if *access* & (MAP_READ_BIT|MAP_WRITE_BIT) is respectively MAP_READ_BIT, MAP_WRITE_BIT, or MAP_READ_BIT|MAP_WRITE_BIT.

- MAP_UNSYNCHRONIZED_BIT indicates that the GL should not attempt to synchronize pending operations on the buffer prior to returning from **Map*BufferRange**. No GL error is generated if pending operations which source or modify the buffer overlap the mapped region, but the result of such previous and any subsequent operations is undefined.

A successful **Map*BufferRange** sets buffer object state values as shown in table 6.4.

Errors

If an error occurs, a NULL pointer is returned.

An INVALID_ENUM error is generated by **MapBufferRange** if *target* is not one of the valid targets listed above.

An INVALID_OPERATION error is generated by **MapBufferRange** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **MapNamedBufferRange** if *buffer* is not the name of an existing buffer object.

An INVALID_VALUE error is generated if *offset* or *length* is negative, if $offset + length$ is greater than the value of BUFFER_SIZE, or if *access* has any bits set other than those defined above.

An INVALID_OPERATION error is generated for any of the following conditions:

- *length* is zero.
- The buffer is already in a mapped state.
- Neither MAP_READ_BIT nor MAP_WRITE_BIT is set.
- MAP_READ_BIT is set and any of MAP_INVALIDATE_RANGE_BIT, MAP_INVALIDATE_BUFFER_BIT, or MAP_UNSYNCHRONIZED_BIT is set.
- MAP_FLUSH_EXPLICIT_BIT is set and MAP_WRITE_BIT is not set.
- Any of MAP_READ_BIT, MAP_WRITE_BIT, MAP_PERSISTENT_BIT, or MAP_COHERENT_BIT are set, but the same bit is not set in the buffer's storage flags.

No error is generated if memory outside the mapped range is modified or queried, but the result is undefined and system errors (possibly including program termination) may occur.

The entire data store of a buffer object can be mapped into the client's address space with the commands

```
void *MapBuffer( enum target, enum access );
void *MapNamedBuffer( uint buffer, enum access );
```

These commands are respectively equivalent to

MapBufferRange(*target*, 0, length, flags);

and

MapNamedBufferRange(*buffer*, 0, length, flags);

where length is equal to the value of BUFFER_SIZE for the target buffer and flags is equal to

- MAP_READ_BIT, if *access* is READ_ONLY
- MAP_WRITE_BIT, if *access* is WRITE_ONLY
- MAP_READ_BIT | MAP_WRITE_BIT, if *access* is READ_WRITE.

The pointer value returned by **MapBuffer** and **MapNamedBuffer** must be aligned to the value of MIN_MAP_BUFFER_ALIGNMENT basic machine units.

Errors

An INVALID_ENUM error is generated if *access* is not READ_ONLY, WRITE_ONLY, or READ_WRITE.

Other errors are generated as described above for **MapBufferRange** and **MapNamedBufferRange**.

If a buffer object is mapped with the MAP_FLUSH_EXPLICIT_BIT flag, modifications to the mapped range may be indicated with the commands

```
void FlushMappedBufferRange( enum target, intptr offset,
    sizeiptr length );
void FlushMappedNamedBufferRange( uint buffer,
    intptr offset, sizeiptr length );
```

For **FlushMappedBufferRange**, the buffer object is that bound to *target*, which must be one of the targets listed in table 6.1. For **FlushMappedNamedBufferRange**, *buffer* is the name of the buffer object.

offset and *length* indicate a modified subrange of the mapping, in basic machine units. The specified subrange to flush is relative to the start of the currently mapped range of the buffer object. **FlushMapped*BufferRange** may be called multiple times to indicate distinct subranges of the mapping which require flushing.

If a buffer range is mapped with both MAP_PERSISTENT_BIT and MAP_FLUSH_EXPLICIT_BIT set, then **FlushMapped*BufferRange** may be called to ensure that data written by the client into the flushed region becomes visible to the server. Data written to a coherent store will always become visible to the server after an unspecified period of time.

Errors

An INVALID_ENUM error is generated by **FlushMappedBufferRange** if *target* is not one of the targets listed in table 6.1.

An INVALID_OPERATION error is generated by **FlushMappedBufferRange** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **FlushMappedNamedBufferRange** if *buffer* is not the name of an existing buffer object.

An INVALID_OPERATION error is generated if the buffer object is not mapped, or is mapped without the MAP_FLUSH_EXPLICIT_BIT flag.

An INVALID_VALUE error is generated if *offset* or *length* is negative, or if $offset + length$ exceeds the size of the mapping.

6.3.1 Unmapping Buffers

After the client has specified the contents of a mapped range of a buffer object, and before the data in that range are dereferenced by any GL commands, the mapping must be relinquished with one of the commands

```
boolean UnmapBuffer( enum target );
boolean UnmapNamedBuffer( uint buffer );
```

For **UnmapBuffer**, the buffer object is that bound to *target*, which must be one of the targets listed in table 6.1. For **UnmapNamedBuffer**, *buffer* is the name of the buffer object.

Unmapping a mapped buffer object invalidates the pointer to its data store and sets the object's BUFFER_MAPPED, BUFFER_MAP_POINTER, BUFFER_ACCESS_FLAGS, BUFFER_MAP_OFFSET, and BUFFER_MAP_LENGTH state variables to the initial values shown in table 6.3.

Unmap*Buffer returns TRUE unless data values in the buffer object's data store have become corrupted during the period that the buffer object was mapped. Such corruption can be the result of a screen resolution change or other window system-dependent event that causes system heaps such as those for high-performance graphics memory to be discarded. GL implementations must guarantee that such corruption can occur only during the periods that a buffer object's data store is mapped. If such corruption has occurred, **Unmap*Buffer** return FALSE, and the contents of the data store become undefined.

Unmapping that occurs as a side effect of buffer deletion (see section 5.1.2) or reinitialization by **BufferData** is not an error.

Buffer mappings are buffer object state, and are not affected by whether or not a context owing a buffer object is current.

If an error is generated, FALSE is returned.

Errors

An INVALID_ENUM error is generated by **UnmapBuffer** if *target* is not one of the targets listed in table 6.1.

An INVALID_OPERATION error is generated by **UnmapBuffer** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **UnmapNamedBuffer** if *buffer* is not the name of an existing buffer object.

An INVALID_OPERATION error is generated if the buffer object's data store is already in the unmapped state.

6.3.2 Effects of Mapping Buffers on Other GL Commands

Any GL command which attempts to read from, write to, or change the state of a buffer object may generate an INVALID_OPERATION error if all or part of the buffer object is mapped, unless it was allocated by a call to ***BufferStorage** with the MAP_PERSISTENT_BIT included in *flags*. However, only commands which explicitly describe this error are required to do so. If an error is not generated, such commands will have undefined results and may result in GL interruption or termination.

6.4 Effects of Accessing Outside Buffer Bounds

Most, but not all GL commands operating on buffer objects will detect attempts to read from or write to a location in a bound buffer object at an offset less than zero, or greater than or equal to the buffer's size. When such an attempt is detected, a GL error is generated. Any command which does not detect these attempts, and performs such an invalid read or write, has undefined results, and may result in GL interruption or termination.

Robust buffer access can be enabled by creating a context with robust access enabled through the window system binding APIs. When enabled, any command unable to generate a GL error as described above, such as buffer object accesses from the active program, will not read or modify memory outside of the data store of the buffer object and will not result in GL interruption or termination. Out-of-bounds reads may return values from within the buffer object or zero values. Out-of-bounds writes may modify values within the buffer object or be discarded. Accesses made through resources attached to binding points are only protected within the buffer object from which the binding point is declared. For example, for an out-of-bounds access to a member variable of a uniform block, the access protection is provided within the uniform buffer object, and not for the bound buffer range for this uniform block.

6.5 Invalidating Buffer Data

All or part of the data store of a buffer object may be invalidated by calling

```
void InvalidateBufferSubData( uint buffer, intptr offset,
    sizeiptr length );
```

with *buffer* set to the name of the buffer whose data store is being invalidated. *offset* and *length* specify the range of the data in the buffer object that is to be invalidated.

Data in the specified range have undefined values after calling **InvalidateBufferSubData**.

Errors

An INVALID_VALUE error is generated if *buffer* is zero or is not the name of an existing buffer object.

An INVALID_VALUE error is generated if *offset* or *length* is negative, or if $offset + length$ is greater than the value of BUFFER_SIZE for *buffer*.

An INVALID_OPERATION error is generated if *buffer* is currently mapped by **MapBuffer** or if the invalidate range intersects the range currently mapped by **MapBufferRange**, unless it was mapped with MAP_PERSISTENT_BIT set in the **MapBufferRange** *access* flags.

The command

```
void InvalidateBufferData( uint buffer );
```

is equivalent to calling **InvalidateBufferSubData** with *offset* equal to zero and *length* equal to the value of BUFFER_SIZE for *buffer*.

6.6 Copying Between Buffers

All or part of the data store of a buffer object may be copied to the data store of another buffer object with the commands

```
void CopyBufferSubData( enum readTarget, enum writeTarget,
    intptr readOffset, intptr writeOffset, sizeiptr size );
void CopyNamedBufferSubData( uint readBuffer,
    uint writeBuffer, intptr readOffset, intptr writeOffset,
    sizeiptr size );
```

For **CopyBufferSubData**, *readTarget* and *writeTarget* are the targets to which the source and destination buffers are bound, and each must be one of the targets listed in table 6.1. For **CopyNamedBufferSubData**, *readBuffer* and *writeBuffer* are the names of the source and destination buffers, respectively.

While any of these targets may be used, the COPY_READ_BUFFER and COPY_WRITE_BUFFER targets are provided specifically for copies, so that they can be done without affecting other buffer binding targets that may be in use.

writeOffset and *size* specify the range of data in the destination buffer object that is to be replaced, in terms of basic machine units. *readOffset* and *size* specify

the range of data in the source buffer object that is to be copied to the corresponding region of *writeTarget*.

Errors

An INVALID_OPERATION error is generated by **CopyBufferSubData** if zero is bound to *readTarget* or *writeTarget*.

An INVALID_ENUM error is generated by **CopyBufferSubData** if *readTarget* or *writeTarget* is not one of the targets listed in table 6.1.

An INVALID_OPERATION error is generated by **CopyNamedBufferSubData** if *readBuffer* or *writeBuffer* is not the name of an existing buffer object.

An INVALID_VALUE error is generated if any of *readOffset*, *writeOffset*, or *size* are negative, if $readOffset + size$ exceeds the size of the source buffer object, or if $writeOffset + size$ exceeds the size of the destination buffer object.

An INVALID_VALUE error is generated if the source and destination are the same buffer object, and the ranges $[readOffset, readOffset + size)$ and $[writeOffset, writeOffset + size)$ overlap.

An INVALID_OPERATION error is generated if either the source or destination buffer objects is mapped, unless they were mapped with MAP_PERSISTENT_BIT set in the **Map*BufferRange** *access* flags.

6.7 Buffer Object Queries

To query information about a buffer object, use the commands

```
void GetBufferParameteriv( enum target, enum pname,
   int *data );
void GetBufferParameteri64v( enum target, enum pname,
   int64 *data );
void GetNamedBufferParameteriv( uint buffer,
   enum pname, int *data );
void GetNamedBufferParameteri64v( uint buffer,
   enum pname, int64 *data );
```

For **GetBufferParameter***, the buffer object is that bound to *target*, which must be one of the targets listed in table 6.1. For **GetNamedBufferParameter***, *buffer* is the name of the buffer object.

pname must be one of the buffer object parameters in table 6.2, other than BUFFER_MAP_POINTER. The value of the specified parameter of the buffer object bound to *target* is returned in *data*.

Errors

An INVALID_ENUM error is generated by **GetBufferParameter*** if *target* is not one of the targets listed in table 6.1.

An INVALID_OPERATION error is generated by **GetBufferParameter*** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **GetNamedBufferParameter*** if *buffer* is not the name of an existing buffer object.

An INVALID_ENUM error is generated if *pname* is not one of the buffer object parameters other than BUFFER_MAP_POINTER.

To query the data store of a buffer object, use the commands

```
void GetBufferSubData( enum target, intptr offset,
    sizeiptr size, void *data );
void GetNamedBufferSubData( uint buffer, intptr offset,
    sizeiptr size, void *data );
```

For **GetBufferSubData**, *target* specifies the target to which the source buffer object is bound, and must be one of the values listed in table 6.1. For **GetNamedBufferSubData**, *buffer* specifies the name of the source buffer object.

offset and *size* indicate the range of data in the source buffer object that is to be queried, in terms of basic machine units. *data* specifies a region of client memory, *size* basic machine units in length, into which the data is to be retrieved.

Errors

An INVALID_ENUM error is generated by **GetBufferSubData** if *target* is not one of the targets listed in table 6.1.

An INVALID_OPERATION error is generated by **GetBufferSubData** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **GetNamedBufferSubData** if *buffer* is not the name of an existing buffer object.

An INVALID_VALUE error is generated if *offset* or *size* is negative, or if $offset + size$ is greater than the value of BUFFER_SIZE for the source buffer object.

An INVALID_OPERATION error is generated if the source buffer object is currently mapped, unless it was mapped with MAP_PERSISTENT_BIT set in the **Map*BufferRange** *access* flags.

While part or all of the data store of a buffer object is mapped, the pointer to the mapped range of the data store may be queried with the commands

```
void GetBufferPointerv( enum target, enum pname, const
    void **params );
void GetNamedBufferPointerv( uint buffer, enum pname,
    const void **params );
```

For **GetBufferPointerv**, the buffer object is that bound to *target*, which must be one of the targets listed in table 6.1. For **GetNamedBufferPointerv**, *buffer* is the name of the buffer object.

pname must be BUFFER_MAP_POINTER. The single buffer map pointer is returned in *params*. A NULL pointer value is returned if the buffer object's data store is not currently mapped; or if the requesting context did not map the buffer object's data store, and the implementation is unable to support mappings on multiple clients.

Errors

An INVALID_ENUM error is generated by **GetBufferPointerv** if *target* is not one of the targets listed in table 6.1.

An INVALID_OPERATION error is generated by **GetBufferPointerv** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **GetNamedBufferPointerv** if *buffer* is not the name of an existing buffer object.

An INVALID_ENUM error is generated if *pname* is not BUFFER_MAP_POINTER.

6.7.1 Indexed Buffer Object Limits and Binding Queries

Several types of buffer bindings support an indexed array of binding points for specific use by the GL, in addition to a single generic binding point for general management of buffers of that type. Each type of binding is described in table 6.5 together with the token names used to refer to each buffer in the array of binding points, the starting offset of the binding for each buffer in the array, any constraints on the corresponding *offset* value passed to **BindBufferRange** (see section 6.1.1), the size of the binding for each buffer in the array, any constraints on the corresponding *size* value passed to **BindBufferRange**, and the size of the array (the number of bind points supported).

To query which buffer objects are bound to an indexed array, call **GetIntegeri_v** with *target* set to the name of the array of binding points. *index* must be in the range zero to the number of bind points supported minus one. The name of the buffer object bound to *index* is returned in *values*. If no buffer object is bound for *index*, zero is returned in *values*.

Atomic counter array bindings (see sec. 7.7.2)	
binding points	ATOMIC_COUNTER_BUFFER_BINDING
starting offset	ATOMIC_COUNTER_BUFFER_START
offset restriction	multiple of 4
binding size	ATOMIC_COUNTER_BUFFER_SIZE
size restriction	none
no. of bind points	value of MAX_ATOMIC_COUNTER_BUFFER_BINDINGS
Shader storage array bindings (see sec. 7.8)	
binding points	SHADER_STORAGE_BUFFER_BINDING
starting offset	SHADER_STORAGE_BUFFER_START
offset restriction	multiple of value of SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT
binding size	SHADER_STORAGE_BUFFER_SIZE
size restriction	none
no. of bind points	value of MAX_SHADER_STORAGE_BUFFER_BINDINGS
Transform feedback array bindings (see sec. 13.3.2)	
binding points	TRANSFORM_FEEDBACK_BUFFER_BINDING
starting offset	TRANSFORM_FEEDBACK_BUFFER_START
offset restriction	multiple of 4
binding size	TRANSFORM_FEEDBACK_BUFFER_SIZE
size restriction	multiple of 4
no. of bind points	value of MAX_TRANSFORM_FEEDBACK_BUFFERS
Uniform buffer array bindings (see sec. 7.6.3)	
binding points	UNIFORM_BUFFER_BINDING
starting offset	UNIFORM_BUFFER_START
offset restriction	multiple of value of UNIFORM_BUFFER_OFFSET_ALIGNMENT
binding size	UNIFORM_BUFFER_SIZE
size restriction	none
no. of bind points	value of MAX_UNIFORM_BUFFER_BINDINGS

Table 6.5: Indexed buffer object limits and binding queries

To query the starting offset or size of the range of a buffer object binding in an indexed array, call **GetInteger64i_v** with *target* set to respectively the starting offset or binding size name from table 6.5 for that array. *index* must be in the range zero to the number of bind points supported minus one. If the starting offset or size was not specified when the buffer object was bound (e.g. if it was bound with **BindBufferBase**), or if no buffer object is bound to the *target* array at *index*, zero is returned[1].

Errors

An `INVALID_VALUE` error is generated by **GetIntegeri_v** and **GetInteger64i_v** if *target* is one of the array binding point names, starting offset names, or binding size names from table 6.5 and *index* is greater than or equal to the number of binding points for *target* as described in the same table.

6.8 Buffer Object State

The state required to support buffer objects consists of binding names for each of the buffer targets in table 6.1, and for each of the indexed buffer targets in section 6.1.1. The state required for index buffer targets for atomic counters, shader storage, transform feedback, and uniform buffer array bindings is summarized in tables 23.57, 23.58, 23.59, and 23.60 respectively.

Additionally, each vertex array has an associated binding so there is a buffer object binding for each of the vertex array, normal array, color array, index array, multiple texture coordinate arrays, edge flag array, secondary color array, fog coordinate array, and vertex attribute arrays. The initial values for all buffer object bindings is zero.

The state of each buffer object consists of a buffer size in basic machine units, a usage parameter, an access parameter, a boolean indicating whether or not buffer storage is immutable, an unsigned integer storing the flags with which it was allocated, a mapped boolean, two integers for the offset and size of the mapped region, a pointer to the mapped buffer (`NULL` if unmapped), and the sized array of basic machine units for the buffer data.

[1] A zero size is a sentinel value indicating that the actual binding range size is determined by the size of the bound buffer at the time the binding is used.

Chapter 7

Programs and Shaders

This chapter specifies commands to create, manage, and destroy program and shader objects. Commands and functionality applicable only to specific shader stages (for example, vertex attributes used as inputs by vertex shaders) are described together with those stages in chapters 10 and 15.

A *shader* specifies operations that are meant to occur on data as it moves through different programmable stages of the OpenGL processing pipeline, starting with vertices specified by the application and ending with fragments prior to being written to the framebuffer. The programming language used for shaders is described in the OpenGL Shading Language Specification.

To use a shader, shader source code is first loaded into a *shader object* and then *compiled*. A shader object corresponds to a stage in the rendering pipeline referred to as its *shader stage* or *shader type*.

Alternatively, pre-compiled shader binary code can be loaded into a shader object. A SPIR-V module can also be associated with a shader and then *specialized*. An implementation must support shader compilation (the boolean value `SHADER_COMPILER` must be `TRUE`). If the integer value of `NUM_SHADER_BINARY_FORMATS` is greater than zero, then shader binary loading is supported.

One or more shader objects are attached to a *program object*. The program object is then *linked*, which generates executable code from all the compiled shader objects attached to the program. Alternatively, pre-compiled program binary code may be directly loaded into a program object (see section 7.5).

When program objects are bound to a shader stage, they become the *current program object* for that stage. When the current program object for a shader stage includes a shader of that type, it is considered the *active program object* for that stage.

The current program object for all stages may be set at once using a single

unified program object, or the current program object may be set for each stage individually using a *separable program object* where different separable program objects may be current for other stages. The set of separable program objects current for all stages are collected in a program pipeline object that must be bound for use. When a linked program object is made active for one of the stages, the corresponding executable code is used to perform processing for that stage.

Shader stages including *vertex shader*s, *tessellation control shader*s, *tessellation evaluation shader*s, *geometry shader*s, *fragment shader*s, and *compute shader*s can be created, compiled, and linked into program objects.

Vertex shaders describe the operations that occur on vertex attributes. Tessellation control and evaluation shaders are used to control the operation of the tessellator, and are described in section 11.2. Geometry shaders affect the processing of primitives assembled from vertices (see section 11.3). Fragment shaders affect the processing of fragments during rasterization (see section 15). A single program object can contain all of these shaders, or any subset thereof.

Compute shaders perform general-purpose computation for dispatched arrays of shader invocations (see section 19), but do not operate on primitives processed by the other shader types.

Shaders can reference several types of variables as they execute. *Uniforms* are per-program variables that are constant during program execution (see section 7.6). *Buffer variables* (see section 7.8) are similar to uniforms, but are stored in buffer object memory which may be written to, and is persistent across multiple shader invocations. *Subroutine uniform variables* (see section 7.10) are similar to uniforms but are context state, rather than program object state. *Samplers* (see section 7.11) are a special form of uniform used for texturing (see chapter 8). *Images* (see section 7.12) are a special form of uniform identifying a level of a texture to be accessed using built-in shader functions as described in section 8.26. *Output variables* hold the results of shader execution that are used later in the pipeline. Each of these variable types is described in more detail below.

7.1 Shader Objects

The name space for shader objects is the unsigned integers, with zero reserved for the GL. This name space is shared with program objects. The following sections define commands that operate on shader and program objects.

To create a shader object, use the command

`uint` **CreateShader**(`enum` *type*);

type	Shader Stage
VERTEX_SHADER	Vertex shader
TESS_CONTROL_SHADER	Tessellation control shader
TESS_EVALUATION_SHADER	Tessellation evaluation shader
GEOMETRY_SHADER	Geometry shader
FRAGMENT_SHADER	Fragment shader
COMPUTE_SHADER	Compute shader

Table 7.1: **CreateShader** *type* values and the corresponding shader stages.

The shader object is empty when it is created. The *type* argument specifies the type of shader object to be created and must be one of the values in table 7.1 indicating the corresponding shader stage. A non-zero name that can be used to reference the shader object is returned.

Errors

An INVALID_ENUM error is generated and zero is returned if *type* is not one of the values in table 7.1.

The command

```
void ShaderSource( uint shader, sizei count, const
   char * const *string, const int *length );
```

loads source code into the shader object named *shader*. *string* is an array of *count* pointers to optionally null-terminated character strings that make up the source code. The *length* argument is an array with the number of chars in each string (the string length). If an element in *length* is negative, its accompanying string is null-terminated. If *length* is NULL, all strings in the *string* argument are considered null-terminated. The **ShaderSource** command sets the source code for the *shader* to the text strings in the *string* array. If *shader* previously had source code loaded into it, the existing source code is completely replaced. Any length passed in excludes the null terminator in its count.

The strings that are loaded into a shader object are expected to form the source code for a valid shader as defined in the OpenGL Shading Language Specification.

If *shader* was previously associated with a SPIR-V module (via the **ShaderBinary** command), that association is broken. Upon successful completion of this command the SPIR_V_BINARY state of *shader* is set to FALSE.

Errors

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

An INVALID_VALUE error is generated if *count* is negative.

Once the source code for a shader has been loaded, a shader object can be compiled with the command

`void` **CompileShader**(`uint` *shader*);

Each shader object has a boolean status, COMPILE_STATUS, that is modified as a result of compilation. This status may be queried with **GetShaderiv** (see section 7.14). This status will be set to TRUE if *shader* was compiled without errors and is ready for use, and FALSE otherwise. Compilation can fail for a variety of reasons as listed in the OpenGL Shading Language Specification. If **CompileShader** failed, any information about a previous compile is lost. Thus a failed compile does not restore the old state of *shader*.

Changing the source code of a shader object with **ShaderSource** does not change its compile status or the compiled shader code.

Each shader object has an information log, which is a text string that is overwritten as a result of compilation. This information log may be queried with **GetShaderInfoLog** to obtain more information about the compilation attempt (see section 7.14).

Errors

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

An INVALID_OPERATION error is generated if the SPIR_V_BINARY state of *shader* is TRUE.

Resources allocated by the shader compiler may be released with the command

`void` **ReleaseShaderCompiler**(`void`);

This is a hint from the application, and does not prevent later use of the shader compiler. If shader source is loaded and compiled after **ReleaseShaderCompiler**

has been called, **CompileShader** must succeed provided there are no errors in the shader source.

The range and precision for different numeric formats supported by the shader compiler may be determined with the command **GetShaderPrecisionFormat** (see section 7.14).

Shader objects can be deleted with the command

`void` **DeleteShader**(`uint` *shader*);

If *shader* is not attached to any program object, it is deleted immediately. Otherwise, *shader* is flagged for deletion and will be deleted when it is no longer attached to any program object. If an object is flagged for deletion, its boolean status bit `DELETE_STATUS` is set to true. The value of `DELETE_STATUS` may be queried with **GetShaderiv** (see section 7.14). **DeleteShader** will silently ignore the value zero.

Errors

An `INVALID_VALUE` error is generated if *shader* is neither zero nor the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *shader* is not zero and is the name of a program object.

The command

`boolean` **IsShader**(`uint` *shader*);

returns `TRUE` if *shader* is the name of a shader object. If *shader* is zero, or a non-zero value that is not the name of a shader object, **IsShader** returns `FALSE`. No error is generated if *shader* is not a valid shader object name.

7.2 Shader Binaries

Precompiled shader binaries may be loaded with the command

`void` **ShaderBinary**(`sizei` *count*, `const uint` **shaders*,
`enum` *binaryformat*, `const void` **binary*, `sizei` *length*);

shaders contains a list of *count* shader object handles. Each handle refers to a unique shader type, and may correspond to any of the shader stages in table 7.1.

binary points to *length* bytes of pre-compiled binary shader code in client memory, and *binaryformat* denotes the format of the pre-compiled code.

The binary image will be decoded according to the extension specification defining the specified *binaryformat*.

GL defines an execution environment for shaders created from SPIR-V modules. To load a SPIR-V binary into GL, set *binaryformat* to SHADER_BINARY_FORMAT_SPIR_V. *binary* should point to the start of a valid SPIR-V module binary and *length* should contain the length of that binary, in bytes. Upon successful consumption of the SPIR-V module:

- each entry of *shaders* will be associated with that SPIR-V module,
- the SPIR_V_BINARY state of each shader is set to TRUE,
- the COMPILE_STATUS of each of these shaders is set to FALSE,
- any existing source string (specified by **ShaderSource**) is removed, and
- any information about a previous compile is lost.

Shaders associated with SPIR-V modules must be finalized by calling **SpecializeShader**, as described in section 7.2.1.

GL also provides a mechanism to obtain token values for such formats provided by extensions. The number of binary formats supported can be obtained by querying the value of NUM_SHADER_BINARY_FORMATS. The list of specific binary formats supported can be obtained by querying the value of SHADER_BINARY_FORMATS.

Depending on the types of the shader objects in *shaders*, **ShaderBinary** will individually load binary shaders, or load an executable binary that contains an optimized set of shaders stored in the same binary.

Errors

An INVALID_VALUE error is generated if *count* or *length* is negative.

An INVALID_ENUM error is generated if *binaryformat* is not a supported format returned in SHADER_BINARY_FORMATS.

An INVALID_VALUE error is generated if the data pointed to by *binary* does not match the specified *binaryformat*.

An INVALID_VALUE error is generated if any of the handles in *shaders* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if any of the handles in *shaders* is the name of a program object.

An INVALID_OPERATION error is generated if *binaryformat* is not SHADER_BINARY_FORMAT_SPIR_V and more than one of the handles in *shaders* refers to the same type of shader object.

Additional errors corresponding to specific binary formats may be generated as specified by the extensions defining those formats.

If **ShaderBinary** fails, the old state of shader objects for which the binary was being loaded will not be restored.

Note that if shader binary interfaces are supported, then a GL implementation may require that an optimized set of shader binaries that were compiled together be specified to **LinkProgram**. Not specifying an optimized set may cause **LinkProgram** to fail.

7.2.1 Shader Specialization

Shaders associated with SPIR-V modules must be specialized before they can be linked into a program object. It is not necessary to specialize the shader before it is attached to a program object. Once specialized, a shader may not be specialized again without first re-associating the original SPIR-V module with it, through **ShaderBinary**.

Specialization does two things:

- Selects the name of the entry point, for that shader's stage, from the SPIR-V module.
- Sets the values of all, or a subset of, the specialization constants in the SPIR-V module.

To specialize a shader created from a SPIR-V module, call:

```
void SpecializeShader( uint shader, const
    char *pEntryPoint, uint numSpecializationConstants,
    const uint *pConstantIndex, const
    uint *pConstantValue );
```

shader is the name of a shader object containing unspecialized SPIR-V as created from a successful call to **ShaderBinary** to which a SPIR-V module was passed. *pEntryPoint* is a pointer to a null-terminated UTF-8 string specifying the name of the entry point in the SPIR-V module to use for this shader. *numSpecializationConstants* is the number of specialization constants whose values to set in this call. *pConstantIndex* is a pointer to an array of *numSpecializationConstants*

unsigned integers, each holding the index of a specialization constant in the SPIR-V module whose value to set. The corresponding entry in *pConstantValue* is used to set the value of the specialization constant indexed by the entry in *pConstantIndex*. Although this array is of unsigned integer, each entry is bitcast to the appropriate type for the module, and therefore, floating-point constants may be set by including their IEEE-754 bit representation in the *pConstantValue* array. Specialization constants not referenced by *pConstantIndex* retain their default values as specified in the SPIR-V module.

On successful shader specialization, the compile status for *shader* is set to `TRUE`. On failure, the compile status for *shader* is set to `FALSE` and additional information about the cause of the failure may be available in the shader compilation log. Specialization can fail if the SPIR-V module fails to meet the requirements listed in appendix C.

Errors

An `INVALID_VALUE` error is generated if *shader* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *shader* is the name of a program object.

An `INVALID_OPERATION` error is generated if the value of `SPIR_V_BINARY` for *shader* is not `TRUE`, or if the shader has already been specialized.

An `INVALID_VALUE` error is generated if *pEntryPoint* does not match the `Name` of any `OpEntryPoint` declaration in the SPIR-V module associated with *shader*.

An `INVALID_OPERATION` error is generated if the `ExecutionMode` of the `OpEntryPoint` indicated by *pEntryPoint* does not match the type of *shader*.

An `INVALID_VALUE` error is generated if any element of *pConstantIndex* refers to a specialization constant that does not exist in the shader module contained in *shader*.

7.3 Program Objects

A program object is created with the command

```
uint CreateProgram( void );
```

Program objects are empty when they are created. A non-zero name that can be used to reference the program object is returned. If an error occurs, zero will be returned.

To attach a shader object to a program object, use the command

`void` **AttachShader**(`uint` *program*, `uint` *shader*);

Shader objects may be attached to program objects before source code has been loaded into the shader object, or before the shader object has been compiled or specialized. Multiple shader objects of the same type may be attached to a single program object, and a single shader object may be attached to more than one program object.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_VALUE` error is generated if *shader* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *shader* is the name of a program object.

An `INVALID_OPERATION` error is generated if *shader* is already attached to *program*.

To detach a shader object from a program object, use the command

`void` **DetachShader**(`uint` *program*, `uint` *shader*);

If *shader* has been flagged for deletion and is not attached to any other program object, it is deleted.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_VALUE` error is generated if *shader* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *shader* is the name of a program object.

An `INVALID_OPERATION` error is generated if *shader* is not attached to

program.

In order to use the shader objects contained in a program object, the program object must be linked. The command

`void` **LinkProgram**(`uint` *program*);

will link the program object named *program*. Each program object has a boolean status, `LINK_STATUS`, that is modified as a result of linking. This status may be queried with **GetProgramiv** (see section 7.14). This status will be set to `TRUE` if a valid executable is created, and `FALSE` otherwise.

It is possible to link successfully compiled or specialized shader objects built from any version of the OpenGL Shading Language or OpenGL ES Shading Language supported by the implementation, and not all shader objects linked together have to use the same version of the shading language.

Linking can fail for a variety of reasons as specified in the shading language specification for source shaders, or if the requirements in appendix C are not met for SPIR-V shaders, as well as any of the following reasons:

- One or more of the shader objects attached to *program* are not compiled or specialized successfully.
- More active uniform or active sampler variables are used in *program* than allowed (see sections 7.6, 7.11, and 11.3.3).
- *program* contains objects to form a tessellation control shader (see section 11.2.1), and
  - the program is not separable and contains no objects to form a vertex shader;
  - the output patch vertex count is not specified in any compiled tessellation control shader object; or
  - the output patch vertex count is specified differently in multiple tessellation control shader objects.
- *program* contains objects to form a tessellation evaluation shader (see section 11.2.3), and
  - the program is not separable and contains no objects to form a vertex shader;

  - the tessellation primitive mode is not specified in any compiled tessellation evaluation shader object; or
  - the tessellation primitive mode, spacing, vertex order, or point mode is specified differently in multiple tessellation evaluation shader objects.

- *program* contains objects to form a geometry shader (see section 11.3), and
  - the program is not separable and contains no objects to form a vertex shader;
  - the input primitive type, output primitive type, or maximum output vertex count is not specified in any compiled geometry shader object; or
  - the input primitive type, output primitive type, or maximum output vertex count is specified differently in multiple geometry shader objects.

- *program* contains objects to form a compute shader (see section 19) and,
  - *program* also contains objects to form any other type of shader.

- All the shader objects attached to *program* do not have the same value for the `SPIR_V_BINARY` state.

If **LinkProgram** failed, any information about a previous link of that program object is lost. Thus, a failed link does not restore the old state of *program*.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

When program objects which have been linked successfully are used for rendering operations, they may access GL state and interface with other stages of the GL pipeline through *active variables* and *active interface blocks*. The GL provides various commands allowing applications to enumerate and query properties of active variables and interface blocks for a specified program. If one of these commands is called with a program for which **LinkProgram** succeeded, the information recorded when the program was linked is returned. If one of these commands is called with a program for which **LinkProgram** failed, no error is generated unless otherwise noted. Implementations may return information on variables and interface blocks that would have been active had the program been linked successfully.

In cases where the link failed because the program required too many resources, these commands may help applications determine why limits were exceeded. However, the information returned in this case is implementation-dependent and may be incomplete. If one of these commands is called with a program for which **LinkProgram** had never been called, no error is generated unless otherwise noted, and the program object is considered to have no active variables or interface blocks.

Each program object has an information log that is overwritten as a result of a link operation. This information log may be queried with **GetProgramInfoLog** to obtain more information about the link operation or the validation information (see section 7.14).

If a program has been linked successfully by **LinkProgram** or loaded by **ProgramBinary** (see section 7.5), it can be made part of the current rendering state for all shader stages with the command

`void` **UseProgram**(`uint` *program*);

If *program* is non-zero, this command will make *program* the current program object. This will install executable code as part of the current rendering state for each shader stage present when the program was last linked successfully. If **UseProgram** is called with *program* set to zero, then there is no current program object.

The executable code for an individual shader stage is taken from the current program for that stage. If there is a current program object established by **UseProgram**, that program is considered current for all stages. Otherwise, if there is a bound program pipeline object (see section 7.4), the program bound to the appropriate stage of the pipeline object is considered current. If there is no current program object or bound program pipeline object, no program is current for any stage. The current program for a stage is considered *active* if it contains executable code for that stage; otherwise, no program is considered active for that stage. If there is no active program for the vertex or fragment shader stages, fixed-function vertex and/or fragment processing will be used to process vertices and/or fragments. If there is no active program for the tessellation control, tessellation evaluation, or geometry shader stages, those stages are ignored. If there is no active program for the compute shader stage, compute dispatches will generate an error. The active program for the compute shader stage has no effect on the processing of vertices, geometric primitives, and fragments, and the active program for all other shader stages has no effect on compute dispatches.

Errors

An `INVALID_VALUE` error is generated if *program* is neither zero nor the

name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is not zero and is the name of a shader object.

An INVALID_OPERATION error is generated if *program* has not been linked successfully. The current rendering state is not modified.

While a program object is in use, applications are free to modify attached shader objects, compile attached shader objects, attach additional shader objects, and detach shader objects. These operations do not affect the link status or executable code of the program object.

If **LinkProgram** or **ProgramBinary** successfully re-links a program object that is active for any shader stage, then the newly generated executable code will be installed as part of the current rendering state for all shader stages where the program is active. Additionally, the newly generated executable code is made part of the state of any program pipeline for all stages where the program is attached.

If a program object that is active for any shader stage is re-linked unsuccessfully, the link status will be set to FALSE, but any existing executables and associated state will remain part of the current rendering state until a subsequent call to **UseProgram**, **UseProgramStages**, or **BindProgramPipeline** removes them from use. If such a program is attached to any program pipeline object, the existing executables and associated state will remain part of the program pipeline object until a subsequent call to **UseProgramStages** removes them from use. A program which has not been linked successfully may not be made part of the current rendering state by **UseProgram** or added to program pipeline objects by **UseProgramStages** until it is re-linked successfully. If such a program was attached to a program pipeline at the time of a failed link, its existing executable may still be made part of the current rendering state indirectly by **BindProgramPipeline**.

To set a program object parameter, call

```
void ProgramParameteri( uint program, enum pname,
    int value );
```

pname identifies which parameter to set for *program*. *value* holds the value being set.

If *pname* is PROGRAM_SEPARABLE, *value* must be TRUE or FALSE, and indicates whether *program* can be bound for individual pipeline stages using **UseProgramStages** after it is next linked.

If *pname* is PROGRAM_BINARY_RETRIEVABLE_HINT, *value* must be TRUE or FALSE, and indicates whether a program binary is likely to be retrieved later, as described for **ProgramBinary** in section 7.5.

State set with this command does not take effect until after the next time **LinkProgram** or **ProgramBinary** is called successfully.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_ENUM` error is generated if *pname* is not `PROGRAM_SEPARABLE` or `PROGRAM_BINARY_RETRIEVABLE_HINT`.

An `INVALID_VALUE` error is generated if *value* is not `TRUE` or `FALSE`.

Program objects can be deleted with the command

`void` **DeleteProgram**(`uint` *program*);

If *program* is not current for any GL context, is not the active program for any program pipeline object, and is not the current program for any stage of any program pipeline object, it is deleted immediately. Otherwise, *program* is flagged for deletion and will be deleted after all of these conditions become true. When a program object is deleted, all shader objects attached to it are detached. **DeleteProgram** will silently ignore the value zero.

Errors

An `INVALID_VALUE` error is generated if *program* is neither zero nor the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is not zero and is the name of a shader object.

The command

`boolean` **IsProgram**(`uint` *program*);

returns `TRUE` if *program* is the name of a program object. If *program* is zero, or a non-zero value that is not the name of a program object, **IsProgram** returns `FALSE`. No error is generated if *program* is not a valid program object name.

The command

`uint` **CreateShaderProgramv**(`enum` *type*, `sizei` *count*,
`const char * const` **strings*);

creates a stand-alone program from an array of null-terminated source code strings for a single shader type. **CreateShaderProgramv** is equivalent (assuming no errors are generated) to:

```
const uint shader = CreateShader(type);
if (shader) {
    ShaderSource(shader, count, strings, NULL);
    CompileShader(shader);
    const uint program = CreateProgram();
    if (program) {
        int compiled = FALSE;
        GetShaderiv(shader, COMPILE_STATUS, &compiled);
        ProgramParameteri(program, PROGRAM_SEPARABLE, TRUE);
        if (compiled) {
            AttachShader(program, shader);
            LinkProgram(program);
            DetachShader(program, shader);
        }
        append-shader-info-log-to-program-info-log
    }
    DeleteShader(shader);
    return program;
} else {
    return 0;
}
```

Because no shader is returned by **CreateShaderProgramv** and the shader that is created is deleted in the course of the command sequence, the info log of the shader object is copied to the program so the shader's failed info log for the failed compilation is accessible to the application.

If an error is generated, zero is returned.

Errors

An INVALID_ENUM error is generated if *type* is not one of the values in table 7.1.

An INVALID_VALUE error is generated if *count* is negative.

Other errors are generated if the supplied shader code fails to compile and link, as described for the commands in the pseudocode sequence above, but all such errors are generated without any side effects of executing those

commands.

7.3.1 Program Interfaces

When a program object is made part of the current rendering state, its executable code may communicate with other GL pipeline stages or application code through a variety of *interfaces*. When a program is linked, the GL builds a list of *active resources* for each interface. Examples of active resources include variables, interface blocks, and subroutines used by shader code. Resources referenced in shader code are considered *active* unless the compiler and linker can conclusively determine that they have no observable effect on the results produced by the executable code of the program. For example, variables might be considered inactive if they are declared but not used in executable code, used only in a clause of an `if` statement that would never be executed, used only in functions that are never called, or used only in computations of temporary variables having no effect on any shader output. In cases where the compiler or linker cannot make a conclusive determination, any resource referenced by shader code will be considered active. The set of active resources for any interface is implementation-dependent because it depends on various analysis and optimizations performed by the compiler and linker.

If a program is linked successfully, the GL will generate lists of active resources based on the executable code produced by the link. If a program is not linked successfully, the link may have failed for a number of reasons, including cases where the program required more resources than supported by the implementation. Implementations are permitted, but not required, to record lists of resources that would have been considered active had the program linked successfully. If an implementation does not record information for any given interface, the corresponding list of active resources is considered empty. If a program has never been linked, all lists of active resources are considered empty.

The GL provides a number of commands to query properties of the interfaces of a program object. Each such command accepts a *programInterface* token, identifying a specific interface. The supported values for *programInterface* are as follows:

- `UNIFORM` corresponds to the set of active uniform variables (see section 7.6) used by *program*.
- `UNIFORM_BLOCK` corresponds to the set of active uniform blocks (see section 7.6) used by *program*.
- `ATOMIC_COUNTER_BUFFER` corresponds to the set of active atomic counter buffer binding points (see section 7.6) used by *program*.

- PROGRAM_INPUT corresponds to the set of active input variables used by the first shader stage of *program*. If *program* includes multiple shader stages, input variables from any shader stage other than the first will not be enumerated.

- PROGRAM_OUTPUT corresponds to the set of active output variables (see section 11.1.2.1) used by the last shader stage of *program*. If *program* includes multiple shader stages, output variables from any shader stage other than the last will not be enumerated.

- VERTEX_SUBROUTINE, TESS_CONTROL_SUBROUTINE, TESS_EVALUATION_SUBROUTINE, GEOMETRY_SUBROUTINE, FRAGMENT_SUBROUTINE, and COMPUTE_SUBROUTINE correspond to the set of active subroutines for the vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shader stages of *program*, respectively (see section 7.10).

- VERTEX_SUBROUTINE_UNIFORM, TESS_CONTROL_SUBROUTINE_UNIFORM, TESS_EVALUATION_SUBROUTINE_UNIFORM, GEOMETRY_SUBROUTINE_UNIFORM, FRAGMENT_SUBROUTINE_UNIFORM, and COMPUTE_SUBROUTINE_UNIFORM correspond to the set of active subroutine uniform variables used by the vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shader stages of *program*, respectively (see section 7.10).

- TRANSFORM_FEEDBACK_VARYING corresponds to the set of output variables in the last non-fragment stage of *program* that would be captured when transform feedback is active (see section 13.3.3). The resources enumerated by this query are listed as specified by the most recent call to **TransformFeedbackVaryings** before the last call to **LinkProgram**. When the resource names an output array variable either a single element of the array or the whole array is captured. If the variable name is specified with an array index syntax "name[x]", name is the name of the array resource and x is the constant-integer index of the element captured. If the resource name is an array and has no array index and square bracket, then the whole array is captured.

- TRANSFORM_FEEDBACK_BUFFER corresponds to the set of active buffer binding points to which output variables in the TRANSFORM_FEEDBACK_VARYING interface are written.

- `BUFFER_VARIABLE` corresponds to the set of active buffer variables used by *program* (see section 7.8).
- `SHADER_STORAGE_BLOCK` corresponds to the set of active shader storage blocks used by *program* (see section 7.8)

7.3.1.1 Naming Active Resources

When building a list of active variable or interface blocks, resources with aggregate types (such as arrays or structures) may produce multiple entries in the active resource list for the corresponding interface. Additionally, each active variable, interface block, or subroutine in the list is assigned an associated name string that can be used by applications to refer to the resource.

For interfaces enumerating active variables, resource list entries for variables declared outside interface blocks are generated as follows:

- For an active variable declared as a single instance of a basic type, a single entry will be generated, using the variable name from the shader source.
- For an active variable declared as an array of basic types (e.g. not an array of structures or an array of arrays), a single entry will be generated, with its name string formed by concatenating the name of the array and the string `"[0]"`.
- For an active variable declared as a structure, a separate entry will be generated for each active structure member. The name of each entry is formed by concatenating the name of the structure, the `"."` character, and the name of the structure member. If a structure member to enumerate is itself a structure or array, these enumeration rules are applied recursively.
- For an active variable declared as an array of an aggregate data type (structures or arrays), a separate entry will be generated for each active array element, unless noted immediately below. The name of each entry is formed by concatenating the name of the array, the `"["` character, an integer identifying the element number, and the `"]"` character. These enumeration rules are applied recursively, treating each enumerated array element as a separate active variable.

For interfaces enumerating active variables, resource list entries for active members of interface blocks are generated as follows:

- For active members of an interface block with no instance name, resource list entries will be generated by treating the block member as though it were declared as a variable outside an interface block. The name strings for these entries will not include the block name.

- For active members of an interface block with an instance name, resource list entries will be generated by applying the rules for variables declared outside an interface block. The name string for each such entry is formed by concatenating the name of the interface block (not the instance name), the "`.`" character, and the name string that would be generated for an equivalent variable declared outside an interface block.

- For active members of an array of interface blocks, entries will be generated using the same rules as for members of a single interface block with an instance name. There will not be separate entries for each instance of the interface block, and the name strings will not contain any text like "`[0]`" to indicate that the member belongs to an array of interface blocks.

- For active shader storage block members that are declared as arrays of an aggregate type, entries will not be generated for each array element. Such block members are referred to as *top-level array*s and will generate entries only for the first array element.

For interfaces enumerating active interface blocks, the entries of active resource lists are generated as follows:

- For an active interface block not declared as an array of block instances, a single entry will be generated, using the block name from the shader source.

- For an active interface block declared as an array of arrays, a separate entry will be generated for each active instance. The name of each instance is formed by concatenating the block name, the "`[`" character, an integer identifying the instance number, and the "`]`" character. These enumeration rules are applied recursively, treating each enumerated array element as a separate active interface block.

For interfaces enumerating active subroutines, a single entry will be generated for each active subroutine, using the subroutine name from the shader source.

When an integer array element or block instance number is part of the name string, it will be specified in decimal form without a "`+`" or "`-`" sign or any extra leading zeroes. Additionally, the name string will not include white space anywhere in the string.

For shaders constructed from SPIR-V binaries (that is with a state of SPIR_V_BINARY equal to TRUE), variables may not have names associated with them, as the OpName and OpMemberName debug instructions are optional and may not be present in a SPIR-V module. When the Op*Name instructions are present, it is implementation-dependent if these are reported via the name reflection APIs. If no *name reflection information* is available, the name string associated with each active variable is the empty string (""). In this case, any queries that operate with a name as input will return INVALID_INDEX or -1 as appropriate, and any queries that return information about the name of a resource will report a name length of one (for the null character) and return an empty string with a length of zero.

The order of the active resource list is implementation-dependent for all interfaces except for TRANSFORM_FEEDBACK_VARYING. If variables in the TRANSFORM_FEEDBACK_VARYING interface were specified using the **TransformFeedbackVaryings** command, the active resource list will be arranged in the variable order specified in the most recent call to **TransformFeedbackVaryings** before the last call to **LinkProgram**. If variables in the TRANSFORM_FEEDBACK_VARYING interface were specified using layout qualifiers in shader code, the order of the active resource list is implementation-dependent.

For the ATOMIC_COUNTER_BUFFER interface, the list of active buffer binding points is built by identifying each unique binding point associated with one or more active atomic counter uniform variables. Active atomic counter buffers do not have an associated name string.

For the UNIFORM, PROGRAM_INPUT, PROGRAM_OUTPUT, and TRANSFORM_FEEDBACK_VARYING interfaces, the active resource list will include all active variables for the interface, including any active built-in variables.

For PROGRAM_INPUT and PROGRAM_OUTPUT interfaces for shaders that recieve or produce patch primitves, the active resource list will include both per-vertex and per-patch inputs and outputs.

For the TRANSFORM_FEEDBACK_BUFFER interface, the list of active buffer binding points is built by identifying each unique binding point to which one or more active output variables will be written in transform feedback mode. Active transform feedback buffers do not have an associated name string.

For the TRANSFORM_FEEDBACK_VARYING interface, the active resource list will include entries for the special variable names gl_NextBuffer, gl_SkipComponents1, gl_SkipComponents2, gl_SkipComponents3, and gl_SkipComponents4 (see section 11.1.2.1). These variables are used to control how output values are written to transform feedback buffers. When enumerating the properties of such resources, these variables are considered to have a TYPE of NONE and an ARRAY_SIZE of 0 (gl_NextBuffer), 1, 2, 3, and 4, respectively.

When a program is linked successfully, active variables in the UNIFORM,

PROGRAM_INPUT, PROGRAM_OUTPUT, or any of the subroutine uniform interfaces, are assigned one or more signed integer *locations*. These locations can be used by commands to assign values to uniforms and subroutine uniforms, to identify generic vertex attributes associated with vertex shader inputs, or to identify fragment color output numbers and indices associated with fragment shader outputs. For such variables declared as arrays, separate locations will be assigned to each active array element and are not required to be sequential. The location for "a[1]" may or may not be equal to the location for "a[0]" +1. Furthermore, since unused elements at the end of uniform arrays may be trimmed, the location of the $i + 1$'th array element may not be valid even if the location of the i'th element is valid. As a direct consequence, the value of the location of "a[0]" +1 may refer to a different uniform entirely. Applications that wish to set individual array elements should query the locations of each element separately.

Not all active variables are assigned valid locations; the following variables will have an effective location of -1:

- uniforms declared as atomic counters
- members of a uniform block
- built-in inputs, outputs, and uniforms (starting with gl_)
- inputs (except for vertex shader inputs) not declared with a location layout qualifier
- outputs (except for fragment shader outputs) not declared with a location layout qualifier

If a program has not been linked successfully, no locations will be assigned.

The command

```
void GetProgramInterfaceiv( uint program,
   enum programInterface, enum pname, int *params );
```

queries a property of the interface *programInterface* in program *program*, returning its value in *params*. The property to return is specified by *pname*.

If *pname* is ACTIVE_RESOURCES, the value returned is the number of resources in the active resource list for *programInterface*. If the list of active resources for *programInterface* is empty, zero is returned.

If *pname* is MAX_NAME_LENGTH, the value returned is the length of the longest active name string for an active resource in *programInterface*. This length includes

an extra character for the null terminator. If the list of active resources for *programInterface* is empty, zero is returned.

If *pname* is MAX_NUM_ACTIVE_VARIABLES, the value returned is the number of active variables belonging to the interface block or atomic counter buffer resource in *programInterface* with the most active variables. If the list of active resources for *programInterface* is empty, zero is returned.

If *pname* is MAX_NUM_COMPATIBLE_SUBROUTINES, the value returned is the number of compatible subroutines for the active subroutine uniform in *programInterface* with the most compatible subroutines. If the list of active resources for *programInterface* is empty, zero is returned.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *programInterface* is not one of the interfaces described in the introduction to section 7.3.1.

An INVALID_ENUM error is generated if *pname* is not ACTIVE_RESOURCES, MAX_NAME_LENGTH, MAX_NUM_ACTIVE_VARIABLES, or MAX_NUM_COMPATIBLE_SUBROUTINES.

An INVALID_OPERATION error is generated if *pname* is MAX_NAME_LENGTH and *programInterface* is ATOMIC_COUNTER_BUFFER or TRANSFORM_FEEDBACK_BUFFER, since active atomic counter and transform feedback buffer resources are not assigned name strings.

An INVALID_OPERATION error is generated if *pname* is MAX_NUM_ACTIVE_VARIABLES and *programInterface* is not ATOMIC_COUNTER_BUFFER, SHADER_STORAGE_BLOCK, TRANSFORM_FEEDBACK_BUFFER, or UNIFORM_BLOCK.

An INVALID_OPERATION error is generated if *pname* is MAX_NUM_COMPATIBLE_SUBROUTINES and *programInterface* is not VERTEX_SUBROUTINE_UNIFORM, TESS_CONTROL_SUBROUTINE_UNIFORM, TESS_EVALUATION_SUBROUTINE_UNIFORM, GEOMETRY_SUBROUTINE_UNIFORM, FRAGMENT_SUBROUTINE_UNIFORM, or COMPUTE_SUBROUTINE_UNIFORM.

Each entry in the active resource list for an interface is assigned a unique unsigned integer index in the range zero to $N - 1$, where N is the number of entries in the active resource list. The command

```
uint GetProgramResourceIndex( uint program,
    enum programInterface, const char *name );
```

returns the unsigned integer index assigned to a resource named *name* in the interface type *programInterface* of program object *program*.

If *name* exactly matches the name string of one of the active resources for *programInterface*, the index of the matched resource is returned.

- For TRANSFORM_FEEDBACK_VARYING resources, *name* must match one of the variables to be captured as specified by a previous call to **TransformFeedbackVaryings**, other than the special *name*s gl_NextBuffer, gl_SkipComponents1, gl_SkipComponents2, gl_SkipComponents3, and gl_SkipComponents4 (see section 11.1.2.1). Otherwise, INVALID_INDEX is returned.
- For all other resource types, if *name* would exactly match the name string of an active resource if "[0]" were appended to *name*, the index of the matched resource is returned. Otherwise, *name* is considered not to be the name of an active resource, and INVALID_INDEX is returned. Note that if an interface enumerates a single active resource list entry for an array variable (e.g., "a[0]"), a *name* identifying any array element other than the first (e.g., "a[1]") is not considered to match.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *programInterface* is not one of the interfaces described in the introduction to section 7.3.1.

An INVALID_ENUM error is generated if *programInterface* is ATOMIC_COUNTER_BUFFER or TRANSFORM_FEEDBACK_BUFFER, since active atomic counter and transform feedback buffer resources are not assigned name strings.

If *name* does not match a resource as described above, the value INVALID_INDEX is returned, but no GL error is generated.

The command

```
void GetProgramResourceName( uint program,
    enum programInterface, uint index, sizei bufSize,
    sizei *length, char *name );
```

returns the name string assigned to the single active resource with an index of *index* in the interface *programInterface* of program object *program*.

The name string assigned to the active resource identified by *index* is returned as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, no length is returned. The maximum number of characters that may be written into *name*, including the null terminator, is specified by *bufSize*. If the length of the name string (including the null terminator) is greater than *bufSize*, the first $bufSize - 1$ characters of the name string will be written to *name*, followed by a null terminator. If *bufSize* is zero, no error is generated but no characters will be written to *name*. The length of the longest name string for *programInterface*, including a null terminator, may be queried by calling **GetProgramInterfaceiv** with a *pname* of `MAX_NAME_LENGTH`.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_ENUM` error is generated if *programInterface* is not one of the interfaces described in the introduction to section 7.3.1.

An `INVALID_ENUM` error is generated if *programInterface* is `ATOMIC_COUNTER_BUFFER` or `TRANSFORM_FEEDBACK_BUFFER`, since active atomic counter and transform feedback buffer resources are not assigned name strings.

An `INVALID_VALUE` error is generated if *index* is greater than or equal to the number of entries in the active resource list for *programInterface*.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

The command

```
void GetProgramResourceiv( uint program,
    enum programInterface, uint index, sizei propCount,
    const enum *props, sizei count, sizei *length,
    int *params );
```

returns values for multiple properties of a single active resource with an index of *index* in the interface *programInterface* of program object *program*. Values for *propCount* properties specified by the array *props* are returned.

The values associated with the properties of the active resource are written to consecutive entries in *params*, in increasing order according to position in *props*. If

no error is generated, only the first *count* integer values will be written to *params*; any extra values will not be written. If *length* is not NULL, the actual number of values written to *params* will be written to *length*.

Property	Supported Interfaces
ACTIVE_VARIABLES, BUFFER_BINDING, NUM_ACTIVE_VARIABLES	ATOMIC_COUNTER_BUFFER, SHADER_STORAGE_BLOCK, TRANSFORM_FEEDBACK_BUFFER, UNIFORM_BLOCK
ARRAY_SIZE	BUFFER_VARIABLE, COMPUTE_SUBROUTINE_UNIFORM, FRAGMENT_SUBROUTINE_UNIFORM, GEOMETRY_SUBROUTINE_UNIFORM, PROGRAM_INPUT, PROGRAM_OUTPUT, TESS_CONTROL_SUBROUTINE_UNIFORM, TESS_EVALUATION_SUBROUTINE_UNIFORM, TRANSFORM_FEEDBACK_VARYING, UNIFORM, VERTEX_SUBROUTINE_UNIFORM
ARRAY_STRIDE, BLOCK_INDEX, IS_ROW_MAJOR, MATRIX_STRIDE	BUFFER_VARIABLE, UNIFORM
ATOMIC_COUNTER_BUFFER_INDEX	UNIFORM
BUFFER_DATA_SIZE	ATOMIC_COUNTER_BUFFER, SHADER_STORAGE_BLOCK, UNIFORM_BLOCK
NUM_COMPATIBLE_SUBROUTINES, COMPATIBLE_SUBROUTINES	COMPUTE_SUBROUTINE_UNIFORM, FRAGMENT_SUBROUTINE_UNIFORM, GEOMETRY_SUBROUTINE_UNIFORM, TESS_CONTROL_SUBROUTINE_UNIFORM, TESS_EVALUATION_SUBROUTINE_UNIFORM, VERTEX_SUBROUTINE_UNIFORM
IS_PER_PATCH	PROGRAM_INPUT, PROGRAM_OUTPUT
GetProgramResourceiv properties continued on next page	

GetProgramResourceiv properties continued from previous page	
Property	Supported Interfaces
LOCATION	COMPUTE_SUBROUTINE_UNIFORM, FRAGMENT_SUBROUTINE_UNIFORM, GEOMETRY_SUBROUTINE_UNIFORM, PROGRAM_INPUT, PROGRAM_OUTPUT, TESS_CONTROL_SUBROUTINE_UNIFORM, TESS_EVALUATION_SUBROUTINE_UNIFORM, UNIFORM, VERTEX_SUBROUTINE_UNIFORM
LOCATION_COMPONENT	PROGRAM_INPUT, PROGRAM_OUTPUT
LOCATION_INDEX	PROGRAM_OUTPUT
NAME_LENGTH	all but ATOMIC_COUNTER_BUFFER and TRANSFORM_FEEDBACK_BUFFER
OFFSET	BUFFER_VARIABLE, TRANSFORM_FEEDBACK_VARYING, UNIFORM
REFERENCED_BY_VERTEX_SHADER, REFERENCED_BY_TESS_CONTROL_SHADER, REFERENCED_BY_TESS_EVALUATION_SHADER, REFERENCED_BY_GEOMETRY_SHADER, REFERENCED_BY_FRAGMENT_SHADER, REFERENCED_BY_COMPUTE_SHADER	ATOMIC_COUNTER_BUFFER, BUFFER_VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT, SHADER_STORAGE_BLOCK, UNIFORM, UNIFORM_BLOCK
TRANSFORM_FEEDBACK_BUFFER_INDEX	TRANSFORM_FEEDBACK_VARYING
TRANSFORM_FEEDBACK_BUFFER_STRIDE	TRANSFORM_FEEDBACK_BUFFER
TOP_LEVEL_ARRAY_SIZE, TOP_LEVEL_ARRAY_STRIDE	BUFFER_VARIABLE
TYPE	BUFFER_VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT, TRANSFORM_FEEDBACK_VARYING, UNIFORM

Table 7.2: **GetProgramResourceiv** properties and supported interfaces

For the property ACTIVE_VARIABLES, an array of active variable indices associated with an atomic counter buffer, active uniform block, shader storage block,

or transform feedback buffer is written to *params*. The number of values written to *params* for an active resource is given by the value of the property NUM_ACTIVE_VARIABLES for the resource.

For the property ARRAY_SIZE, a single integer identifying the number of active array elements of an active variable is written to *params*. The array size returned is in units of the type associated with the property TYPE. For active variables not corresponding to an array of basic types, the value one is written to *params*. If the variable is an array whose size is not declared or determined when the program is linked, the value zero is written to *params*.

For the property ARRAY_STRIDE, a single integer identifying the stride between array elements in an active variable is written to *params*. For active variables declared as an array of basic types, the value written is the difference, in basic machine units, between the offsets of consecutive elements in an array. For active variables not declared as an array of basic types, zero is written to *params*. For active variables not backed by a buffer object, -1 is written to *params*, regardless of the variable type.

For the property ATOMIC_COUNTER_BUFFER_INDEX, a single integer identifying the index of the active atomic counter buffer containing an active variable is written to *params*. If the variable is not an atomic counter uniform, the value -1 is written to *params*.

For the property BLOCK_INDEX, a single integer identifying the index of the active interface block containing an active variable is written to *params*. The index written for a member of an interface block declared as an array of block instances is the index of the first block of the array. If the variable is not the member of an interface block, the value -1 is written to *params*.

For the property BUFFER_BINDING, a single integer identifying the index of the buffer binding point associated with the active uniform block, atomic counter buffer, shader storage block, or transform feedback buffer is written to *params*.

For the property BUFFER_DATA_SIZE, a single integer identifying the implementation-dependent minimum total buffer object size is written to *params*. This value is the size, in basic machine units, required to hold all active variables associated with an active uniform block, shader storage block, or atomic counter buffer. If the final member of an active shader storage block is an array with no declared size, the minimum buffer size is computed assuming the array was declared as an array with one element.

For the property IS_PER_PATCH, a single integer identifying whether the input or output is a per-patch attribute is written to *params*. If the active variable is a per-patch attribute (declared with the patch qualifier), the value one is written to *params*; otherwise, the value zero is written to *params*.

For the property IS_ROW_MAJOR, a single integer identifying whether an active

variable is a row-major matrix is written to *params*. For active variables backed by a buffer object, declared as a single matrix or array of matrices, and stored in row-major order, one is written to *params*. For all other active variables, zero is written to *params*.

For the property `LOCATION`, a single integer identifying the assigned location for an active uniform, input, output, or subroutine uniform variable is written to *params*. For input, output, or uniform variables with locations specified by a `layout` qualifier, the specified location is used. For vertex shader input, fragment shader output, or uniform variables without a `layout` qualifier, the location assigned when a program is linked is written to *params*. For all other input and output variables, the value -1 is written to *params*. For atomic counter uniforms and uniforms in uniform blocks, the value -1 is written to *params*.

For the property `LOCATION_COMPONENT`, a single integer indicating the first component of the location assigned to an active input or output variable is written to *params*. For input and output variables with a `component` specified by a *layout* qualifier, the specified component is written. For all other input and output variables, the value zero is written.

For the property `LOCATION_INDEX`, a single integer identifying the fragment color index of an active fragment shader output variable is written to *params*. If the active variable is not an output for a fragment shader, the value -1 will be written to *params*.

For the property `MATRIX_STRIDE`, a single integer identifying the stride between columns of a column-major matrix or rows of a row-major matrix is written to *params*. For active variables declared a single matrix or array of matrices, the value written is the difference, in basic machine units, between the offsets of consecutive columns or rows in each matrix. For active variables not declared as a matrix or array of matrices, zero is written to *params*. For active variables not backed by a buffer object, -1 is written to *params*, regardless of the variable type.

For the property `NAME_LENGTH`, a single integer identifying the length of the name string associated with an active variable, interface block, or subroutine is written to *params*. The name length includes a terminating null character.

For the property `NUM_ACTIVE_VARIABLES`, a single integer identifying the number of active variables associated with an active uniform block, atomic counter buffer, shader storage block, or transform feedback buffer is written to *params*.

For the property `OFFSET`, a single integer identifying the offset of an active variable is written to *params*. For variables in the `BUFFER_VARIABLE` and `UNIFORM` interfaces that are backed by a buffer object, the value written is the offset of that variable relative to the base of the buffer range holding its value. For variables in the `TRANSFORM_FEEDBACK_VARYING` interface, the value written is the offset in the transform feedback buffer storage assigned to each ver-

tex captured in transform feedback mode where the value of the variable will be stored. Such offsets are specified via the `xfb_offset layout` qualifier or assigned according to the variables position in the list of strings passed to **TransformFeedbackVaryings**. Offsets are expressed in basic machine units. For all variables not recorded in transform feedback mode, including the special names `gl_NextBuffer`, `gl_SkipComponents1`, `gl_SkipComponents2`, `gl_SkipComponents3`, and `gl_SkipComponents4`, -1 is written to *params*.

For the properties `REFERENCED_BY_VERTEX_SHADER`, `REFERENCED_BY_TESS_CONTROL_SHADER`, `REFERENCED_BY_TESS_EVALUATION_SHADER`, `REFERENCED_BY_GEOMETRY_SHADER`, `REFERENCED_BY_FRAGMENT_SHADER`, and `REFERENCED_BY_COMPUTE_SHADER`, a single integer is written to *params*, identifying whether the active resource is referenced by the vertex, tessellation control, tessellation evaluation, geometry, fragment, or compute shaders, respectively, in the program object. The value one is written to *params* if an active variable is referenced by the corresponding shader, or if an active uniform block, shader storage block, or atomic counter buffer contains at least one variable referenced by the corresponding shader. Otherwise, the value zero is written to *params*.

For the property `TOP_LEVEL_ARRAY_SIZE`, a single integer identifying the number of active array elements of the top-level shader storage block member containing the active variable is written to *params*. If the top-level block member is not declared as an array f an aggregate type, the value one is written to *params*. If the top-level block member is an array of an aggregate type whose size is not declared or determined when the program is linked, the value zero is written to *params*.

For the property `TOP_LEVEL_ARRAY_STRIDE`, a single integer identifying the stride between array elements of the top-level shader storage block member containing the active variable is written to *params*. For top-level block members declared as arrays of an aggregate type, the value written is the difference, in basic machine units, between the offsets of the active variable for consecutive elements in the top-level array. For top-level block members not declared as an array of an aggregate type, zero is written to *params*.

For the property `TRANSFORM_FEEDBACK_BUFFER_INDEX`, a single integer identifying the index of the active transform feedback buffer associated with an active variable is written to *params*. For variables corresponding to the special names `gl_NextBuffer`, `gl_SkipComponents1`, `gl_SkipComponents2`, `gl_SkipComponents3`, and `gl_SkipComponents4`, -1 is written to *params*.

For the property `TRANSFORM_FEEDBACK_BUFFER_STRIDE`, a single integer identifying the stride, in basic machine units, between consecutive vertices written to the transform feedback buffer is written to *params*.

For the property `TYPE`, a single integer identifying the type of an active variable is written to *params*. The integer returned is one of the values found in table 7.3.

Type Name Token	Keyword	Attrib	Xfb	Buffer
FLOAT	float	●	●	●
FLOAT_VEC2	vec2	●	●	●
FLOAT_VEC3	vec3	●	●	●
FLOAT_VEC4	vec4	●	●	●
DOUBLE	double	●	●	●
DOUBLE_VEC2	dvec2	●	●	●
DOUBLE_VEC3	dvec3	●	●	●
DOUBLE_VEC4	dvec4	●	●	●
INT	int	●	●	●
INT_VEC2	ivec2	●	●	●
INT_VEC3	ivec3	●	●	●
INT_VEC4	ivec4	●	●	●
UNSIGNED_INT	uint	●	●	●
UNSIGNED_INT_VEC2	uvec2	●	●	●
UNSIGNED_INT_VEC3	uvec3	●	●	●
UNSIGNED_INT_VEC4	uvec4	●	●	●
BOOL	bool			●
BOOL_VEC2	bvec2			●
BOOL_VEC3	bvec3			●
BOOL_VEC4	bvec4			●
FLOAT_MAT2	mat2	●	●	●
FLOAT_MAT3	mat3	●	●	●
FLOAT_MAT4	mat4	●	●	●
FLOAT_MAT2x3	mat2x3	●	●	●
FLOAT_MAT2x4	mat2x4	●	●	●
FLOAT_MAT3x2	mat3x2	●	●	●
FLOAT_MAT3x4	mat3x4	●	●	●
FLOAT_MAT4x2	mat4x2	●	●	●
FLOAT_MAT4x3	mat4x3	●	●	●
DOUBLE_MAT2	dmat2	●	●	●
DOUBLE_MAT3	dmat3	●	●	●
DOUBLE_MAT4	dmat4	●	●	●
DOUBLE_MAT2x3	dmat2x3	●	●	●
DOUBLE_MAT2x4	dmat2x4	●	●	●
DOUBLE_MAT3x2	dmat3x2	●	●	●

(Continued on next page)

OpenGL Shading Language Type Tokens (continued)				
Type Name Token	Keyword	Attrib	Xfb	Buffer
DOUBLE_MAT3x4	dmat3x4	•	•	•
DOUBLE_MAT4x2	dmat4x2	•	•	•
DOUBLE_MAT4x3	dmat4x3	•	•	•
SAMPLER_1D	sampler1D			
SAMPLER_2D	sampler2D			
SAMPLER_3D	sampler3D			
SAMPLER_CUBE	samplerCube			
SAMPLER_1D_SHADOW	sampler1DShadow			
SAMPLER_2D_SHADOW	sampler2DShadow			
SAMPLER_1D_ARRAY	sampler1DArray			
SAMPLER_2D_ARRAY	sampler2DArray			
SAMPLER_CUBE_MAP_ARRAY	samplerCubeArray			
SAMPLER_1D_ARRAY_SHADOW	sampler1DArrayShadow			
SAMPLER_2D_ARRAY_SHADOW	sampler2DArrayShadow			
SAMPLER_2D_MULTISAMPLE	sampler2DMS			
SAMPLER_2D_MULTISAMPLE_-ARRAY	sampler2DMSArray			
SAMPLER_CUBE_SHADOW	samplerCubeShadow			
SAMPLER_CUBE_MAP_ARRAY_-SHADOW	samplerCube-ArrayShadow			
SAMPLER_BUFFER	samplerBuffer			
SAMPLER_2D_RECT	sampler2DRect			
SAMPLER_2D_RECT_SHADOW	sampler2DRectShadow			
INT_SAMPLER_1D	isampler1D			
INT_SAMPLER_2D	isampler2D			
INT_SAMPLER_3D	isampler3D			
INT_SAMPLER_CUBE	isamplerCube			
INT_SAMPLER_1D_ARRAY	isampler1DArray			
INT_SAMPLER_2D_ARRAY	isampler2DArray			
INT_SAMPLER_CUBE_MAP_-ARRAY	isamplerCubeArray			
INT_SAMPLER_2D_-MULTISAMPLE	isampler2DMS			
INT_SAMPLER_2D_-MULTISAMPLE_ARRAY	isampler2DMSArray			

(Continued on next page)

OpenGL Shading Language Type Tokens (continued)				
Type Name Token	Keyword	Attrib	Xfb	Buffer
INT_SAMPLER_BUFFER	isamplerBuffer			
INT_SAMPLER_2D_RECT	isampler2DRect			
UNSIGNED_INT_SAMPLER_1D	usampler1D			
UNSIGNED_INT_SAMPLER_2D	usampler2D			
UNSIGNED_INT_SAMPLER_3D	usampler3D			
UNSIGNED_INT_SAMPLER_-CUBE	usamplerCube			
UNSIGNED_INT_SAMPLER_-1D_ARRAY	usampler1DArray			
UNSIGNED_INT_SAMPLER_-2D_ARRAY	usampler2DArray			
UNSIGNED_INT_SAMPLER_-CUBE_MAP_ARRAY	usamplerCubeArray			
UNSIGNED_INT_SAMPLER_-2D_MULTISAMPLE	usampler2DMS			
UNSIGNED_INT_SAMPLER_-2D_MULTISAMPLE_ARRAY	usampler2DMSArray			
UNSIGNED_INT_SAMPLER_-BUFFER	usamplerBuffer			
UNSIGNED_INT_SAMPLER_-2D_RECT	usampler2DRect			
IMAGE_1D	image1D			
IMAGE_2D	image2D			
IMAGE_3D	image3D			
IMAGE_2D_RECT	image2DRect			
IMAGE_CUBE	imageCube			
IMAGE_BUFFER	imageBuffer			
IMAGE_1D_ARRAY	image1DArray			
IMAGE_2D_ARRAY	image2DArray			
IMAGE_CUBE_MAP_ARRAY	imageCubeArray			
IMAGE_2D_MULTISAMPLE	image2DMS			
IMAGE_2D_MULTISAMPLE_-ARRAY	image2DMSArray			
INT_IMAGE_1D	iimage1D			
INT_IMAGE_2D	iimage2D			

(Continued on next page)

OpenGL Shading Language Type Tokens (continued)				
Type Name Token	Keyword	Attrib	Xfb	Buffer
INT_IMAGE_3D	iimage3D			
INT_IMAGE_2D_RECT	iimage2DRect			
INT_IMAGE_CUBE	iimageCube			
INT_IMAGE_BUFFER	iimageBuffer			
INT_IMAGE_1D_ARRAY	iimage1DArray			
INT_IMAGE_2D_ARRAY	iimage2DArray			
INT_IMAGE_CUBE_MAP_ARRAY	iimageCubeArray			
INT_IMAGE_2D_MULTISAMPLE	iimage2DMS			
INT_IMAGE_2D_-MULTISAMPLE_ARRAY	iimage2DMSArray			
UNSIGNED_INT_IMAGE_1D	uimage1D			
UNSIGNED_INT_IMAGE_2D	uimage2D			
UNSIGNED_INT_IMAGE_3D	uimage3D			
UNSIGNED_INT_IMAGE_2D_-RECT	uimage2DRect			
UNSIGNED_INT_IMAGE_CUBE	uimageCube			
UNSIGNED_INT_IMAGE_-BUFFER	uimageBuffer			
UNSIGNED_INT_IMAGE_1D_-ARRAY	uimage1DArray			
UNSIGNED_INT_IMAGE_2D_-ARRAY	uimage2DArray			
UNSIGNED_INT_IMAGE_-CUBE_MAP_ARRAY	uimageCubeArray			
UNSIGNED_INT_IMAGE_2D_-MULTISAMPLE	uimage2DMS			
UNSIGNED_INT_IMAGE_2D_-MULTISAMPLE_ARRAY	uimage2DMSArray			
UNSIGNED_INT_ATOMIC_-COUNTER	atomic_uint			

Table 7.3: OpenGL Shading Language type tokens, and corresponding shading language keywords declaring each such type. Types whose "Attrib" column is marked may be declared as vertex attributes (see section 11.1.1). Types whose "Xfb" column is marked may be the types of variables returned by transform feedback (see section 11.1.2.1). Types whose "Buffer" column is marked may be declared as buffer variables (see section 7.8).

For the property COMPATIBLE_SUBROUTINES, an array of integers is written to *params*, with each integer specifying the index of an active subroutine that can be assigned to the selected subroutine uniform. The number of values written to *params* for an active subroutine is given by the value of the property NUM_COMPATIBLE_SUBROUTINES for the resource.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *programInterface* is not one of the interfaces described in the introduction to section 7.3.1.

An INVALID_VALUE error is generated if *propCount* is less than or equal to zero, or if *count* is negative.

An INVALID_ENUM error is generated if any value in *props* is not one of the properties described above.

An INVALID_OPERATION error is generated if any value in *props* is not allowed for *programInterface*. The set of allowed *programInterface* values for each property can be found in table 7.2.

The commands

```
int GetProgramResourceLocation( uint program,
    enum programInterface, const char *name );
int GetProgramResourceLocationIndex( uint program,
    enum programInterface, const char *name );
```

return the location or the fragment color index, respectively, assigned to the variable named *name* in interface *programInterface* of program object *program*. For **GetProgramResourceLocation**, *programInterface* must be one of UNIFORM, PROGRAM_INPUT, PROGRAM_OUTPUT, VERTEX_SUBROUTINE_UNIFORM, TESS_CONTROL_SUBROUTINE_UNIFORM, TESS_EVALUATION_SUBROUTINE_UNIFORM, GEOMETRY_SUBROUTINE_UNIFORM, FRAGMENT_SUBROUTINE_UNIFORM, or COMPUTE_SUBROUTINE_UNIFORM. For **GetProgramResourceLocationIndex**, *programInterface* must be PROGRAM_OUTPUT. The value -1 will be returned by either command if an error occurs, if *name* does not identify an active variable on *programInterface*, or if *name* identifies an active variable that does

not have a valid location assigned, as described above. The locations returned by these commands are the same locations returned when querying the LOCATION and LOCATION_INDEX resource properties.

A string provided to **GetProgramResourceLocation** or **GetProgramResourceLocationIndex** is considered to match an active variable if

- the string exactly matches the name of the active variable;
- if the string identifies the base name of an active array, where the string would exactly match the name of the variable if the suffix "[0]" were appended to the string; or
- if the string identifies an active element of the array, where the string ends with the concatenation of the "[" character, an integer (with no "+" sign, extra leading zeroes, or whitespace) identifying an array element, and the "]" character, the integer is less than the number of active elements of the array variable, and where the string would exactly match the enumerated name of the array if the decimal integer were replaced with zero.

Any other string is considered not to identify an active variable. If the string specifies an element of an array variable, **GetProgramResourceLocation** and **GetProgramResourceLocationIndex** return the location or fragment color index assigned to that element. If it specifies the base name of an array, it identifies the resources associated with the first element of the array.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_OPERATION error is generated if *program* has not been linked successfully.

An INVALID_ENUM error is generated if *programInterface* is not one of the interfaces named above.

7.4 Program Pipeline Objects

Instead of packaging all shader stages into a single program object, shader types might be contained in multiple program objects each consisting of part of the complete pipeline. A program object may even contain only a single shader stage.

This facilitates greater flexibility when combining different shaders in various ways without requiring a program object for each combination.

A program pipeline object contains bindings for each shader type associating that shader type with a program object.

The command

`void` **GenProgramPipelines**(`sizei` *n*, `uint` **pipelines*);

returns *n* previously unused program pipeline object names in *pipelines*. These names are marked as used, for the purposes of **GenProgramPipelines** only, but they acquire state only when they are first bound.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

Program pipeline objects are deleted by calling

`void` **DeleteProgramPipelines**(`sizei` *n*, `const uint` **pipelines*);

pipelines contains *n* names of program pipeline objects to be deleted. Once a program pipeline object is deleted, it has no contents and its name becomes unused. If an object that is currently bound is deleted, the binding for that object reverts to zero and no program pipeline object becomes current. Unused names in *pipelines* that have been marked as used for the purposes of **GenProgramPipelines** are marked as unused again. Unused names in *pipelines* are silently ignored, as is the value zero.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

The command

`boolean` **IsProgramPipeline**(`uint` *pipeline*);

returns `TRUE` if *pipeline* is the name of a program pipeline object. If *pipeline* is zero, or a non-zero value that is not the name of a program pipeline object, **IsProgramPipeline** returns `FALSE`. No error is generated if *pipeline* is not a valid program pipeline object name.

A program pipeline object is created by binding a name returned by **GenProgramPipelines** with the command

`void` **BindProgramPipeline**(`uint` *pipeline*);

pipeline is the program pipeline object name. The resulting program pipeline object is a new state vector, comprising all the state and with the same initial values listed in table 23.42.

BindProgramPipeline may also be used to bind an existing program pipeline object. If the bind is successful, no change is made to the state of the bound program pipeline object, and any previous binding is broken. If **BindProgramPipeline** is called with *pipeline* set to zero, then there is no current program pipeline object.

If no current program object has been established by **UseProgram**, the program objects used for each shader stage and for uniform updates are taken from the bound program pipeline object, if any. If there is a current program object established by **UseProgram**, the bound program pipeline object has no effect on rendering or uniform updates. When a bound program pipeline object is used for rendering, individual shader executables are taken from its program objects as described in the discussion of **UseProgram** in section 7.3).

Errors

An `INVALID_OPERATION` error is generated if *pipeline* is not zero or a name returned from a previous call to **GenProgramPipelines**, or if such a name has since been deleted with **DeleteProgramPipelines**.

Program pipeline objects may also be created with the command

`void` **CreateProgramPipelines**(`sizei` *n*, `uint` **pipelines*);

CreateProgramPipelines returns *n* previously unused program pipeline names in *pipelines*, each representing a new program pipeline object which is a state vector comprising all the state and with the same initial values listed in table 23.42.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

The executables in a program object associated with one or more shader stages can be made part of the program pipeline state for those shader stages with the command

`void` **UseProgramStages**(`uint` *pipeline*, `bitfield` *stages*,
`uint` *program*);

where *pipeline* is the program pipeline object to be updated, *stages* is the bitwise OR of accepted constants representing shader stages, and *program* identifies the program from which the executables are taken.

The bits set in *stages* indicate the program stages for which the program object named by *program* becomes current. These stages may include compute, vertex, tessellation control, tessellation evaluation, geometry, or fragment, indicated respectively by `COMPUTE_SHADER_BIT`, `VERTEX_SHADER_BIT`, `TESS_CONTROL_SHADER_BIT`, `TESS_EVALUATION_SHADER_BIT`, `GEOMETRY_SHADER_BIT`, or `FRAGMENT_SHADER_BIT`. The constant `ALL_SHADER_BITS` indicates *program* is to be made current for all shader stages.

If *program* refers to a program object with a valid shader attached for an indicated shader stage, this call installs the executable code for that stage in the indicated program pipeline object state. If **UseProgramStages** is called with *program* set to zero or with a program object that contains no executable code for any stage in *stages*, it is as if the pipeline object has no programmable stage configured for that stage.

If *pipeline* is a name that has been generated (without subsequent deletion) by **GenProgramPipelines**, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when **BindProgramPipeline** creates a new program pipeline object.

Errors

An `INVALID_VALUE` error is generated if *stages* is not the special value `ALL_SHADER_BITS`, and has any bits set other than `VERTEX_SHADER_BIT`, `COMPUTE_SHADER_BIT`, `TESS_CONTROL_SHADER_BIT`, `TESS_EVALUATION_SHADER_BIT`, `GEOMETRY_SHADER_BIT`, and `FRAGMENT_SHADER_BIT`.

An `INVALID_VALUE` error is generated if *program* is not zero and is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_OPERATION` error is generated if *program* is not zero and was linked without the `PROGRAM_SEPARABLE` parameter set, or has not been linked successfully. The corresponding shader stages in *pipeline* are not modified.

An `INVALID_OPERATION` error is generated if *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**.

The command

`void` **ActiveShaderProgram**(`uint` *pipeline*, `uint` *program*);

sets the linked program named by *program* to be the active program (see section 7.6.1) used for uniform updates for the program pipeline object *pipeline*. If *program* is zero, then it is as if there is no active program for *pipeline*.

If *pipeline* is a name that has been generated (without subsequent deletion) by **GenProgramPipelines**, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when **BindProgramPipeline** creates a new program pipeline object.

Errors

An `INVALID_OPERATION` error is generated if *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**.

An `INVALID_VALUE` error is generated if *program* is not zero and is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_OPERATION` error is generated if *program* is not zero and has not been linked successfully. The active program is not modified.

7.4.1 Shader Interface Matching

When multiple shader stages are active, the outputs of one stage form an interface with the inputs of the next stage. At each such interface, shader inputs are matched up against outputs from the previous stage:

- An output block is considered to match an input block in the subsequent shader if the two blocks have the same block name, and the members of the block match exactly in name, type, qualification, and declaration order.
  - For the purposes of shader interface matching, the `gl_PointSize` member of the intrinsically declared `gl_PerVertex` shader interface block is ignored.
  - Output blocks that do not match in name, but have a location and match in every other way listed above may be considered to match by some implementations, but not all - so this behaviour should not be relied upon.

- An output variable is considered to match an input variable in the subsequent shader if:
  - the two variables match in name, type, and qualification, and neither has a `location` qualifier, or
  - the two variables are declared with the same `location` and `component layout` qualifiers and match in type and qualification.

For the purposes of interface matching, variables declared with a `location layout` qualifier but without a `component layout` qualifier are considered to have declared a `component layout` qualifier of zero. Variables or block members declared as structures are considered to match in type if and only if structure members match in name, type, qualification, and declaration order. Variables or block members declared as arrays are considered to match in type only if both declarations specify the same element type and array size. The rules for determining if variables or block members match in qualification are found in the OpenGL Shading Language Specification.

Tessellation control shader per-vertex output variables and blocks and tessellation control, tessellation evaluation, and geometry shader per-vertex input variables and blocks are required to be declared as arrays, with each element representing input or output values for a single vertex of a multi-vertex primitive. For the purposes of interface matching, such variables and blocks are treated as though they were not declared as arrays.

For program objects containing multiple shaders, **LinkProgram** will check for mismatches on interfaces between shader stages in the program being linked and generate a link error if a mismatch is detected. A link error is generated if any statically referenced input variable or block does not have a matching output. If either shader redeclares the built-in arrays `gl_ClipDistance[]`, `gl_CullDistance[]` or `gl_TexCoord[]`, the array must have the same size in both shaders. If either shader redeclares built-in input or output color variables, they must have matching interpolation qualifiers, as described in the OpenGL Shading Language Specification.

With separable program objects, interfaces between shader stages may involve the outputs from one program object and the inputs from a second program object. For such interfaces, it is not possible to detect mismatches at link time, because the programs are linked separately. When each such program is linked, all inputs or outputs interfacing with another program stage are treated as active. The linker will generate an executable that assumes the presence of a compatible program on the other side of the interface. If a mismatch between programs occurs, no GL error is generated, but some or all of the inputs on the interface will be undefined.

At an interface between program objects, the set of inputs and outputs are considered to match exactly if and only if:

- The built-in input and output blocks used on the interface (`gl_PerVertex` or `gl_PerFragment`) match, as described below.
- Every declared input block or variable has a matching output, as described above. However, the intrinsically declared `gl_PerVertex` shader interface block must be redeclared, and all members of the redeclared `gl_PerVertex` shader interface block, including the `gl_PointSize` member if present in the redeclaration, must match exactly in name, type, qualification and declaration order.
- There are no output blocks or user-defined output variables declared without a matching input block or variable declaration.

When the set of inputs and outputs on an interface between programs matches exactly, all inputs are well-defined except when the corresponding outputs were not written in the previous shader. However, any mismatch between inputs and outputs results in all inputs being undefined except for cases noted below. Even if an input has a corresponding output that matches exactly, mismatches on other inputs or outputs may adversely affect the executable code generated to read or write the matching variable.

The inputs and outputs on an interface between programs need not match exactly when input and output location qualifiers (sections 4.4.1(“Input Layout Qualifiers”) and 4.4.2(“Output Layout Qualifiers”) of the OpenGL Shading Language Specification) are used. When using location qualifiers, any input with an input location qualifier will be well-defined as long as the other program writes to a matching output, as described above. The names of variables need not match when matching by location.

Additionally, scalar and vector inputs with `location layout` qualifiers will be well-defined if there is a corresponding output satisfying all of the following conditions:

- the input and output match exactly in qualification, including in the `location layout` qualifier;
- the output is a vector with the same basic component type and has more components than the input; and
- the common component type of the input and output is `int`, `uint`, or `float` (scalars, vectors, and matrices with `double` component type are excluded).

In this case, the components of the input will be taken from the first components of the matching output, and the extra components of the output will be ignored.

To use any built-in input or output in the `gl_PerVertex` and `gl_PerFragment` blocks in separable program objects, shader code must redeclare those blocks prior to use. A separable program will fail to link if:

- it contains multiple shaders of a single type with different redeclarations of these built-in input and output blocks; or
- any shader uses a built-in block member not found in the redeclaration of that block.
- if both of the shaders are OpenGL ES shading language shaders, all matched input and output variables (in a block or otherwise) have identical precision qualification.
- if one of the shaders is an OpenGL ES shading language shader and the other an OpenGL shading language shader, all matched input and output variables in the OpenGL ES shading language shader (in a block or otherwise) have highp precision qualification.

There is one exception to this rule described below.

As described above, an exact interface match requires matching built-in input and output blocks. At an interface between two non-fragment shader stages, the `gl_PerVertex` input and output blocks are considered to match if and only if the block members match exactly in name, type, qualification, and declaration order. At an interface involving the fragment shader stage, a `gl_PerVertex` output block is considered to match a `gl_PerFragment` input block if all of the following conditions apply:

- the `gl_PerVertex` block includes either `gl_FrontColor` or `gl_BackColor` if and only if the `gl_PerFragment` block includes `gl_Color`;
- the `gl_PerVertex` block includes either `gl_FrontSecondaryColor` or `gl_BackSecondaryColor` if and only if the `gl_PerFragment` block includes `gl_SecondaryColor`;
- the `gl_PerVertex` block includes `gl_FogFragCoord` if and only if the `gl_PerFragment` block also includes `gl_FogFragCoord`; and
- the size of `gl_TexCoord[]` in `gl_PerVertex` and `gl_PerFragment` is identical.

At an interface between `gl_PerVertex` outputs and `gl_PerFragment` inputs, the presence or absence of any block members other than those listed immediately above does not affect interface matching.

Built-in inputs or outputs not found in blocks do not affect interface matching. Any such built-in inputs are well-defined unless they are derived from built-in outputs not written by the previous shader stage.

7.4.2 SPIR-V Shader Interface Matching

SPIR-V shaders must also follow the rules in this section, whether they add to or override those given in section 7.4.1. Most importantly, SPIR-V variables and structure members do not have names and so no interface matching is done by name strings.

All variables forming the input or output interfaces of shader stages must be listed as operands to the `OpEntryPoint` instruction and are declared with the `Input` or `Output` Storage Classes, respectively, in the SPIR-V module.

Shader built-in variables meeting the following requirements define the built-in interface block. They must:

- be explicitly declared (there are no implicit built-ins),
- be decorated with the `BuiltIn` decoration,
- be declared in a block whose top-level members are the built-ins, and
- not have any `Location` or `Component` decorations.

Built-ins only participate in interface matching if they are declared in such a block. There must be no more than one built-in interface block per shader per interface.

User-defined interface variables must be decorated with a `Location` and can also be decorated with a `Component`. These correspond to the location and component discussed in section 7.4.1. Uniform and shader storage block variables must also be decorated with a `Binding`.

A user-defined output variable is considered to match an input variable in the subsequent stage only if the two variables are declared with the same `Location` and `Component` decoration and match in type and decoration, except that interpolation decorations are not required to match.

Variables or block members declared as structures are considered to match in type if and only if the structure members match in type, decoration, number, and declaration order. Variables or block members declared as arrays are considered to match in type only if both declarations specify the same element type and size.

At an interface between two non-fragment shader stages, the built-in interface block must match exactly, as described above. At an interface involving the fragment shader inputs, the presence or absence of any built-in output does not affect the interface matching.

At an interface between two shader stages, the user-defined variable interface must match exactly. Additionally, scalar and vector inputs are well-defined if there is a corresponding output satisfying all of the following conditions:

- the input and output match exactly in decoration,
- the output is a vector with the same basic type and has at least as many components as the input, and
- the common component type of the input and output is 32-bit integer or floating-point (64-bit component types are excluded).

In this case, the components of the input will be taken from the first components of the output, and any extra components of the output will be ignored.

7.4.3 Program Pipeline Object State

The state required to support program pipeline objects consists of a single binding name of the current program pipeline object. This binding is initially zero indicating no program pipeline object is bound.

The state of each program pipeline object consists of:

- Unsigned integers holding the names of the active program and each of the current vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute stage programs. Each integer is initially zero.
- A boolean holding the status of the last validation attempt, initially false.
- An array of type `char` containing the information log (see section 7.14), initially empty.
- An integer holding the length of the information log.

7.5 Program Binaries

The command

`void` **GetProgramBinary**(`uint` *program*, `sizei` *bufSize*, `sizei` **length*, `enum` **binaryFormat*, `void` **binary*);

returns a binary representation of the program object's compiled and linked executable source, henceforth referred to as its program binary. The maximum number of bytes that may be written into *binary* is specified by *bufSize*. The actual number of bytes written into *binary* is returned in *length* and its format is returned in *binaryFormat*. If *length* is NULL, then no length is returned.

The number of bytes in the program binary may be queried by calling **GetProgramiv** with *pname* PROGRAM_BINARY_LENGTH.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_OPERATION error is generated if the value of NUM_PROGRAM_BINARY_FORMATS is zero.

An INVALID_OPERATION error is generated if *program* has not been linked successfully. In this case its program binary length is zero.

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated if *bufSize* is less than the number of bytes in the program binary.

The command

```
void ProgramBinary( uint program, enum binaryFormat,
   const void *binary, sizei length );
```

loads a program object with a program binary previously returned from **GetProgramBinary**. This is useful to avoid online compilation, while still using OpenGL Shading Language source shaders as a portable initial format. *binaryFormat* and *binary* must be those returned by a previous call to **GetProgramBinary**, and *length* must be the length of the program binary as returned by **GetProgramBinary** or **GetProgramiv** with *pname* PROGRAM_BINARY_LENGTH. Loading the program binary will fail, setting the LINK_STATUS of *program* to FALSE, if these conditions are not met.

Loading a program binary may also fail if the implementation determines that there has been a change in hardware or software configuration from when the program binary was produced such as having been compiled with an incompatible or outdated version of the compiler. In this case the application should fall back to providing the original OpenGL Shading Language source shaders, and perhaps again retrieve the program binary for future use.

A program object's program binary is replaced by calls to **LinkProgram** or **ProgramBinary**. Where linking success or failure is concerned, **ProgramBinary** can be considered to perform an implicit linking operation. **LinkProgram** and **ProgramBinary** both set the program object's LINK_STATUS to TRUE or FALSE, as queried with **GetProgramiv**, to reflect success or failure and update the information log, queried with **GetProgramInfoLog**, to provide details about warnings or errors.

A successful call to **ProgramBinary** will reset all uniform variables in the default uniform block, all uniform block buffer bindings, and all shader storage block buffer bindings to their initial values. The initial value is either the value of the variable's initializer as specified in the original shader source, or zero if no initializer was present.

Additionally, values of the following state that were in effect when the program was linked before saving are restored when **ProgramBinary** is called successfully:

- Program parameter PROGRAM_SEPARABLE,
- all vertex shader input and fragment shader output assignments, and
- atomic counter binding, offset and stride assignments.

If **ProgramBinary** fails to load a binary, no error is generated, but any information about a previous link or load of that program object is lost. Thus, a failed load does not restore the old state of *program*. The failure does not alter other program state not affected by linking such as the attached shaders, and the vertex attribute and fragment data location bindings as set by **BindAttribLocation** and **BindFragDataLocation**.

OpenGL defines no specific binary formats. Queries of values NUM_PROGRAM_BINARY_FORMATS and PROGRAM_BINARY_FORMATS return the number of program binary formats and the list of program binary format values supported by an implementation. The *binaryFormat* returned by **GetProgramBinary** must be present in this list.

Any program binary retrieved using **GetProgramBinary** and submitted using **ProgramBinary** under the same configuration must be successful. Any programs loaded successfully by **ProgramBinary** must be run properly with any legal GL state vector.

If an implementation needs to recompile or otherwise modify program executables based on GL state outside the program, **GetProgramBinary** is required to save enough information to allow such recompilation.

To indicate that a program binary is likely to be retrieved, **ProgramParameteri** should be called with *pname* set to PROGRAM_BINARY_RETRIEVABLE_HINT and

value set to TRUE. This setting will not be in effect until the next time **LinkProgram** or **ProgramBinary** has been called successfully. Additionally, the application may defer **GetProgramBinary** calls until after using the program with all non-program state vectors that it is likely to encounter. Such deferral may allow implementations to save additional information in the program binary that would minimize recompilation in future uses of the program binary.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *binaryFormat* is not a binary format present in the list of specific binary formats supported.

An INVALID_VALUE error is generated if *length* is negative.

7.6 Uniform Variables

Shaders can declare named *uniform variables*, as described in the OpenGL Shading Language Specification. A uniform is considered an *active uniform* if the compiler and linker determine that the uniform will actually be accessed when the executable code is executed. In cases where the compiler and linker cannot make a conclusive determination, the uniform will be considered active.

Sets of uniforms, except for atomic counters, images, samplers, and subroutine uniforms, can be grouped into *uniform blocks*.

Named uniform blocks, as described in the OpenGL Shading Language Specification, store uniform values in the data store of a buffer object corresponding to the uniform block. Such blocks are assigned a *uniform block index*.

Uniforms that are declared outside of a named uniform block are part of the *default uniform block*. The default uniform block has no name or uniform block index. Uniforms in the default uniform block, except for subroutine uniforms, are program object-specific state. They retain their values once loaded, and their values are restored whenever a program object is used, as long as the program object has not been re-linked.

Like uniforms, uniform blocks can be active or inactive. Active uniform blocks are those that contain active uniforms after a program has been compiled and linked. Uniform blocks declared in an array are considered active if any member of the array would otherwise be considered active.

Shader Stage	*pname* for querying default uniform block storage, in components
Vertex (see section 11.1.2)	MAX_VERTEX_UNIFORM_COMPONENTS
Tessellation control (see section 11.2.1.1)	MAX_TESS_CONTROL_UNIFORM_COMPONENTS
Tessellation evaluation (see section 11.2.3.1)	MAX_TESS_EVALUATION_UNIFORM_COMPONENTS
Geometry (see section 11.3.3)	MAX_GEOMETRY_UNIFORM_COMPONENTS
Fragment (see section 15.1)	MAX_FRAGMENT_UNIFORM_COMPONENTS
Compute (see section 19.1)	MAX_COMPUTE_UNIFORM_COMPONENTS

Table 7.4: Query targets for default uniform block storage, in components.

Shader Stage	*pname* for querying combined uniform block storage, in components
Vertex	MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS
Tessellation control	MAX_COMBINED_TESS_CONTROL_UNIFORM_COMPONENTS
Tessellation evaluation	MAX_COMBINED_TESS_EVALUATION_UNIFORM_COMPONENTS
Geometry	MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS
Fragment	MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS
Compute	MAX_COMBINED_COMPUTE_UNIFORM_COMPONENTS

Table 7.5: Query targets for combined uniform block storage, in components.

All members of a named uniform block declared with a `shared` or `std140` layout qualifier are considered active, even if they are not referenced in any shader in the program. The uniform block itself is also considered active, even if no member of the block is referenced.

The implementation-dependent amount of storage available for uniform variables, except for subroutine uniforms and atomic counters, in the default uniform block accessed by a shader for a particular shader stage may be queried by calling **GetIntegerv** with *pname* as specified in table 7.4 for that stage.

The implementation-dependent constants `MAX_VERTEX_UNIFORM_VECTORS` and `MAX_FRAGMENT_UNIFORM_VECTORS` have values respectively equal to the values of `MAX_VERTEX_UNIFORM_COMPONENTS` and `MAX_FRAGMENT_UNIFORM_COMPONENTS` divided by four.

The total amount of combined storage available for uniform variables in all uniform blocks accessed by a shader for a particular shader stage can be queried by calling **GetIntegerv** with *pname* as specified in table 7.5 for that stage.

These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a shader. For uniforms with boolean, integer, or floating-point components,

- A scalar uniform will consume no more than 1 component
- A vector uniform will consume no more than n components, where n is the vector component count
- A matrix uniform will consume no more than $4 \times min(r, c)$ components, where r and c are the number of rows and columns in the matrix.

Scalar, vector, and matrix uniforms with double-precision components will consume no more than twice the number of components of equivalent uniforms with floating-point components.

Errors

A link error is generated if an attempt is made to utilize more than the space available for uniform variables in a shader stage.

When a program is linked successfully, all active uniforms, except for atomic counters, belonging to the program object's default uniform block are initialized as defined by the version of the OpenGL Shading Language used to compile the program. A successful link will also generate a location for each active uniform in the default uniform block which doesn't already have an explicit location defined in the shader. The generated locations will never take the location of a uniform with an explicit location defined in the shader, even if that uniform is determined to be inactive. The values of active uniforms in the default uniform block can be changed using this location and the appropriate **Uniform*** or **ProgramUniform*** command (see section 7.6.1). These generated locations are invalidated and new ones assigned after each successful re-link. The explicitly defined locations and the generated locations must be in the range of zero to the value of `MAX_UNIFORM_LOCATIONS` minus one.

Similarly, when a program is linked successfully, all active atomic counters are assigned bindings, offsets (and strides for arrays of atomic counters) according to layout rules described in section 7.6.2.2. Atomic counter uniform buffer objects provide the storage for atomic counters, so the values of atomic counters may be changed by modifying the contents of the buffer object using the commands in sections 6.2, 6.2.1, 6.3, 6.5, and 6.6. Atomic counters are not assigned a location and may not be modified using the **Uniform*** commands. The bindings, offsets,

and strides belonging to atomic counters of a program object are invalidated and new ones assigned after each successful re-link.

Similarly, when a program is linked successfully, all active uniforms belonging to the program's named uniform blocks are assigned offsets (and strides for array and matrix type uniforms) within the uniform block according to layout rules described below. Uniform buffer objects provide the storage for named uniform blocks, so the values of active uniforms in named uniform blocks may be changed by modifying the contents of the buffer object. Uniforms in a named uniform block are not assigned a location and may not be modified using the **Uniform*** commands. The offsets and strides of all active uniforms belonging to named uniform blocks of a program object are invalidated and new ones assigned after each successful re-link.

To determine the set of active uniform variables used by a program, applications can query the properties and active resources of the `UNIFORM` interface of a program.

Additionally, several dedicated commands are provided to query properties of active uniforms. The command

```
int GetUniformLocation( uint program, const
    char *name );
```

is equivalent to

```
GetProgramResourceLocation(program, UNIFORM, name);
```

The command

```
void GetActiveUniformName( uint program,
    uint uniformIndex, sizei bufSize, sizei *length,
    char *uniformName );
```

is equivalent to

```
GetProgramResourceName(program, UNIFORM, uniformIndex,
    bufSize, length, uniformName);
```

The command

```
void GetUniformIndices( uint program,
    sizei uniformCount, const char *
    const *uniformNames, uint *uniformIndices );
```

is equivalent (assuming no errors are generated) to:

```
for (int i = 0; i < uniformCount; i++) {
    uniformIndices[i] = GetProgramResourceIndex(program,
        UNIFORM, uniformNames[i]);
}
```

The command

```
void GetActiveUniform( uint program, uint index,
    sizei bufSize, sizei *length, int *size, enum *type,
    char *name );
```

is equivalent (assuming no errors are generated) to:

```
const enum props[] = { ARRAY_SIZE, TYPE };
GetProgramResourceName(program, UNIFORM, index,
    bufSize, length, name);
GetProgramResourceiv(program, UNIFORM, index,
    1, &props[0], 1, NULL, size);
GetProgramResourceiv(program, UNIFORM, index,
    1, &props[1], 1, NULL, (int *)type);
```

The command

```
void GetActiveUniformsiv( uint program,
    sizei uniformCount, const uint *uniformIndices,
    enum pname, int *params );
```

is equivalent (assuming no errors are generated) to:

```
GLenum prop;
for (int i = 0; i < uniformCount; i++) {
    GetProgramResourceiv(program, UNIFORM, uniformIndices[i],
        1, &prop, 1, NULL, &params[i]);
}
```

where the value of *prop* is taken from table 7.6, based on the value of *pname*.

To determine the set of active uniform blocks used by a program, applications can query the properties and active resources of the UNIFORM_BLOCK interface.

Additionally, several commands are provided to query properties of active uniform blocks. The command

pname	*prop*
UNIFORM_TYPE	TYPE
UNIFORM_SIZE	ARRAY_SIZE
UNIFORM_NAME_LENGTH	NAME_LENGTH
UNIFORM_BLOCK_INDEX	BLOCK_INDEX
UNIFORM_OFFSET	OFFSET
UNIFORM_ARRAY_STRIDE	ARRAY_STRIDE
UNIFORM_MATRIX_STRIDE	MATRIX_STRIDE
UNIFORM_IS_ROW_MAJOR	IS_ROW_MAJOR
UNIFORM_ATOMIC_COUNTER_BUFFER_INDEX	ATOMIC_COUNTER_BUFFER_INDEX

Table 7.6: **GetProgramResourceiv** properties used by **GetActiveUniformsiv**.

```
uint GetUniformBlockIndex( uint program, const
    char *uniformBlockName );
```

is equivalent to

```
GetProgramResourceIndex(program, UNIFORM_BLOCK, uniformBlockName);
```

The command

```
void GetActiveUniformBlockName( uint program,
    uint uniformBlockIndex, sizei bufSize, sizei length,
    char *uniformBlockName );
```

is equivalent to

```
GetProgramResourceName(program, UNIFORM_BLOCK,
    uniformBlockIndex, bufSize, length, uniformBlockName);
```

The command

```
void GetActiveUniformBlockiv( uint program,
    uint uniformBlockIndex, enum pname, int *params );
```

is equivalent to

```
GLenum prop;
GetProgramResourceiv(program, UNIFORM_BLOCK,
    uniformBlockIndex, 1, &prop, maxSize, NULL, params);
```

pname	*prop*
UNIFORM_BLOCK_BINDING	BUFFER_BINDING
UNIFORM_BLOCK_DATA_SIZE	BUFFER_DATA_SIZE
UNIFORM_BLOCK_NAME_LENGTH	NAME_LENGTH
UNIFORM_BLOCK_ACTIVE_UNIFORMS	NUM_ACTIVE_VARIABLES
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES	ACTIVE_VARIABLES
UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER	REFERENCED_BY_VERTEX_SHADER
UNIFORM_BLOCK_REFERENCED_BY_TESS_CONTROL_SHADER	REFERENCED_BY_TESS_CONTROL_SHADER
UNIFORM_BLOCK_REFERENCED_BY_TESS_EVALUATION_SHADER	REFERENCED_BY_TESS_EVALUATION_SHADER
UNIFORM_BLOCK_REFERENCED_BY_GEOMETRY_SHADER	REFERENCED_BY_GEOMETRY_SHADER
UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER	REFERENCED_BY_FRAGMENT_SHADER
UNIFORM_BLOCK_REFERENCED_BY_COMPUTE_SHADER	REFERENCED_BY_COMPUTE_SHADER

Table 7.7: **GetProgramResourceiv** properties used by **GetActiveUniformBlockiv**.

where the value of *prop* is taken from table 7.7, based on the value of *pname*, and *maxSize* is taken to specify a sufficiently large buffer to receive all values that would be written to *params*.

To determine the set of active atomic counter buffer binding points used by a program, applications can query the properties and active resources of the ATOMIC_COUNTER_BUFFER interface of a program.

Additionally, the command

```
void GetActiveAtomicCounterBufferiv( uint program,
    uint bufferIndex, enum pname, int *params );
```

can be used to determine properties of active atomic counter buffer bindings used by *program* and is equivalent to

```
GLenum prop;
GetProgramResourceiv(program, ATOMIC_COUNTER_BUFFER,
    bufferIndex, 1, &prop, maxSize, NULL, params);
```

pname	*prop*
ATOMIC_COUNTER_BUFFER_BINDING	BUFFER_BINDING
ATOMIC_COUNTER_BUFFER_DATA_-SIZE	BUFFER_DATA_SIZE
ATOMIC_COUNTER_BUFFER_ACTIVE_-ATOMIC_COUNTERS	NUM_ACTIVE_VARIABLES
ATOMIC_COUNTER_BUFFER_ACTIVE_-ATOMIC_COUNTER_INDICES	ACTIVE_VARIABLES
ATOMIC_COUNTER_BUFFER_-REFERENCED_BY_VERTEX_SHADER	REFERENCED_BY_VERTEX_SHADER
ATOMIC_COUNTER_BUFFER_-REFERENCED_BY_TESS_CONTROL_-SHADER	REFERENCED_BY_TESS_CONTROL_-SHADER
ATOMIC_COUNTER_BUFFER_-REFERENCED_BY_TESS_-EVALUATION_SHADER	REFERENCED_BY_TESS_-EVALUATION_SHADER
ATOMIC_COUNTER_BUFFER_-REFERENCED_BY_GEOMETRY_SHADER	REFERENCED_BY_GEOMETRY_SHADER
ATOMIC_COUNTER_BUFFER_-REFERENCED_BY_FRAGMENT_SHADER	REFERENCED_BY_FRAGMENT_SHADER
ATOMIC_COUNTER_BUFFER_-REFERENCED_BY_COMPUTE_SHADER	REFERENCED_BY_COMPUTE_SHADER

Table 7.8: **GetProgramResourceiv** properties used by **GetActiveAtomicCounterBufferiv**.

where the value of *prop* is taken from table 7.8, based on the value of *pname*, and *maxSize* is taken to specify a sufficiently large buffer to receive all values that would be written to *params*.

7.6.1 Loading Uniform Variables In The Default Uniform Block

To load values into the uniform variables except for subroutine uniforms and atomic counters, of the default uniform block of the active program object, use the commands

```
void Uniform{1234}{ifd ui}( int location, T value );
void Uniform{1234}{ifd ui}v( int location, sizei count,
   const T *value );
```

```
void UniformMatrix{234}{fd}v( int location, sizei count,
    boolean transpose, const float *value );
void UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}{fd}v(
    int location, sizei count, boolean transpose, const
    float *value );
```

If a non-zero program object is bound by **UseProgram**, it is the active program object whose uniforms are updated by these commands. If no program object is bound using **UseProgram**, the active program object of the current program pipeline object set by **ActiveShaderProgram** is the active program object. If the current program pipeline object has no active program or there is no current program pipeline object, then there is no active program.

The given values are loaded into the default uniform block uniform variable location identified by *location* and associated with a uniform variable.

The **Uniform*f{v}** commands will load *count* sets of one to four floating-point values into a uniform defined as a float, a floating-point vector, or an array of either of these types.

The **Uniform*d{v}** commands will load *count* sets of one to four double-precision floating-point values into a uniform defined as a double, a double vector, or an array of either of these types.

The **Uniform*i{v}** commands will load *count* sets of one to four integer values into a uniform defined as a sampler, an image, an integer, an integer vector, or an array of any of these types. Only the **Uniform1i{v}** commands can be used to load sampler and image values (see sections 7.11 and 7.12).

The **Uniform*ui{v}** commands will load *count* sets of one to four unsigned integer values into a uniform defined as a unsigned integer, an unsigned integer vector, or an array of either of these types.

The **UniformMatrix{234}fv** and **UniformMatrix{234}dv** commands will load *count* 2×2, 3×3, or 4×4 matrices (corresponding to **2**, **3**, or **4** in the command name) of single- or double-precision floating-point values, respectively, into a uniform defined as a matrix or an array of matrices. If *transpose* is FALSE, the matrix is specified in column major order, otherwise in row major order.

The **UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv** and **UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}dv** commands will load *count* 2×3, 3×2, 2×4, 4×2, 3×4, or 4×3 matrices (corresponding to the numbers in the command name) of single- or double-precision floating-point values, respectively, into a uniform defined as a matrix or an array of matrices. The first number in the command name is the number of columns; the second is the number of rows. For example, **UniformMatrix2x4fv** is used to load a single-precision matrix consisting of two columns and four rows. If *transpose* is FALSE, the matrix is

specified in column major order, otherwise in row major order.

When loading values for a uniform declared as a boolean, a boolean vector, or an array of either of these types, any of the **Uniform*i{v}**, **Uniform*ui{v}**, and **Uniform*f{v}** commands can be used. Type conversion is done by the GL. Boolean values are set to `FALSE` if the corresponding input value is 0 or 0.0f, and set to `TRUE` otherwise. The **Uniform*** command used must match the size of the uniform, as declared in the shader. For example, to load a uniform declared as a `bvec2`, any of the **Uniform2{if ui}*** commands may be used.

For all other uniform types loadable with **Uniform*** commands, the command used must match the size and type of the uniform, as declared in the shader, and no type conversions are done. For example, to load a uniform declared as a `vec4`, **Uniform4f{v}** must be used, and to load a uniform declared as a `dmat3`, **UniformMatrix3dv** must be used.

When loading N elements starting at an arbitrary position k in a uniform declared as an array, elements k through $k + N - 1$ in the array will be replaced with the new values. Values for any array element that exceeds the highest array element index used, as reported by **GetActiveUniform**, will be ignored by the GL.

If the value of *location* is -1, the **Uniform*** commands will silently ignore the data passed in, and the current uniform values will not be changed.

Errors

An `INVALID_VALUE` error is generated if *count* is negative.

An `INVALID_VALUE` error is generated if **Uniform1i{v}** is used to set a sampler uniform to a value less than zero or greater than or equal to the value of `MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

An `INVALID_VALUE` error is generated if **Uniform1i{v}** is used to set an image uniform to a value less than zero or greater than or equal to the value of `MAX_IMAGE_UNITS`.

An `INVALID_OPERATION` error is generated if any of the following conditions occur:

- the size indicated in the name of the **Uniform*** command used does not match the size of the uniform declared in the shader,
- the component type and count indicated in the name of the **Uniform*** command used does not match the type of the uniform declared in the shader, where a `boolean` uniform component type is considered to match any of the **Uniform*i{v}**, **Uniform*ui{v}**, or **Uniform*f{v}**

commands.

- *count* is greater than one, and the uniform declared in the shader is not an array variable,
- no variable with a location of *location* exists in the program object currently in use and *location* is not -1, or
- a sampler or image uniform is loaded with any of the **Uniform*** commands other than **Uniform1i{v}**.
- there is no active program object in use.

To load values into the uniform variables of the default uniform block of a program which may not necessarily be bound, use the commands

```
void ProgramUniform{1234}{ifd}( uint program,
    int location, T value );
void ProgramUniform{1234}{ifd}v( uint program,
    int location, sizei count, const T *value );
void ProgramUniform{1234}ui( uint program, int location,
    T value );
void ProgramUniform{1234}uiv( uint program,
    int location, sizei count, const T *value );
void ProgramUniformMatrix{234}{fd}v( uint program,
    int location, sizei count, boolean transpose, const
    T *value );
void ProgramUniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}{fd}v(
    uint program, int location, sizei count,
    boolean transpose, const T *value );
```

These commands operate identically to the corresponding commands above without **Program** in the command name except, rather than updating the currently active program object, these **Program** commands update the program object named by the initial *program* parameter. The remaining parameters following the initial *program* parameter match the parameters for the corresponding non-**Program** uniform command.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_OPERATION error is generated if *program* has not been linked successfully.

In addition, all errors described for the corresponding **Uniform*** commands apply.

7.6.2 Uniform Blocks

The values of uniforms arranged in named uniform blocks are extracted from buffer object storage. The mechanisms for placing individual uniforms in a buffer object and connecting a uniform block to an individual buffer object are described below.

If the number of active uniform blocks referenced by the shaders in a program exceeds implementation-dependent limits, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shaders can be obtained by calling **GetIntegerv** with *pname* values of MAX_VERTEX_UNIFORM_BLOCKS, MAX_TESS_CONTROL_UNIFORM_BLOCKS, MAX_TESS_EVALUATION_UNIFORM_BLOCKS, MAX_GEOMETRY_UNIFORM_BLOCKS, MAX_FRAGMENT_UNIFORM_BLOCKS, and MAX_COMPUTE_UNIFORM_BLOCKS, respectively.

Additionally, a program will fail to link if the sum of the number of active uniform blocks referenced by each shader stage in a program exceeds the value of the implementation-dependent limit MAX_COMBINED_UNIFORM_BLOCKS. If a uniform block is referenced by multiple shaders, each such reference counts separately against this combined limit.

Finally, the total amount of buffer object storage available for any given uniform block is subject to an implementation-dependent limit. The maximum amount of available space, in basic machine units, can be queried by calling **GetIntegerv** with a *pname* of MAX_UNIFORM_BLOCK_SIZE. If the amount of storage required for a uniform block exceeds this limit, a program will fail to link.

When a named uniform block is declared by multiple shaders in a program, it must be declared identically in each shader. The uniforms within the block must be declared with the same names, types and layout qualifiers, and in the same order. If a program contains multiple shaders with different declarations for the same named uniform block, the program will fail to link.

7.6.2.1 Uniform Buffer Object Storage

When stored in buffer objects associated with uniform blocks, uniforms are represented in memory as follows:

- Members of type `bool`, `int`, `uint`, `float`, and `double` are respectively extracted from a buffer object by reading a single `uint`, `int`, `uint`, `float`, or `double` value at the specified offset.

- Vectors with N elements with basic data types of `bool`, `int`, `uint`, `float`, or `double` are extracted as N values in consecutive memory locations beginning at the specified offset, with components stored in order with the first (X) component at the lowest offset. The GL data type used for component extraction is derived according to the rules for scalar members above.

- Column-major matrices with C columns and R rows (using the types `dmat`C`x`R and `mat`C`x`R for double-precision and floating-point components respectively, or simply `dmat`C and `mat`C respectively if $C = R$) are treated as an array of C column vectors, each consisting of R double-precision or floating-point components. The column vectors will be stored in order, with column zero at the lowest offset. The difference in offsets between consecutive columns of the matrix will be referred to as the column stride, and is constant across the matrix. The column stride is an implementation-dependent function of the matrix type, and may be determined after a program is linked by querying the `MATRIX_STRIDE` property using **GetProgramResourceiv** (see section 7.3.1).

- Row-major matrices with C columns and R rows (using the types `dmat`C`x`R and `mat`C`x`R for double-precision and floating-point components respectively, or simply `dmat`C and `mat`C respectively if $C = R$) are treated as an array of R row vectors, each consisting of C double-precision or floating-point components. The row vectors will be stored in order, with row zero at the lowest offset. The difference in offsets between consecutive rows of the matrix will be referred to as the row stride, and is constant across the matrix. The row stride is an implementation-dependent function of the matrix type, and may be determined after a program is linked by querying the `MATRIX_STRIDE` property using **GetProgramResourceiv** (see section 7.3.1).

- Arrays of scalars, vectors, and matrices are stored in memory by element order, with array member zero at the lowest offset. The difference in offsets between each pair of elements in the array in basic machine units is referred to as the array stride, and is constant across the entire array. The array stride (the value of `UNIFORM_ARRAY_STRIDE`) is an implementation-dependent value and may be queried after a program is linked.

7.6.2.2 Standard Uniform Block Layout

By default, uniforms contained within a uniform block are extracted from buffer storage in an implementation-dependent manner. Applications may query the offsets assigned to uniforms inside uniform blocks with query functions provided by the GL.

The `layout` qualifier provides shaders with control of the layout of uniforms within a uniform block. When the `std140` layout is specified, the offset of each uniform in a uniform block can be derived from the definition of the uniform block by applying the set of rules described below.

When using the `std140` storage layout, structures will be laid out in buffer storage with their members stored in monotonically increasing order based on their location in the declaration. A structure and each structure member have a base offset and a base alignment, from which an aligned offset is computed by rounding the base offset up to a multiple of the base alignment. The base offset of the first member of a structure is taken from the aligned offset of the structure itself. The base offset of all other structure members is derived by taking the offset of the last basic machine unit consumed by the previous member and adding one. Each structure member is stored in memory at its aligned offset. The members of a top-level uniform block are laid out in buffer storage by treating the uniform block as a structure with a base offset of zero.

1. If the member is a scalar consuming N basic machine units, the base alignment is N.

2. If the member is a two- or four-component vector with components consuming N basic machine units, the base alignment is $2N$ or $4N$, respectively.

3. If the member is a three-component vector with components consuming N basic machine units, the base alignment is $4N$.

4. If the member is an array of scalars or vectors, the base alignment and array stride are set to match the base alignment of a single array element, according to rules (1), (2), and (3), and rounded up to the base alignment of a `vec4`. The array may have padding at the end; the base offset of the member following the array is rounded up to the next multiple of the base alignment.

5. If the member is a column-major matrix with C columns and R rows, the matrix is stored identically to an array of C column vectors with R components each, according to rule (4).

6. If the member is an array of S column-major matrices with C columns and R rows, the matrix is stored identically to a row of $S \times C$ column vectors with R components each, according to rule (4).

7. If the member is a row-major matrix with C columns and R rows, the matrix is stored identically to an array of R row vectors with C components each, according to rule (4).

8. If the member is an array of S row-major matrices with C columns and R rows, the matrix is stored identically to a row of $S \times R$ row vectors with C components each, according to rule (4).

9. If the member is a structure, the base alignment of the structure is N, where N is the largest base alignment value of any of its members, and rounded up to the base alignment of a `vec4`. The individual members of this sub-structure are then assigned offsets by applying this set of rules recursively, where the base offset of the first member of the sub-structure is equal to the aligned offset of the structure. The structure may have padding at the end; the base offset of the member following the sub-structure is rounded up to the next multiple of the base alignment of the structure.

10. If the member is an array of S structures, the S elements of the array are laid out in order, according to rule (9).

Shader storage blocks (see section 7.8) also support the `std140 layout` qualifier, as well as a `std430` qualifier not supported for uniform blocks. When using the `std430` storage layout, shader storage blocks will be laid out in buffer storage identically to uniform and shader storage blocks using the `std140` layout, except that the base alignment and stride of arrays of scalars and vectors in rule 4 and of structures in rule 9 are not rounded up a multiple of the base alignment of a `vec4`.

7.6.2.3 SPIR-V Uniform Offsets and Strides

The SPIR-V decorations `GLSLShared` or `GLSLPacked` must not be used. A variable in the `Uniform` Storage Class decorated as a `Block` or `BufferBlock` must be explicitly laid out using the `Offset`, `ArrayStride`, and `MatrixStride` decorations. These must follow the alignment rules listed above, but not necessarily the packing rules. More specifically, explicit SPIR-V offsets and strides must obey the following:

Define the *base alignment* recursively as:

- A scalar of size N has a base alignment of N.

- A two-component vector, with components of size N, has a base alignment of $2N$.
- A three- or four-component vector, with components of size N, has a base alignment of $4N$.
- An array has a base alignment equal to the base alignment of its element type, rounded up to a multiple of 16.
- A structure has a base alignment equal to the largest base alignment of any of its members, rounded up to a multiple of 16.
- A row-major matrix of C columns has a base alignment equal to the base alignment of a vector of C matrix components.
- A column-major matrix has a base alignment equal to the base alignment of the matrix column type.

Given this definition, blocks must be laid out such that:

- The `Offset` decoration must be a multiple of its base alignment.
- Any `ArrayStride` or `MatrixStride` decoration must be an integer multiple of the base alignment of the array or matrix.
- The `Offset` decoration of a member must not place it between the end of a structure or an array and the next multiple of the base alignment of that structure or array.
- The numeric order of `Offset` decorations need not follow member declaration order.

With one exception: variables in the `Uniform` storage class with a decoration of `BufferBlock` do not need their base alignments rounded up to a multiple of 16.

7.6.3 Uniform Buffer Object Bindings

The value of an active uniform inside a named uniform block is extracted from the data store of a buffer object bound to one of an array of uniform buffer binding points. The number of binding points may be queried using **GetIntegerv** with the constant `MAX_UNIFORM_BUFFER_BINDINGS`.

Regions of buffer objects are bound as storage for uniform blocks by calling **BindBuffer*** commands (see section 6) with *target* set to `UNIFORM_BUFFER`.

Each of a program's active uniform blocks has a corresponding uniform buffer object binding point. The binding is established when a program is linked or relinked, and the initial value of the binding is specified by a `layout` qualifier (if present), or zero otherwise. The binding point can be assigned by calling

```
void UniformBlockBinding( uint program,
    uint uniformBlockIndex, uint uniformBlockBinding );
```

program is a name of a program object for which the command **LinkProgram** has been issued in the past.

If successful, **UniformBlockBinding** specifies that *program* will use the data store of the buffer object bound to the binding point *uniformBlockBinding* to extract the values of the uniforms in the uniform block identified by *uniformBlockIndex*.

When executing shaders that access uniform blocks, the binding point corresponding to each active uniform block must be populated with a buffer object with a size no smaller than the minimum required size of the uniform block (the value of `UNIFORM_BLOCK_DATA_SIZE`). For binding points populated by **BindBufferRange**, the size in question is the value of the *size* parameter. If any active uniform block is not backed by a sufficiently large buffer object, the results of shader execution may be undefined or modified, as described in section 6.4. Shaders may be executed to process the primitives and vertices specified by any command that transfers vertices to the GL. Shaders may also be executed as a result of **DrawPixels**, **Bitmap**, or **RasterPos*** commands.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_VALUE` error is generated if *uniformBlockIndex* is not an active uniform block index of *program*, or if *uniformBlockBinding* is greater than or equal to the value of `MAX_UNIFORM_BUFFER_BINDINGS`.

7.7 Atomic Counter Buffers

The values of atomic counters are backed by buffer object storage. The mechanisms for accessing individual atomic counters in a buffer object and connecting to an atomic counter are described in this section.

There is a set of implementation-dependent maximums for the number of active atomic counter buffers referenced by each shader. If the number of atomic counter buffer bindings referenced by any shader in the program exceeds the corresponding limit, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shaders can be obtained by calling **GetIntegerv** with *pname* values of `MAX_VERTEX_ATOMIC_COUNTER_BUFFERS`, `MAX_TESS_CONTROL_ATOMIC_COUNTER_BUFFERS`, `MAX_TESS_EVALUATION_ATOMIC_COUNTER_BUFFERS`, `MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS`, `MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS`, and `MAX_COMPUTE_ATOMIC_COUNTER_BUFFERS`, respectively.

Additionally, there is an implementation-dependent limit on the sum of the number of active atomic counter buffers used by each shader stage of a program. If an atomic counter buffer is used by multiple shader stages, each such use counts separately against this combined limit. The combined atomic counter buffer use limit can be obtained by calling **GetIntegerv** with a *pname* of `MAX_COMBINED_ATOMIC_COUNTER_BUFFERS`.

7.7.1 Atomic Counter Buffer Object Storage

Atomic counters stored in buffer objects are represented in memory as follows:

- Members of type `atomic_uint` are extracted from a buffer object by reading a single `uint`-typed value at the specified offset.
- Arrays of type `atomic_uint` are stored in memory by element order, with array element member zero at the lowest offset. The difference in offsets between each pair of elements in the array in basic machine units is referred to as the array stride, and is constant across the entire array. The array stride, (the value of `UNIFORM_ARRAY_STRIDE`) is always 4, and may be queried after a program is linked.

7.7.2 Atomic Counter Buffer Bindings

The value of an active atomic counter is extracted from or written to the data store of a buffer object bound to one of an array of atomic counter buffer binding points. The number of binding points may be queried by calling **GetIntegerv** with a *pname* of `MAX_ATOMIC_COUNTER_BUFFER_BINDINGS`.

Regions of buffer objects are bound as storage for atomic counters by calling one of the **BindBuffer*** commands (see section 6) with *target* set to `ATOMIC_COUNTER_BUFFER`.

Each of a program's active atomic counter buffer bindings has a corresponding atomic counter buffer binding point. This binding point is established with the `layout` qualifier in the shader text, either explicitly or implicitly, as described in the OpenGL Shading Language Specification.

When executing shaders that access atomic counters, each active atomic counter buffer must be populated with a buffer object with a size no smaller than the minimum required size for that buffer (the value of `BUFFER_DATA_SIZE` returned by **GetProgramResourceiv**). For binding points populated by **BindBufferRange**, the size in question is the value of the *size* parameter. If any active atomic counter buffer is not backed by a sufficiently large buffer object, the results of shader execution may be undefined or modified, as described in section 6.4.

7.8 Shader Buffer Variables and Shader Storage Blocks

Shaders can declare named *buffer variables*, as described in the OpenGL Shading Language Specification. Sets of buffer variables are grouped into interface blocks called *shader storage blocks*. The values of each buffer variable in a shader storage block are read from or written to the data store of a buffer object bound to the binding point associated with the block. Buffer variables are considered active in the same way as uniform variables (see section 7.6). The values of active buffer variables may be changed by executing shaders that assign values to them or perform atomic memory operations on them; by modifying the contents of the bound buffer object's data store with the commands in sections 6.2, 6.2.1, 6.3, 6.5, and 6.6; by binding a new buffer object to the binding point associated with the block; or by changing the binding point associated with the block.

Like buffer variables, shader storage blocks can be active or inactive. Whether a shader storage block is active or inactive is determined in the same way as for uniform blocks (see section 7.6). Additionally though, all members of a named shader storage block declared with a `std430` layout qualifier are considered active, even if they are not referenced in any shader in the program.

Buffer variables in shader storage blocks are represented in memory in the same way as uniforms stored in uniform blocks, as described in section 7.6.2.1. When a program is linked successfully, each active buffer variable is assigned an offset relative to the base of the buffer object binding associated with its shader storage block. For buffer variables declared as arrays and matrices, strides between array elements or matrix columns or rows will also be assigned. Offsets and strides of buffer variables will be assigned in an implementation-dependent manner unless the shader storage block is declared using the `std140` or `std430` storage `layout` qualifiers. For `std140` and `std430` shader storage blocks, offsets will be assigned

using the method described in section 7.6.2.2. If a program is re-linked, existing buffer variable offsets and strides are invalidated, and a new set of active variables, offsets, and strides will be generated.

The total amount of buffer object storage that can be accessed in any shader storage block is subject to an implementation-dependent limit. The maximum amount of available space, in basic machine units, may be queried by calling **GetIntegerv** with *pname* `MAX_SHADER_STORAGE_BLOCK_SIZE`. If the amount of storage required for the fixed size portion of any shader storage block (as reported by `BUFFER_DATA_SIZE`) exceeds this limit, a program will fail to link.

If the number of active shader storage blocks referenced by the shaders in a program exceeds implementation-dependent limits, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shaders can be obtained by calling **GetIntegerv** with *pname* values of `MAX_VERTEX_SHADER_STORAGE_BLOCKS`, `MAX_TESS_CONTROL_SHADER_STORAGE_BLOCKS`, `MAX_TESS_EVALUATION_SHADER_STORAGE_BLOCKS`, `MAX_GEOMETRY_SHADER_STORAGE_BLOCKS`, `MAX_FRAGMENT_SHADER_STORAGE_BLOCKS`, and `MAX_COMPUTE_SHADER_STORAGE_BLOCKS`, respectively.

Additionally, a program will fail to link if the sum of the number of active shader storage blocks referenced by each shader stage in a program exceeds the value of the implementation-dependent limit `MAX_COMBINED_SHADER_STORAGE_BLOCKS`. If a shader storage block in a program is referenced by multiple shaders, each such reference counts separately against this combined limit.

When a named shader storage block is declared by multiple shaders in a program, it must be declared identically in each shader. The buffer variables within the block must be declared with the same names, types, qualification, and declaration order. If a program contains multiple shaders with different declarations for the same named shader storage block, the program will fail to link.

Regions of buffer objects are bound as storage for shader storage blocks by calling one of the **BindBuffer*** commands (see section 6) with *target* `SHADER_STORAGE_BUFFER`.

Each of a program's active shader storage blocks has a corresponding shader storage buffer object binding point. When a program object is linked, the shader storage buffer object binding point assigned to each of its active shader storage blocks is reset to the value specified by the corresponding `binding layout` qualifier, if present, or zero otherwise. After a program is linked, the command

```
void ShaderStorageBlockBinding( uint program,
    uint storageBlockIndex, uint storageBlockBinding );
```

changes the active shader storage block with an assigned index of *storageBlockIndex* in program object *program*. **ShaderStorageBlockBinding** specifies that *program* will use the data store of the buffer object bound to the binding point *storageBlockBinding* to read and write the values of the buffer variables in the shader storage block identified by *storageBlockIndex*.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An `INVALID_VALUE` error is generated if *storageBlockIndex* is not an active shader storage block index in *program*, or if *storageBlockBinding* is greater than or equal to the value of `MAX_SHADER_STORAGE_BUFFER_BINDINGS`.

When executing shaders that access shader storage blocks, the binding point corresponding to each active shader storage block must be populated with a buffer object with a size no smaller than the minimum required size of the shader storage block (the value of `BUFFER_SIZE` for the appropriate `SHADER_STORAGE_BUFFER` resource). For binding points populated by **BindBufferRange**, the size in question is the value of the *size* parameter or the size of the buffer minus the value of the *offset* parameter, whichever is smaller. If any active shader storage block is not backed by a sufficiently large buffer object, the results of shader execution may be undefined or modified, as described in section 6.4.

7.9 Invocation Groups

An *invocation group* (see section 8.18("Shader Invocation Group Functions") of the OpenGL Shading Language Specification) for a compute shader is the set of invocations in a single workgroup. For graphics shaders, an invocation group is an implementation-dependent subset[1] of the set of shader invocations of a given shader stage which are produced by a single drawing command. For **MultiDraw*** commands with *drawcount* greater than one, invocations from separate draws are in distinct invocation groups.

[1] Because the partitioning of invocations into invocation groups is implementation-dependent and not observable, applications generally need to assume the worst case of all invocations in a draw belong to a single invocation group.

7.10 Subroutine Uniform Variables

Subroutine uniform variables are similar to uniform variables, except they are context state rather than program state, and apply only to a single program stage. Having subroutine uniforms be context state allows them to have different values if the program is used in multiple contexts simultaneously. There is a set of subroutine uniforms for each shader stage.

A subroutine uniform may have an explicit location specified in the shader. At link time, all active subroutine uniforms without an explicit location will be assigned a unique location. The value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS for a program object is the largest specified or assigned location plus one. An assigned location will never take the location of an explicitly specified location, even if that subroutine uniform is inactive. Between the location zero and the value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS minus one there may be unused locations, either because they were not assigned a subroutine uniform or because the subroutine uniform was determined to be inactive by the linker. These locations will be ignored when assigning the subroutine index as described below.

There is an implementation-dependent limit on the number of active subroutine uniform locations in each shader stage; a program will fail to link if the number of subroutine uniform locations required is greater than the value of MAX_SUBROUTINE_UNIFORM_LOCATIONS or if an explicit subroutine uniform location is outside this limit. For active subroutine uniforms declared as arrays, the declared array elements are assigned consecutive locations.

Each function in a shader associated with a subroutine type is considered an active subroutine, unless the compiler conclusively determines that the function could never be assigned to an active subroutine uniform. The subroutine functions can be assigned an explicit index in the shader between zero and the value of MAX_SUBROUTINES minus one. At link time, all active subroutines without an explicit index will be assigned an index between zero and the value of ACTIVE_SUBROUTINES minus one. An assigned index will never take the same index of an explicitly specified index in the shader, even if that subroutine is inactive. Between index zero and the vaue of ACTIVE_SUBROUTINES minus one there may be unused indices either because they weren't assigned an index by the linker or because the subroutine was determined to be inactive by the linker. If there are no explicitly defined subroutine indices in the shader the implementation must assign indices between zero and the value of ACTIVE_SUBROUTINES minus one with no index unused. It is recommended, but not required, that the application assigns a range of tightly packed indices starting from zero to avoid indices between zero and the value of ACTIVE_SUBROUTINES minus one being unused.

To determine the set of active subroutines and subroutines used by a partic-

Interface	Shader Type
VERTEX_SUBROUTINE	VERTEX_SHADER
TESS_CONTROL_SUBROUTINE	TESS_CONTROL_SHADER
TESS_EVALUATION_SUBROUTINE	TESS_EVALUATION_SHADER
GEOMETRY_SUBROUTINE	GEOMETRY_SHADER
FRAGMENT_SUBROUTINE	FRAGMENT_SHADER
COMPUTE_SUBROUTINE	COMPUTE_SHADER

Table 7.9: Interfaces for active subroutines for a particular shader type in a program.

Interface	Shader Type
VERTEX_SUBROUTINE_UNIFORM	VERTEX_SHADER
TESS_CONTROL_SUBROUTINE_UNIFORM	TESS_CONTROL_SHADER
TESS_EVALUATION_SUBROUTINE_UNIFORM	TESS_EVALUATION_SHADER
GEOMETRY_SUBROUTINE_UNIFORM	GEOMETRY_SHADER
FRAGMENT_SUBROUTINE_UNIFORM	FRAGMENT_SHADER
COMPUTE_SUBROUTINE_UNIFORM	COMPUTE_SHADER

Table 7.10: Interfaces for active subroutine uniforms for a particular shader type in a program.

ular shader stage of a program, applications can query the properties and active resources of the interfaces for the shader type, as listed in tables 7.9 and 7.10.

Additionally, dedicated commands are provided to determine properties of active subroutines and active subroutine uniforms. The commands

```
uint GetSubroutineIndex( uint program, enum shadertype,
    const char *name );
void GetActiveSubroutineName( uint program,
    enum shadertype, uint index, sizei count,
    sizei *length, char *name );
```

are equivalent to

```
GetProgramResourceIndex(program, programInterface, name);
```

and

GetProgramResourceName(*program*, *programInterface*, *index*, *count*, *length*, *name*);

respectively, where *programInterface* is taken from table 7.9 according to the value of *shadertype*.

The commands

```
int GetSubroutineUniformLocation( uint program,
    enum shadertype, const char *name );
void GetActiveSubroutineUniformName( uint program,
    enum shadertype, uint index, sizei count,
    sizei *length, char *name );
void GetActiveSubroutineUniformiv( uint program,
    enum shadertype, uint index, enum pname, int *values );
```

are equivalent to

GetProgramResourceLocation(*program*, *programInterface*, *name*);

GetProgramResourceName(*program*, *programInterface*, *index*, *count*, *length*, *name*);

and

GetProgramResourceiv(*program*, *programInterface*, *index*, 1, &*pname*, *maxSize*, NULL, *values*);

respectively, where *programInterface* is taken from table 7.10 according to the value of *shadertype*. For **GetActiveSubroutineUniformiv**, *pname* must be one of NUM_COMPATIBLE_SUBROUTINES or COMPATIBLE_SUBROUTINES, and *maxSize* is taken to specify a sufficiently large buffer to receive all values that would be written to *params*.

The command

```
void UniformSubroutinesuiv( enum shadertype, sizei count,
    const uint *indices );
```

will load all active subroutine uniforms for shader stage *shadertype* with subroutine indices from *indices*, storing *indices*[i] into the uniform at location i. The indices for any locations between zero and the value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS minus one which are not used will be ignored.

Errors

An INVALID_ENUM error is generated if *shadertype* is not one of the values in table 7.1.

An INVALID_VALUE error is generated if *count* is negative, is not equal to the value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS for the program currently in use at shader stage *shadertype*, or if the uniform at location i is used and the value in *indices*[i] is greater than or equal to the value of ACTIVE_SUBROUTINES for the shader stage.

An INVALID_VALUE error is generated if the value of *indices*[i] for a used uniform location specifies an unused subroutine index.

An INVALID_OPERATION error is generated if, for any subroutine index being loaded to a particular uniform location, the function corresponding to the subroutine index was not associated (as defined in section 6.1.2 of the OpenGL Shading Language Specification) with the type of the subroutine variable at that location.

An INVALID_OPERATION error is generated if no program is active for the shader stage identified by *shadertype*.

Each subroutine uniform must have at least one subroutine to assign to the uniform. A program will fail to link if any stage has one or more subroutine uniforms that has no subroutine associated with the subroutine type of the uniform.

When the active program for a shader stage is re-linked or changed by a call to **UseProgram**, **BindProgramPipeline**, or **UseProgramStages**, subroutine uniforms for that stage are reset to arbitrarily chosen default functions with compatible subroutine types.

7.11 Samplers

Samplers are special uniforms used in the OpenGL Shading Language to identify the texture object used for each texture lookup. The value of a sampler indicates the texture image unit being accessed. Setting a sampler's value to i selects texture image unit number i. The value of i may range from zero to the implementation-dependent maximum supported number of texture image units minus one.

The type of the sampler identifies the target on the texture image unit, as shown in table 7.3 for sampler* types. The texture object bound to that texture image unit's target is then used for the texture lookup. For example, a variable of type sampler2D selects target TEXTURE_2D on its texture image unit. Binding of texture objects to targets is done as usual with **BindTexture**. Selecting the texture image unit to bind to is done as usual with **ActiveTexture**.

The location of a sampler is queried with **GetUniformLocation**, just like any uniform variable. Sampler values must be set by calling **Uniform1i{v}**.

Errors

It is not allowed to have variables of different sampler types pointing to the same texture image unit within a program object. This situation can only be detected at the next rendering command issued which triggers shader invocations, and an `INVALID_OPERATION` error will then be generated.

Active samplers are samplers actually being used in a program object. The **LinkProgram** command determines if a sampler is active or not. The **LinkProgram** command will attempt to determine if the active samplers in the shader(s) contained in the program object exceed the maximum allowable limits. If it determines that the count of active samplers exceeds the allowable limits, then the link fails (these limits can be different for different types of shaders). Each active sampler variable counts against the limit, even if multiple samplers refer to the same texture image unit.

Errors

If this cannot be determined at link time (for example, if the program object only contains a vertex shader), then it will be determined at the next rendering command issued which triggers shader invocations, and an `INVALID_OPERATION` error will then be generated.

7.12 Images

Images are special uniforms used in the OpenGL Shading Language to identify a level of a texture to be read or written using built-in image load, store, or atomic functions in the manner described in section 8.26. The value of an image uniform is an integer specifying the image unit accessed. Image units are numbered beginning at zero, and there is an implementation-dependent number of available image units (the value of `MAX_IMAGE_UNITS`).

Note that image units used for image variables are independent of the texture image units used for sampler variables; the number of units provided by the implementation may differ. Textures are bound independently and separately to image and texture image units.

The type of an image variable must match the texture target of the image currently bound to the image unit; otherwise the result of a load, store, or atomic

operation is undefined (see section 4.1.7.2 of the OpenGL Shading Language Specification for more details).

The location of an image variable needs to be queried with **GetUniformLocation**, just like any uniform variable. Image values must be set by calling **Uniform1i{v}**.

Unlike samplers, there is no limit on the number of active image variables that may be used by a program or by any particular shader. However, given that there is an implementation-dependent limit on the number of unique image units, the actual number of images that may be used by all shaders in a program is limited.

7.13 Shader Memory Access

As described in the OpenGL Shading Language Specification, shaders may perform random-access reads and writes to buffer object memory by reading from, assigning to, or performing atomic memory operation on shader buffer variables, or to texture or buffer object memory by using built-in image load, store, or atomic functions operating on shader image variables, or performing atomic operations on atomic counter variables. The ability to perform such random-access reads and writes in systems that may be highly pipelined results in ordering and synchronization issues discussed in the sections below.

7.13.1 Shader Memory Access Ordering

The order in which texture or buffer object memory is read or written by shaders is largely undefined. For some shader types (vertex, tessellation evaluation, and in some cases, fragment), even the number of shader invocations that might perform loads and stores is undefined.

In particular, the following rules apply:

- While a vertex or tessellation evaluation shader will be executed at least once for each unique vertex specified by the application (vertex shaders) or generated by the tessellation primitive generator (tessellation evaluation shaders), it may be executed more than once for implementation-dependent reasons. Additionally, if the same vertex is specified multiple times in a collection of primitives (e.g., repeating an index in **DrawElements**), the vertex shader might be run only once.

- For each fragment generated by the GL, the number of fragment shader invocations depends on a number of factors. If the fragment fails the pixel

ownership test (see section 14.9.1), scissor test (see section 14.9.2), or is discarded by any of the multisample fragment operations (see section 14.9.3), the fragment shader will not be executed

In addition, if early per-fragment tests are enabled (see section 14.9), the fragment shader will not be executed if the fragment is discarded during the early per-fragment tests.

When fragment shaders are executed, the number of invocations per fragment is exactly one when the framebuffer has no multisample buffer (the value of `SAMPLE_BUFFERS` is zero). Otherwise, the number of invocations is in the range $[1, N]$ where N is the number of samples covered by the fragment; if the fragment shader specifies per-sample shading, it will be invoked exactly N times.

- If a fragment shader is invoked to process fragments or samples not covered by a primitive being rasterized to facilitate the approximation of derivatives for texture lookups, then stores, atomics, and atomic counter updates have no effect.

- The relative order of invocations of the same shader type are undefined. A store issued by a shader when working on primitive B might complete prior to a store for primitive A, even if primitive A is specified prior to primitive B. This applies even to fragment shaders; while fragment shader outputs are always written to the framebuffer in primitive order, stores executed by fragment shader invocations are not.

- The relative order of invocations of different shader types is largely undefined. However, when executing a shader whose inputs are generated from a previous programmable stage, the shader invocations from the previous stage are guaranteed to have executed far enough to generate final values for all next-stage inputs. That implies shader completion for all stages except geometry; geometry shaders are guaranteed only to have executed far enough to emit all vertices used to generate the primitive being processed by the fragment shader.

The above limitations on shader invocation order also make some forms of synchronization between shader invocations within a single set of primitives unimplementable. For example, having one invocation poll memory written by another invocation assumes that the other invocation has been launched and can complete its writes. The only case where such a guarantee is made is when the inputs of

one shader invocation are generated from the outputs of a shader invocation in a previous stage.

Stores issued to different memory locations within a single shader invocation may not be visible to other invocations in the order they were performed. The built-in function `memoryBarrier` may be used to provide stronger ordering of reads and writes performed by a single invocation. Calling `memoryBarrier` guarantees that any memory transactions issued by the shader invocation prior to the call complete prior to the memory transactions issued after the call. Memory barriers may be needed for algorithms that require multiple invocations to access the same memory and require the operations to be performed in a partially-defined relative order. For example, if one shader invocation does a series of writes, followed by a `memoryBarrier` call, followed by another write, and another invocation sees the result of the final write and calls `memoryBarrier`, then the second invocation will also see the previous writes. Without either memory barrier, the final write may be visible before the previous writes.

The built-in atomic memory transaction and atomic counter functions may be used to read and write a given memory address atomically. While built-in atomic functions issued by multiple shader invocations are executed in undefined order relative to each other, these functions perform both a read and a write of a memory address and guarantee that no other memory transaction will write to the underlying memory between the read and write. Atomics allow shaders to use shared global addresses for mutual exclusion or as counters, among other uses.

7.13.2 Shader Memory Access Synchronization

Data written to textures or buffer objects by a shader invocation may eventually be read by other shader invocations, sourced by other fixed pipeline stages, or read back by the application. When data is written using API commands such as **TexSubImage*** or **BufferSubData**, the GL implementation knows when and where writes occur and can perform implicit synchronization to ensure that operations requested before the update see the original data and that subsequent operations see the modified data. Without logic to track the target address of each shader instruction performing a store, automatic synchronization of stores performed by a shader invocation would require the GL implementation to make worst-case assumptions at significant performance cost. To permit cases where textures or buffers may be read or written in different pipeline stages without the overhead of automatic synchronization, buffer object and texture stores performed by shaders are not automatically synchronized with other GL operations using the same memory.

Explicit synchronization is required to ensure that the effects of buffer and texture data stores performed by shaders will be visible to subsequent operations using

the same objects and will not overwrite data still to be read by previously requested operations. Without manual synchronization, shader stores for a "new" primitive may complete before processing of an "old" primitive completes. Additionally, stores for an "old" primitive might not be completed before processing of a "new" primitive starts. The command

`void` **MemoryBarrier**(`bitfield` *barriers*);

defines a barrier ordering the memory transactions issued prior to the command relative to those issued after the barrier. For the purposes of this ordering, memory transactions performed by shaders are considered to be issued by the rendering command that triggered the execution of the shader. *barriers* is a bitfield indicating the set of operations that are synchronized with shader stores; the bits used in *barriers* are as follows:

- `VERTEX_ATTRIB_ARRAY_BARRIER_BIT`: If set, vertex data sourced from buffer objects after the barrier will reflect data written by shaders prior to the barrier. The set of buffer objects affected by this bit is derived from the buffer object bindings used for arrays of named vertex attributes (e.g., vertex, color, normal) as well as for arrays of generic vertex attributes (`VERTEX_ATTRIB_ARRAY_BUFFER` bindings).

- `ELEMENT_ARRAY_BARRIER_BIT`: If set, vertex array indices sourced from buffer objects after the barrier will reflect data written by shaders prior to the barrier. The buffer objects affected by this bit are derived from the `ELEMENT_ARRAY_BUFFER` binding.

- `UNIFORM_BARRIER_BIT`: Shader uniforms sourced from buffer objects after the barrier will reflect data written by shaders prior to the barrier.

- `TEXTURE_FETCH_BARRIER_BIT`: Texture fetches from shaders, including fetches from buffer object memory via buffer textures, after the barrier will reflect data written by shaders prior to the barrier.

- `SHADER_IMAGE_ACCESS_BARRIER_BIT`: Memory accesses using shader built-in image load, store, and atomic functions issued after the barrier will reflect data written by shaders prior to the barrier. Additionally, image stores and atomics issued after the barrier will not execute until all memory accesses (e.g., loads, stores, texture fetches, vertex fetches) initiated prior to the barrier complete.

- COMMAND_BARRIER_BIT: Command data sourced from buffer objects by **Draw*Indirect**, **MultiDraw*IndirectCount** and **DispatchComputeIndirect** commands after the barrier will reflect data written by shaders prior to the barrier. The buffer objects affected by this bit are derived from the DRAW_INDIRECT_BUFFER, PARAMETER_BUFFER and DISPATCH_INDIRECT_BUFFER bindings, respectively.

- PIXEL_BUFFER_BARRIER_BIT: Reads/writes of buffer objects via the PIXEL_PACK_BUFFER and PIXEL_UNPACK_BUFFER bindings (**ReadPixels**, **TexSubImage**, etc.) after the barrier will reflect data written by shaders prior to the barrier. Additionally, buffer object writes issued after the barrier will wait on the completion of all shader writes initiated prior to the barrier.

- TEXTURE_UPDATE_BARRIER_BIT: Writes to a texture via **Tex(Sub)Image***, **ClearTex*Image**, **CopyTex***, or **CompressedTex***, and reads via **GetTexImage** after the barrier will not execute until all shader writes initiated prior to the barrier complete.

- BUFFER_UPDATE_BARRIER_BIT: Reads and writes to buffer object memory after the barrier using the commands in sections 6.2, 6.2.1, 6.3, 6.6, and 6.5 will reflect data written by shaders prior to the barrier. Additionally, writes via these commands issued after the barrier will wait on the completion of any shader writes to the same memory initiated prior to the barrier.

- CLIENT_MAPPED_BUFFER_BARRIER_BIT: Access by the client to persistent mapped regions of buffer objects will reflect data written by shaders prior to the barrier. Note that this may cause additional synchronization operations.

- QUERY_BUFFER_BARRIER_BIT: Writes of buffer objects via the QUERY_BUFFER binding (see section 4.2.3) after the barrier will reflect data written by shaders prior to the barrier. Additionally, buffer object writes issued after the barrier will wait on the completion of all shader writes initiated prior to the barrier.

- FRAMEBUFFER_BARRIER_BIT: Reads and writes via framebuffer object attachments after the barrier will reflect data written by shaders prior to the barrier. Additionally, framebuffer writes issued after the barrier will wait on the completion of all shader writes issued prior to the barrier.

- TRANSFORM_FEEDBACK_BARRIER_BIT: Writes via transform feedback bindings after the barrier will reflect data written by shaders prior to the

barrier. Additionally, transform feedback writes issued after the barrier will wait on the completion of all shader writes issued prior to the barrier.

- `ATOMIC_COUNTER_BARRIER_BIT`: Memory accesses using shader atomic counter built-in functions issued after the barrier will reflect data written by shaders prior to the barrier. Additionally, atomic counter function invocations after the barrier will not execute until all memory accesses (e.g., loads, stores, texture fetches, vertex fetches) initiated prior to the barrier complete.
- `SHADER_STORAGE_BARRIER_BIT`: Memory accesses using shader buffer variables issued after the barrier will reflect data written by shaders prior to the barrier. Additionally, assignments to and atomic operations performed on shader buffer variables after the barrier will not execute until all memory accesses initiated prior to the barrier complete.

If *barriers* is `ALL_BARRIER_BITS`, shader memory accesses will be synchronized relative to all the operations described above.

Errors

An `INVALID_VALUE` error is generated if *barriers* is not the special value `ALL_BARRIER_BITS`, and has any bits set other than those described above.

Implementations may cache buffer object and texture image memory that could be written by shaders in multiple caches; for example, there may be separate caches for texture, vertex fetching, and one or more caches for shader memory accesses. Implementations are not required to keep these caches coherent with shader memory writes. Stores issued by one invocation may not be immediately observable by other pipeline stages or other shader invocations because the value stored may remain in a cache local to the processor executing the store, or because data overwritten by the store is still in a cache elsewhere in the system. When **MemoryBarrier** is called, the GL flushes and/or invalidates any caches relevant to the operations specified by the *barriers* parameter to ensure consistent ordering of operations across the barrier.

To allow for independent shader invocations to communicate by reads and writes to a common memory address, image variables in the OpenGL Shading Language may be declared as `coherent`. Buffer object or texture image memory accessed through such variables may be cached only if caches are automatically updated due to stores issued by any other shader invocation. If the same address is accessed using both coherent and non-coherent variables, the accesses using variables declared as coherent will observe the results stored using coherent variables in other invocations. Using variables declared as `coherent` guarantees only

that the results of stores will be immediately visible to shader invocations using similarly-declared variables; calling **MemoryBarrier** is required to ensure that the stores are visible to other operations.

The following guidelines may be helpful in choosing when to use coherent memory accesses and when to use barriers.

- Data that are read-only or constant may be accessed without using coherent variables or calling **MemoryBarrier**. Updates to the read-only data via commands such as **BufferSubData** will invalidate shader caches implicitly as required.
- Data that are shared between shader invocations at a fine granularity (e.g., written by one invocation, consumed by another invocation) should use coherent variables to read and write the shared data.
- Data written by one shader invocation and consumed by other shader invocations launched as a result of its execution (*dependent invocations*) should use coherent variables in the producing shader invocation and call `memoryBarrier` after the last write. The consuming shader invocation should also use coherent variables.
- Data written to image variables in one rendering pass and read by the shader in a later pass need not use coherent variables or `memoryBarrier`. Calling **MemoryBarrier** with the `SHADER_IMAGE_ACCESS_BARRIER_BIT` set in *barriers* between passes is necessary.
- Data written by the shader in one rendering pass and read by another mechanism (e.g., vertex or index buffer pulling) in a later pass need not use coherent variables or `memoryBarrier`. Calling **MemoryBarrier** with the appropriate bits set in *barriers* between passes is necessary.

The command

`void` **MemoryBarrierByRegion**(`bitfield` *barriers*);

behaves as described above for **MemoryBarrier**, with two differences:

First, it narrows the region under consideration so that only reads and writes of prior fragment shaders that are invoked for a smaller region of the framebuffer will be completed/reflected prior to subsequent reads and writes of following fragment shaders. The size of the region is implementation-dependent and may be as small as one framebuffer pixel.

Second, it only applies to memory transactions that may be read by or written by a fragment shader. Therefore, only the barrier bits

- ATOMIC_COUNTER_BARRIER_BIT
- FRAMEBUFFER_BARRIER_BIT
- SHADER_IMAGE_ACCESS_BARRIER_BIT
- SHADER_STORAGE_BARRIER_BIT
- TEXTURE_FETCH_BARRIER_BIT
- UNIFORM_BARRIER_BIT

are supported.

When barriers is ALL_BARRIER_BITS, shader memory accesses will be synchronized relative to all these barrier bits, but not to other barrier bits specific to **MemoryBarrier**. This implies that reads and writes for scatter/gather-like algorithms may or may not be completed/reflected after a **MemoryBarrierByRegion** command. However, for uses such as deferred shading, where a linked list of visible surfaces with the head at a framebuffer address may be constructed, and the entirety of the list is only dependent on previous executions at that framebuffer address, **MemoryBarrierByRegion** may be significantly more efficient than **MemoryBarrier**.

Errors

An INVALID_VALUE error is generated if *barriers* is not the special value ALL_BARRIER_BITS, and has any bits set other than those described above.

7.14 Shader, Program, and Program Pipeline Queries

The command

`void` **GetShaderiv**(`uint` *shader*, `enum` *pname*, `int` **params*);

returns properties of the shader object named *shader* in *params*. The parameter value to return is specified by *pname*.

If *pname* is SHADER_TYPE, one of the values from table 7.1 corresponding to the type of *shader* is returned.

If *pname* is DELETE_STATUS, TRUE is returned if the shader has been flagged for deletion and FALSE is returned otherwise.

If *pname* is COMPILE_STATUS, TRUE is returned if the shader was last compiled or specialized successfully, and FALSE is returned otherwise.

If *pname* is INFO_LOG_LENGTH, the length of the info log, including a null terminator, is returned. If there is an empty info log, zero is returned.

If *pname* is SHADER_SOURCE_LENGTH, the length of the concatenation of the source strings making up the shader source, including a null terminator, is returned. If no source has been defined, zero is returned.

If *pname* is SPIR_V_BINARY, TRUE is returned if the shader has been successfully associated with a SPIR-V binary module by the **ShaderBinary** command, and FALSE is returned otherwise.

Errors

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

An INVALID_ENUM error is generated if *pname* is not SHADER_TYPE, DELETE_STATUS, COMPILE_STATUS, INFO_LOG_LENGTH, SHADER_SOURCE_LENGTH, or SPIR_V_BINARY.

The command

```
void GetProgramiv( uint program, enum pname,
    int *params );
```

returns properties of the program object named *program* in *params*. The parameter value to return is specified by *pname*.

Most properties set within program objects are specified not to take effect until the next call to **LinkProgram** or **ProgramBinary**. Some properties further require a successful call to either of these commands before taking effect. **GetProgramiv** returns the properties currently in effect for *program*, which may differ from the properties set within *program* since the most recent call to **LinkProgram** or **ProgramBinary**, which have not yet taken effect. If there has been no such call putting changes to *pname* into effect, initial values are returned.

If *pname* is DELETE_STATUS, TRUE is returned if the program has been flagged for deletion, and FALSE is returned otherwise.

If *pname* is LINK_STATUS, TRUE is returned if the program was last linked successfully, and FALSE is returned otherwise.

If *pname* is VALIDATE_STATUS, TRUE is returned if the last call to **ValidateProgram** (see section 11.1.3.12) with *program* was successful, and FALSE is returned otherwise.

If *pname* is INFO_LOG_LENGTH, the length of the info log, including a null terminator, is returned. If there is an empty info log, zero is returned.

If *pname* is ATTACHED_SHADERS, the number of objects attached is returned.

If *pname* is ACTIVE_ATTRIBUTES, the number of active attributes (see section 7.3.1) in *program* is returned. If no active attributes exist, zero is returned.

If *pname* is ACTIVE_ATTRIBUTE_MAX_LENGTH, the length of the longest active attribute name, including a null terminator, is returned. If no active attributes exist, zero is returned. If no name reflection information is available, one is returned.

If *pname* is ACTIVE_UNIFORMS, the number of active uniforms is returned. If no active uniforms exist, zero is returned.

If *pname* is ACTIVE_UNIFORM_MAX_LENGTH, the length of the longest active uniform name, including a null terminator, is returned. If no active uniforms exist, zero is returned. If no name reflection information is available, one is returned.

If *pname* is TRANSFORM_FEEDBACK_BUFFER_MODE, the buffer mode used when transform feedback (see section 11.1.2.1) is active is returned. It can be one of SEPARATE_ATTRIBS or INTERLEAVED_ATTRIBS.

If *pname* is TRANSFORM_FEEDBACK_VARYINGS, the number of output variables to capture in transform feedback mode for the program is returned.

If *pname* is TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, the length of the longest output variable name specified to be used for transform feedback, including a null terminator, is returned. If no outputs are used for transform feedback, zero is returned. If no name reflection information is available, one is returned.

If *pname* is ACTIVE_UNIFORM_BLOCKS, the number of uniform blocks for *program* containing active uniforms is returned.

If *pname* is ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, the length of the longest active uniform block name, including the null terminator, is returned. If no active uniform blocks exist, zero is returned. If no name reflection information is available, one is returned.

If *pname* is GEOMETRY_VERTICES_OUT, the maximum number of vertices the geometry shader (see section 11.3) will output is returned.

If *pname* is GEOMETRY_INPUT_TYPE, the geometry shader input type, which must be one of POINTS, LINES, LINES_ADJACENCY, TRIANGLES or TRIANGLES_ADJACENCY, is returned.

If *pname* is GEOMETRY_OUTPUT_TYPE, the geometry shader output type, which must be one of POINTS, LINE_STRIP or TRIANGLE_STRIP, is returned.

If *pname* is GEOMETRY_SHADER_INVOCATIONS, the number of geometry shader invocations per primitive will be returned.

If *pname* is TESS_CONTROL_OUTPUT_VERTICES, the number of vertices in the tessellation control shader (see section 11.2.1) output patch is returned.

If *pname* is TESS_GEN_MODE, QUADS, TRIANGLES, or ISOLINES is returned, depending on the primitive mode declaration in the tessellation evaluation shader (see section 11.2.3).

If *pname* is TESS_GEN_SPACING, EQUAL, FRACTIONAL_EVEN, or FRACTIONAL_ODD is returned, depending on the spacing declaration in the tessellation evaluation shader.

If *pname* is TESS_GEN_VERTEX_ORDER, CCW or CW is returned, depending on the vertex order declaration in the tessellation evaluation shader.

If *pname* is TESS_GEN_POINT_MODE, TRUE is returned if point mode is enabled in a tessellation evaluation shader declaration; FALSE is returned otherwise.

If *pname* is COMPUTE_WORK_GROUP_SIZE, an array of three integers containing the workgroup size of the compute program (see chapter 19), as specified by its input layout qualifier(s), is returned.

If *pname* is PROGRAM_SEPARABLE, TRUE is returned if the program has been flagged for use as a separable program object that can be bound to individual shader stages with **UseProgramStages**.

If *pname* is PROGRAM_BINARY_RETRIEVABLE_HINT, the value of whether the binary retrieval hint is enabled for *program* is returned.

If *pname* is ACTIVE_ATOMIC_COUNTER_BUFFERS, the number of active atomic counter buffers used by *program* is returned.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *pname* is not one of the values listed above.

An INVALID_OPERATION error is generated if GEOMETRY_VERTICES_OUT, GEOMETRY_INPUT_TYPE, GEOMETRY_OUTPUT_TYPE, or GEOMETRY_SHADER_INVOCATIONS are queried for a program which has not been linked successfully, or which does not contain objects to form a geometry shader.

An INVALID_OPERATION error is generated if TESS_CONTROL_OUTPUT_VERTICES is queried for a program which has not been linked successfully, or which does not contain objects to form a tessellation control shader.

An INVALID_OPERATION error is generated if TESS_GEN_MODE, TESS_GEN_SPACING, TESS_GEN_VERTEX_ORDER, or TESS_GEN_POINT_

MODE are queried for a program which has not been linked successfully, or which does not contain objects to form a tessellation evaluation shader.

An INVALID_OPERATION error is generated if COMPUTE_WORK_GROUP_SIZE is queried for a program which has not been linked successfully, or which does not contain objects to form a compute shader,

The command

```
void GetProgramPipelineiv( uint pipeline, enum pname,
    int *params );
```

returns properties of the program pipeline object named *pipeline* in *params*. The parameter value to return is specified by *pname*.

If *pipeline* is a name that has been generated (without subsequent deletion) by **GenProgramPipelines**, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when **BindProgramPipeline** creates a new program pipeline object.

If *pname* is ACTIVE_PROGRAM, the name of the active program object (used for uniform updates) of *pipeline* is returned.

If *pname* is one of the shader stage *type* arguments in table 7.1, the name of the program object current for the corresponding shader stage of *pipeline* is returned.

If pname is VALIDATE_STATUS, the validation status of *pipeline*, as determined by **ValidateProgramPipeline** (see section 11.1.3.12) is returned.

If *pname* is INFO_LOG_LENGTH, the length of the info log for *pipeline*, including a null terminator, is returned. If there is an empty info log, zero is returned.

Errors

An INVALID_OPERATION error is generated if *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**.

An INVALID_ENUM error is generated if *pname* is not ACTIVE_PROGRAM, INFO_LOG_LENGTH, VALIDATE_STATUS, or one of the *type* arguments in table 7.1.

The command

```
void GetAttachedShaders( uint program, sizei maxCount,
    sizei *count, uint *shaders );
```

returns the names of shader objects attached to *program* in *shaders*. The actual number of shader names written into *shaders* is returned in *count*. If no shaders

are attached, *count* is set to zero. If *count* is NULL then it is ignored. The maximum number of shader names that may be written into *shaders* is specified by *maxCount*. The number of objects attached to *program* may be queried by calling **GetProgramiv** with ATTACHED_SHADERS.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_VALUE error is generated if *maxCount* is negative.

A string that contains information about the last compilation attempt on a shader object, last link or validation attempt on a program object, or last validation attempt on a program pipeline object, called the *info log*, can be obtained with the commands

```
void GetShaderInfoLog( uint shader, sizei bufSize,
    sizei *length, char *infoLog );
void GetProgramInfoLog( uint program, sizei bufSize,
    sizei *length, char *infoLog );
void GetProgramPipelineInfoLog( uint pipeline,
    sizei bufSize, sizei *length, char *infoLog );
```

These commands return an info log string for the corresponding type of object in *infoLog*. This string will be null-terminated even if the INFO_LOG_LENGTH query returns zero. The actual number of characters written into *infoLog*, excluding the null terminator, is returned in *length*. If *length* is NULL, then no length is returned. The maximum number of characters that may be written into *infoLog*, including the null terminator, is specified by *bufSize*. The number of characters in the info log for a shader object, program object, or program pipeline object may be queried respectively with **GetShaderiv**, **GetProgramiv**, or **GetProgramPipelineiv** with *pname* INFO_LOG_LENGTH.

If *shader* is a shader object, **GetShaderInfoLog** will return either an empty string or information about the last compilation or specialization attempt for that object.

If *program* is a program object, **GetProgramInfoLog** will return either an empty string or information about the last link attempt or last validation attempt (see section 11.1.3.12) for that object.

If *pipeline* is a program pipeline object, **GetProgramPipelineInfoLog** will return either an empty string or information about the last validation attempt for that object.

The info log is typically only useful during application development and an application should not expect different GL implementations to produce identical info logs.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

An INVALID_VALUE error is generated if *pipeline* is not the name of an existing program pipeline object.

An INVALID_VALUE error is generated if *bufSize* is negative.

The command

```
void GetShaderSource( uint shader, sizei bufSize,
   sizei *length, char *source );
```

returns in *source* the string making up the source code for the shader object *shader*. The string *source* will be null-terminated. The actual number of characters written into *source*, excluding the null terminator, is returned in *length*. If *length* is NULL, no length is returned. The maximum number of characters that may be written into *source*, including the null terminator, is specified by *bufSize*. The string *source* is a concatenation of the strings passed to the GL using **ShaderSource**. The length of this concatenation is given by SHADER_SOURCE_LENGTH, which may be queried with **GetShaderiv**.

Errors

An INVALID_VALUE error is generated if *shader* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *shader* is the name of a program object.

An INVALID_VALUE error is generated if *bufSize* is negative.

The command

```
void GetShaderPrecisionFormat( enum shadertype,
    enum precisiontype, int *range, int *precision );
```

returns the range and precision for different numeric formats supported by the shader compiler. *shadertype* must be VERTEX_SHADER or FRAGMENT_SHADER. *precisiontype* must be one of LOW_FLOAT, MEDIUM_FLOAT, HIGH_FLOAT, LOW_INT, MEDIUM_INT or HIGH_INT. *range* points to an array of two integers in which encodings of the format's numeric range are returned. If min and max are the smallest and largest values representable in the format, then the values returned are defined to be

$$range[0] = \lfloor log_2(|min|) \rfloor$$
$$range[1] = \lfloor log_2(|max|) \rfloor$$

precision points to an integer in which the number of bits of precision of the format is returned. If the smallest representable value greater than 1 is $1 + \epsilon$, then **precision* will contain $\lfloor -log_2(\epsilon) \rfloor$, and every value in the range

$$[-2^{range[0]}, 2^{range[1]}]$$

can be represented to at least one part in $2^{*precision}$. For example, an IEEE single-precision floating-point format would return $range[0] = 127$, $range[1] = 127$, and $*precision = 23$, while a 32-bit two's-complement integer format would return $range[0] = 31$, $range[1] = 30$, and $*precision = 0$.

The minimum required precision and range for formats corresponding to the different values of *precisiontype* are described in section 4.7("Precision and Precision Qualifiers") of the OpenGL Shading Language Specification.

Errors

An INVALID_ENUM error is generated if *shadertype* is not VERTEX_SHADER or FRAGMENT_SHADER.

The commands

```
void GetUniformfv( uint program, int location,
    float *params );
```

```
void GetUniformdv( uint program, int location,
   double *params );
void GetUniformiv( uint program, int location,
   int *params );
void GetUniformuiv( uint program, int location,
   uint *params );
void GetnUniformfv( uint program, int location,
   sizei bufSize, float *params );
void GetnUniformdv( uint program, int location,
   sizei bufSize, double *params );
void GetnUniformiv( uint program, int location,
   sizei bufSize, int *params );
void GetnUniformuiv( uint program, int location,
   sizei bufSize, uint *params );
```

return the value or values of the uniform at location *location* of the default uniform block for program object *program* in the array *params*. The type of the uniform at *location* determines the number of values returned.

In order to query the values of an array of uniforms, a **GetUniform*** command needs to be issued for each array element. If the uniform queried is a matrix, the values of the matrix are returned in column major order.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_OPERATION error is generated if *program* has not been linked successfully, or if *location* is not a valid location for *program*.

An INVALID_VALUE error is generated by **GetnUniform*** if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetnUniform*** if the buffer size required to store the requested data is greater than *bufSize*.

The command

```
void GetUniformSubroutineuiv( enum shadertype,
   int location, uint *params );
```

returns the value of the subroutine uniform at location *location* for shader stage

shadertype of the current program. If *location* represents an unused location, the value INVALID_INDEX is returned and no error is generated.

Errors

An INVALID_ENUM error is generated if *shadertype* is not one of the values in table 7.1.

An INVALID_VALUE error is generated if *location* is greater than or equal to the value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS for the shader currently in use at shader stage *shadertype*.

An INVALID_OPERATION error is generated if no program is active.

The command

```
void GetProgramStageiv( uint program, enum shadertype,
    enum pname, int *values );
```

returns properties of the program object *program* specific to the programmable stage corresponding to *shadertype* in *values*. The parameter value to return is specified by *pname*. If *pname* is ACTIVE_SUBROUTINE_UNIFORMS, the number of active subroutine variables in the stage is returned. If *pname* is ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS, the number of active subroutine variable locations in the stage is returned. If *pname* is ACTIVE_SUBROUTINES, the number of active subroutines in the stage is returned. If *pname* is ACTIVE_SUBROUTINE_UNIFORM_MAX_LENGTH or ACTIVE_SUBROUTINE_MAX_LENGTH, the length of the longest subroutine uniform or subroutine name, respectively, for the stage is returned. The returned name length includes space for a null terminator. If there is no shader of type *shadertype* in *program*, the values returned will be consistent with a shader with no subroutines or subroutine uniforms.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_ENUM error is generated if *shadertype* is not one of the values in table 7.1.

7.15 Required State

The GL maintains state to indicate which shader and program object names are in use. Initially, no shader or program objects exist, and no names are in use.

The state required per shader object consists of:

- An unsigned integer specifying the shader object name.
- An integer holding the value of `SHADER_TYPE`.
- A boolean holding the delete status, initially `FALSE`.
- A boolean holding the status of the last compile, initially `FALSE`.
- A boolean holding the SPIR-V binary status, initially `FALSE`.
- An array of type `char` containing the information log, initially empty.
- An integer holding the length of the information log.
- An array of type `char` containing the concatenated shader string, initially empty.
- An integer holding the length of the concatenated shader string.

The state required per program object consists of:

- An unsigned integer indicating the program object name.
- A boolean holding the delete status, initially `FALSE`.
- A boolean holding the status of the last link attempt, initially `FALSE`.
- A boolean holding the status of the last validation attempt, initially `FALSE`.
- An integer holding the number of attached shader objects.
- A list of unsigned integers to keep track of the names of the shader objects attached.
- An array of type `char` containing the information log, initially empty.
- An integer holding the length of the information log.
- An integer holding the number of active uniforms.

- For each active uniform, three integers, holding its location, size, and type, and an array of type `char` holding its name.
- An array holding the values of each active uniform.
- An integer holding the number of active attributes.
- For each active attribute, three integers holding its location, size, and type, and an array of type `char` holding its name.
- A boolean holding the hint to the retrievability of the program binary, initially `FALSE`.

Additional state required to support vertex shaders consists of:

- A bit indicating whether or not vertex program two-sided color mode is enabled, initially disabled.
- A bit indicating whether or not program point size mode (section 14.4.1) is enabled, initially disabled.

Additional state required to support transform feedback consists of:

- An integer holding the transform feedback mode, initially `INTERLEAVED_ATTRIBS`.
- An integer holding the number of outputs to be captured, initially zero.
- An integer holding the length of the longest output name being captured, initially zero.
- For each output being captured, two integers holding its size and type, and an array of type `char` holding its name.

Additionally, one unsigned integer is required to hold the name of the current program object, if any.

This list of program object state is not complete. Tables 23.43-23.53 describe additional program object state specific to program binaries, geometry shaders, tessellation control and evaluation shaders, shader subroutines, and uniform blocks.

Table 23.54 describes state related to vertex and geometry shaders that is not program object state.

Chapter 8

Textures and Samplers

Texturing maps a portion of one or more specified images onto each primitive for which texturing is enabled. This mapping is accomplished in shaders by *sampling* the color of an image at the location indicated by specified (s, t, r) *texture coordinates*. It is accomplished in fixed-function processing by using the color of an image at the location indicated by a texture coordinate set's (s, t, r, q) values. Texture lookups are typically used to modify a fragment's RGBA color but may be used for any purpose in a shader.

This chapter first describes how pixel rectangles, texture images, and texture and sampler object parameters are specified and queried, in sections 8.1-8.11. The remainder of the chapter in sections 8.12-8.26 describe how texture sampling is performed in fixed-function fragment processing and shaders.

The internal data type of a texture may be signed or unsigned normalized fixed-point, signed or unsigned integer, or floating-point, depending on the internal format of the texture. The correspondence between the internal format and the internal data type is given in tables 8.19-8.21. Fixed-point and floating-point textures return a floating-point value and integer textures return signed or unsigned integer values. When a fragment shader is active, the shader is responsible for interpreting the result of a texture lookup as the correct data type, otherwise the result is undefined. When not using a fragment shader, floating-point texture values are assumed, and the results of using either signed normalized fixed-point or integer textures in this case are undefined.

Each of the supported types of texture is a collection of *texture images* built from one-, two-, or three-dimensional arrays of texels (see section 2.6.6). One-, two-, and three-dimensional textures consist respectively of one-, two-, or three-dimensional texture images. One- and two-dimensional array textures are arrays of one- or two-dimensional images. Each image consists of one or more *layers*.

Two-dimensional multisample and two-dimensional multisample array textures are special two-dimensional and two-dimensional array textures, respectively, containing multiple samples in each texel. Cube maps are special two-dimensional array textures with six layers that represent the faces of a cube. When accessing a cube map, the texture coordinates are projected onto one of the six faces of the cube. A cube map array is a collection of cube map layers stored as a two-dimensional array texture. When accessing a cube map array, the texture coordinates s, t, and r are applied similarly as cube maps while the last texture coordinate q is used as the index of one of the cube map slices. Rectangle textures are special two-dimensional textures consisting of only a single image and accessed using unnormalized coordinates. Buffer textures are special one-dimensional textures whose texture images are stored in separate buffer objects.

Implementations must support texturing using multiple images. For fixed-function fragment processsing, each fragment or vertex carries multiple sets of texture coordinates (s, t, r, q) which are used to index separate images to produce color values which are collectively used to modify the resulting transformed vertex or fragment color. Texturing is specified only for RGBA mode; its use in color index mode is undefined.

The following subsections (up to and including section 8.14) specify the GL operation with a single texture. Multiple texture images may be sampled and combined by shaders as described in section 11.1.3.5. For fixed-function fragment processing, section 16.2 specifies the details of how multiple texture units interact.

The GL provides two ways to specify the details of how texturing of a primitive is effected. The first is referred to as *fixed-function fragment shading*, or simply *fixed-function*, and is described in this section. The second is texture access from programmable shaders, as described in section 11.1.3.5. The specification of the image to be texture mapped and the means by which the image is filtered when applied to the primitive are common to both methods and are discussed in this section. The fixed-function method for determining what RGBA value is produced is also described in this section. If a fragment shader is active, the method for determining the RGBA value is specified by an application-supplied fragment shader as described in the OpenGL Shading Language Specification.

When no fragment shader is active, and when cube map texturing is not enabled, the coordinates used for texturing are $(s/q, t/q, r/q)$, derived from the original texture coordinates (s, t, r, q). If the q texture coordinate is less than or equal to zero, the coordinates used for texturing are undefined. Otherwise, when cube map texturing is enabled, texture coordinates are treated as described in section 8.13 When a fragment shader is active, the (s, t, r, q) coordinates are available to the fragment shader.

The coordinates used for texturing in a fragment shader are defined by the OpenGL Shading Language Specification.

The command

`void` **ActiveTexture**(`enum` *texture*);

specifies the *active texture unit selector*. The selector may be queried by calling **GetIntegerv** with *pname* set to `ACTIVE_TEXTURE`.

Each texture unit contains up to two distinct sub-units: a texture coordinate processing unit consisting of a texture matrix stack and texture coordinate generation state, and a texture image unit consisting of all the texture state defined in section 8.22. In implementations with a different number of supported texture coordinate sets and texture image units, some texture units may consist of only one of the two sub-units.

The active texture unit selector selects the texture image unit accessed by commands involving texture image processing (see chapter 8). Such commands include all variants of **TexEnv** (except for those controlling point sprite coordinate replacement), **TexParameter**, **TexImage**, **BindTexture**, **Enable/Disable** for any texture target (e.g., `TEXTURE_2D`), and queries of all such state.

Errors

An `INVALID_OPERATION` error is generated by any such command if the texture image unit number corresponding to the current value of `ACTIVE_TEXTURE` is greater than or equal to the value of the implementation-dependent constant `MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

The active texture unit selector also specifies the texture coordinate set accessed by commands involving texture coordinate processing (see section 12.1.1).

Errors

An `INVALID_ENUM` error is generated if an invalid *texture* is specified. *texture* is a symbolic constant of the form `TEXTURE`i, indicating that texture unit i is to be modified. Each `TEXTURE`i adheres to `TEXTURE`i = `TEXTURE0` + i, where i is in the range zero to $k - 1$, and k is the larger of the values of `MAX_TEXTURE_COORDS` and `MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

For backwards compatibility, the implementation-dependent constant `MAX_TEXTURE_UNITS` specifies the number of conventional texture units sup-

ported by the implementation. Its value must be no larger than the minimum of MAX_TEXTURE_COORDS and MAX_COMBINED_TEXTURE_IMAGE_UNITS.

The state required for the active texture image unit selector is a single integer. The initial value is TEXTURE0.

8.1 Texture Objects

Textures in GL are represented by named objects. The name space for texture objects is the unsigned integers, with zero reserved by the GL to represent the default texture object. The default texture object is bound to each of the TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_BUFFER, TEXTURE_CUBE_MAP, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_2D_MULTISAMPLE, and TEXTURE_2D_MULTISAMPLE_ARRAY targets during context initialization.

A new texture object is created by binding an unused name to one of these texture targets. The command

`void` **GenTextures**(`sizei` *n*, `uint` **textures*);

returns *n* previously unused texture names in *textures*. These names are marked as used, for the purposes of **GenTextures** only, but they acquire texture state and a dimensionality only when they are first bound, just as if they were unused.

Errors

An INVALID_VALUE error is generated if *n* is negative.

The binding is effected by calling

`void` **BindTexture**(`enum` *target*, `uint` *texture*);

with *target* set to the desired texture target and *texture* set to the unused name. The resulting texture object is a new state vector, comprising all the state and with the same initial values listed in section 8.22. The new texture object bound to *target* is, and remains a texture of the dimensionality and type specified by *target* until it is deleted.

BindTexture may also be used to bind an existing texture object to any of these targets. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to *target* is broken.

While a texture object is bound, GL operations on the target to which it is bound affect the bound object, and queries of the target to which it is bound return state from the bound object. If texture mapping of the dimensionality of the target to which a texture object is bound is enabled, the state of the bound texture object directs the texturing operation.

Errors

An `INVALID_ENUM` error is generated if *target* is not one of the texture targets described in the introduction to section 8.1.

An `INVALID_OPERATION` error is generated if an attempt is made to bind a texture object of different dimensionality than the specified *target*.

The command

```
void BindTextures( uint first, sizei count, const
   uint *textures );
```

binds *count* existing texture objects to texture image units numbered *first* through $first + count - 1$. If *textures* is not `NULL`, it specifies an array of *count* values, each of which must be zero or the name of an existing texture object. When an entry in *textures* is the name of an existing texture object, that object is bound to the target, in the corresponding texture unit, that was specified when the object was created. When an entry in *textures* is zero, each of the targets enumerated at the beginning of this section is reset to its default texture for the corresponding texture image unit. If *textures* is `NULL`, each target of each affected texture image unit from *first* to $first + count - 1$ is reset to its default texture.

BindTextures is equivalent (assuming no errors are generated to):

```
for (i = 0; i < count; i++) {
   uint texture;
   if (textures == NULL) {
      texture = 0;
   } else {
      texture = textures[i];
   }
   ActiveTexture(TEXTURE0 + first + i);
   if (texture != 0) {
      enum target = /* target of textures[i] */;
      BindTexture(target, textures[i]);
   } else {
```

```
        for (target in all supported targets) {
            BindTexture(target, 0);
        }
    }
}
```

except that the active texture selector retains its original value upon completion of the command, and that textures will not be created if they do not exist.

The values specified in *textures* will be checked separately for each texture image unit. When a value for a specific texture image unit is invalid, the state for that texture image unit will be unchanged and an error will be generated. However, state for other texture image units will still be changed if their corresponding values are valid.

Errors

An INVALID_OPERATION error is generated if $first + count$ is greater than the number of texture image units supported by the implementation.

An INVALID_VALUE error is generated if *count* is negative.

An INVALID_OPERATION error is generated if any value in *textures* is not zero or the name of an existing texture object (per binding).

The command

`void` **BindTextureUnit**(`uint` *unit*, `uint` *texture*);

binds an existing texture object to the texture unit numbered *unit*. *texture* must be zero or the name of an existing texture object. When *texture* is the name of an existing texture object, that object is bound to the target, in the corresponding texture unit, that was specified when the object was created. When *texture* is zero, each of the targets enumerated at the beginning of this section is reset to its default texture for the corresponding texture image unit.

Errors

An INVALID_OPERATION error is generated by BindTextureUnit if *texture* is not zero or the name of an existing texture object.

Texture objects may also be created with the command

`void` **CreateTextures**(`enum` *target*, `sizei` *n*, `uint` **textures*);

CreateTextures returns *n* previously unused texture names in *textures*, each representing a new texture object that is a state vector comprising all the state and with the same initial values listed in section 8.22. The new texture objects are and remain textures of the dimensionality and type specified by *target* until they are deleted.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

Texture objects are deleted by calling

`void` **DeleteTextures**(`sizei` *n*, `const uint` **textures*);

textures contains *n* names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound to any of the target bindings of **BindTexture** is deleted, it is as though **BindTexture** had been executed with the same target and texture zero. Additionally, special care must be taken when deleting a texture if any of the images of the texture are attached to a framebuffer object. See section 9.2.8 for details.

Unused names in *textures* that have been marked as used for the purposes of **GenTextures** are marked as unused again. Unused names in *textures* are silently ignored, as is the name zero.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

The command

`boolean` **IsTexture**(`uint` *texture*);

returns `TRUE` if *texture* is the name of a texture object. If *texture* is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, **IsTexture** returns `FALSE`.

An implementation may choose to establish a working set of texture objects on which binding operations are performed with higher performance. A texture object that is currently part of the working set is said to be *resident*. The command

`boolean` **AreTexturesResident**(`sizei` *n*, `const uint` **textures*, `boolean` **residences*);

returns TRUE if all of the *n* texture objects named in *textures* are resident, or if the implementation does not distinguish a working set. If at least one of the texture objects named in *textures* is not resident, then FALSE is returned, and the residence of each texture object is returned in *residences*. Otherwise the contents of *residences* are not changed.

AreTexturesResident indicates only whether a texture object is currently resident, not whether it could not be made resident. An implementation may choose to make a texture object resident only on first use, for example. The client may guide the GL implementation in determining which texture objects should be resident by specifying a priority for each texture object.

If an error is generated, FALSE is returned.

Errors

An INVALID_VALUE error is generated if *n* is negative.

An INVALID_VALUE error is generated and FALSE is returned if any of the names in *textures* are unused or are zero.

The residence status of a single bound texture object can also be queried by calling **GetTexParameteriv** or **GetTexParameterfv** with *target* set to the target to which the texture object is bound, and *pname* set to TEXTURE_RESIDENT.

The command

```
void PrioritizeTextures( sizei n, uint *textures, const
    float *priorities );
```

sets the priorities of the *n* texture objects named in *textures* to the values in *priorities*. Each priority value is clamped to the range $[0, 1]$ before it is assigned. Zero indicates the lowest priority, with the least likelihood of being resident. One indicates the highest priority, with the greatest likelihood of being resident. The priority of a single bound texture object may also be changed by calling **TexParameteri**, **TexParameterf**, **TexParameteriv**, or **TexParameterfv** with *target* set to the target to which the texture object is bound, *pname* set to TEXTURE_PRIORITY, and *param* or *params* specifying the new priority value (which is clamped to the range $[0, 1]$ before being assigned). **PrioritizeTextures** silently ignores attempts to prioritize unused texture object names or zero (default textures).

Errors

An INVALID_VALUE error is generated if *n* is negative.

The texture object name space, including the initial one-, two-, and three- dimensional, one- and two-dimensional array, rectangle, buffer, cube map, cube map array, two-dimensional multisample, and two-dimensional multisample array texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture binding is affected by the setting of the state `ACTIVE_TEXTURE`. If a texture object is deleted, it as if all texture units which are bound to that texture object are rebound to texture object zero.

8.2 Sampler Objects

The state necessary for texturing can be divided into two categories as described in section 8.22. A GL texture object includes both categories. The first category represents dimensionality and other image parameters, and the second category represents sampling state. Additionally, a sampler object may be created to encapsulate only the sampling state of a texture object.

A new sampler object is created by binding an unused name to a texture unit. The command

`void` **GenSamplers**(`sizei` *count*, `uint` **samplers*);

returns *count* previously unused sampler object names in *samplers*. The name zero is reserved by the GL to represent no sampler being bound to a sampler unit. The names are marked as used, for the purposes of **GenSamplers** only, but they acquire state only when they are first used as a parameter to **BindSampler**, **SamplerParameter***, **GetSamplerParameter***, or **IsSampler**. When a sampler object is first used in one of these functions, the resulting sampler object is initialized with a new state vector, comprising all the state and with the same initial values listed in table 23.23.

Errors

An `INVALID_VALUE` error is generated if *count* is negative.

Sampler objects may also be created with the command

`void` **CreateSamplers**(`sizei` *n*, `uint` **samplers*);

CreateSamplers returns *n* previously unused sampler names in *samplers*, each representing a new sampler object which is a state vector comprising all the state and with the same initial values listed in table 23.23[1].

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

When a sampler object is bound to a texture unit, its state supersedes that of the texture object bound to that texture unit. If the sampler name zero is bound to a texture unit, the currently bound texture's sampler state becomes active. A single sampler object may be bound to multiple texture units simultaneously.

A sampler object binding is effected with the command

`void` **BindSampler**(`uint` *unit*, `uint` *sampler*);

with *unit* set to the zero-based index of the texture unit to which to bind the sampler and *sampler* set to the name of a sampler object returned from a previous call to **GenSamplers**.

If the bind is successful no change is made to the state of the bound sampler object, and any previous binding to *unit* is broken.

If state is present in a sampler object bound to a texture unit that would have been rejected by a call to **TexParameter*** for the texture bound to that unit, the behavior of the implementation is as if the texture were incomplete. For example, if `TEXTURE_WRAP_S` or `TEXTURE_WRAP_T` is set to `REPEAT`, `MIRRORED_REPEAT`, or `MIRROR_CLAMP_TO_EDGE` on the sampler object bound to a texture unit and the texture bound to that unit is a rectangle texture, the texture will be considered incomplete.

Sampler object state which does not affect sampling for the type of texture bound to a texture unit, such as `TEXTURE_WRAP_R` for a rectangle texture, does not affect completeness.

The currently bound sampler may be queried by calling **GetIntegerv** with *pname* set to `SAMPLER_BINDING`. When a sampler object is unbound from the texture unit (by binding the sampler object named zero to that unit), the modified state is again replaced with the sampler state associated with the texture object bound to that texture unit.

[1] Note that unlike texture objects, the initial sampler object state for `TEXTURE_MIN_FILTER` and `TEXTURE_WRAP_`$\star$ are fixed, rather than dependent on the type of texture image.

Errors

An INVALID_VALUE error is generated if *unit* is greater than or equal to the value of MAX_COMBINED_TEXTURE_IMAGE_UNITS.

An INVALID_OPERATION error is generated if *sampler* is not zero or a name returned from a previous call to **GenSamplers**, or if such a name has since been deleted with **DeleteSamplers**.

The command

```
void BindSamplers( uint first, sizei count, const
   uint *samplers );
```

binds *count* existing sampler objects to texture image units numbered *first* through $first + count - 1$. If *samplers* is not NULL, it specifies an array of *count* values, each of which must be zero or the name of an existing sampler object. If *samplers* is NULL, each affected texture image unit from *first* through $first + count - 1$ will be reset to have no bound sampler object.

BindSamplers is equivalent (assuming no errors are generated to):

```
for (i = 0; i < count; i++) {
   if (samplers == NULL) {
      BindSampler(first + i, 0);
   } else {
      BindSampler(first + i, samplers[i]);
   }
}
```

The values specified in *samplers* will be checked separately for each texture image unit. When a value for a specific texture image unit is invalid, the state for that texture image unit will be unchanged and an error will be generated. However, state for other texture image units will still be changed if their corresponding values are valid.

Errors

An INVALID_OPERATION error is generated if $first + count$ is greater than the number of texture image units supported by the implementation.

An INVALID_VALUE error is generated if *count* is negative.

An INVALID_OPERATION error is generated if any value in *samplers* is

not zero or the name of an existing sampler object (per binding).

The parameters represented by a sampler object are a subset of those described in section 8.10. Each parameter of a sampler object is set by calling

```
void SamplerParameter{if}( uint sampler, enum pname,
    T param );
void SamplerParameter{if}v( uint sampler, enum pname,
    const T *param );
void SamplerParameterI{i ui}v( uint sampler, enum pname,
    const T *params );
```

sampler is the name of a sampler object previously reserved by a call to **GenSamplers**. *pname* is the name of a parameter to modify and *param* is the new value of that parameter. *pname* must be one of the sampler state names in table 23.23.

Texture state listed in tables 23.21- 23.22 but not listed here and in the sampler state in table 23.23 is not part of the sampler state, and remains in the texture object.

Data conversions are performed as specified in section 2.2.1, with these exceptions:

- If the values for TEXTURE_BORDER_COLOR are specified with **SamplerParameterIiv** or **SamplerParameterIuiv**, they are unmodified and stored with an internal data type of integer. If specified with **SamplerParameteriv**, they are converted to floating-point using equation 2.2. Otherwise, the values are unmodified and stored as floating-point.

Modifying a parameter of a sampler object affects all texture units to which that sampler object is bound. Calling **TexParameter** has no effect on the sampler object bound to the active texture unit. It will modify the parameters of the texture object bound to that unit.

Errors

An INVALID_OPERATION error is generated if *sampler* is not the name of a sampler object previously returned from a call to **GenSamplers**.

An INVALID_ENUM error is generated if *pname* is not one of the sampler state names in table 23.23.

An INVALID_ENUM error is generated if **SamplerParameter{if}** is called for a non-scalar parameter (*pname* TEXTURE_BORDER_COLOR or TEXTURE_SWIZZLE_RGBA).

If the value of *param* is not an acceptable value for the parameter specified

in *pname*, an error is generated as specified in the description of **TexParameter***.

Sampler objects are deleted by calling

```
void DeleteSamplers( sizei count, const uint *samplers );
```

samplers contains *count* names of sampler objects to be deleted. After a sampler object is deleted, its name is again unused. If a sampler object that is currently bound to one or more texture units is deleted, it is as though **BindSampler** is called once for each texture unit to which the sampler is bound, with *unit* set to the texture unit and *sampler* set to zero. Unused names in *samplers* that have been marked as used for the purposes of **GenSamplers** are marked as unused again. Unused names in *samplers* are silently ignored, as is the reserved name zero.

Errors

An `INVALID_VALUE` error is generated if *count* is negative.

The command

```
boolean IsSampler( uint sampler );
```

may be called to determine whether *sampler* is the name of a sampler object. **IsSampler** will return `TRUE` if *sampler* is the name of a sampler object previously returned from a call to **GenSamplers** and `FALSE` otherwise. Zero is not the name of a sampler object.

8.3 Sampler Object Queries

The current values of the parameters of a sampler object may be queried by calling

```
void GetSamplerParameter{if}v( uint sampler,
    enum pname, T *params );
void GetSamplerParameterI{i ui}v( uint sampler,
    enum pname, T *params );
```

sampler is the name of the sampler object from which to retrieve parameters. *pname* is the name of the parameter to be queried, and must be one of the sampler state names in table 23.23. *params* is the address of an array into which the current value of the parameter will be placed.

Querying TEXTURE_BORDER_COLOR with **GetSamplerParameterIiv** or **GetSamplerParameterIuiv** returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 2.2.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.

Errors

An INVALID_OPERATION error is generated if *sampler* is not the name of a sampler object previously returned from a call to **GenSamplers**.

An INVALID_ENUM error is generated if *pname* is not one of the sampler state names in table 23.23.

8.4 Pixel Rectangles

Rectangles of color, depth, and certain other values may be specified to the GL using **TexImage*D** (see section 8.5) or converted to fragments using the **DrawPixels** command (described in section 18.1) Some of the parameters and operations governing the operation of these commands are shared by **CopyPixels** (used to copy pixels from one framebuffer location to another) and **ReadPixels** (used to obtain pixel values from the framebuffer); the discussion of **CopyPixels** and **ReadPixels**, however, is deferred until chapter 9 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to these commands also pertain to **CopyPixels** or **ReadPixels**.

A number of parameters control the encoding of pixels in buffer object or client memory (for reading and writing) and how pixels are processed before being placed in or after being read from the framebuffer (for reading, writing, and copying). These parameters are set with the commands **PixelStore***, **PixelTransfer***, and **PixelMap***.

8.4.1 Pixel Storage Modes and Pixel Buffer Objects

Pixel storage modes affect the operation of **TexImage*D**, **TexSubImage*D**, **CompressedTexImage*D**, **CompressedTexSubImage*D**, **DrawPixels**, and **ReadPixels** (as well as other commands; see sections 14.6.2 and 14.8) when one of these commands is issued. This may differ from the time that the command is executed if the command is placed in a display list (see section 21.4). Pixel storage modes are set with

`void` **PixelStore**{**if**}(`enum` *pname*, `T` *param*);

Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
UNPACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
UNPACK_ROW_LENGTH	integer	0	$[0, \infty)$
UNPACK_SKIP_ROWS	integer	0	$[0, \infty)$
UNPACK_SKIP_PIXELS	integer	0	$[0, \infty)$
UNPACK_ALIGNMENT	integer	4	1,2,4,8
UNPACK_IMAGE_HEIGHT	integer	0	$[0, \infty)$
UNPACK_SKIP_IMAGES	integer	0	$[0, \infty)$
UNPACK_COMPRESSED_BLOCK_WIDTH	integer	0	$[0, \infty)$
UNPACK_COMPRESSED_BLOCK_HEIGHT	integer	0	$[0, \infty)$
UNPACK_COMPRESSED_BLOCK_DEPTH	integer	0	$[0, \infty)$
UNPACK_COMPRESSED_BLOCK_SIZE	integer	0	$[0, \infty)$

Table 8.1: **PixelStore*** parameters pertaining to one or more of **DrawPixels**, **ColorTable**, **ColorSubTable**, **ConvolutionFilter1D**, **ConvolutionFilter2D**, **SeparableFilter2D**, **PolygonStipple**, **TexImage*D**, **TexSubImage*D**, **CompressedTexImage*D** and **CompressedTexSubImage*D**.

pname is a symbolic constant indicating a parameter to be set, and *param* is the value to set it to. Tables 8.1 and 18.1 summarize the pixel storage parameters, their types, their initial values, and their allowable ranges.

Errors

An INVALID_ENUM error is generated if *pname* is not one of the paramater names in table 8.1 or 18.1.

An INVALID_VALUE error is generated if *param* is outside the given range for the corresponding *pname* in table 8.1 or 18.1.

Data conversions are performed as specified in section 2.2.1.

In addition to storing pixel data in client memory, pixel data may also be stored in buffer objects (described in section 6). The current pixel unpack and pack buffer objects are designated by the PIXEL_UNPACK_BUFFER and PIXEL_PACK_BUFFER targets respectively.

Initially, zero is bound for the PIXEL_UNPACK_BUFFER, indicating that image specification commands such as **DrawPixels** source their pixels from client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel unpack buffer, then the pointer parameter is treated as an offset into

the designated buffer object.

8.4.2 The Imaging Subset

Some pixel transfer operations are only made available in GL implementations which incorporate the optional *imaging subset*. The imaging subset includes both new commands, and new enumerants allowed as parameters to existing commands. If the subset is supported, *all* of these calls and enumerants must be implemented as described later in this section.

Errors

If the imaging subset is not supported, calling any unsupported command generates an `INVALID_OPERATION` error, and using any of the new enumerants generates an `INVALID_ENUM` error.

The individual operations available only in the imaging subset are described in section 8.4.3. Imaging subset operations include:

- Color tables, including all commands and enumerants described in subsections **Color Table Specification**, **Alternate Color Table Specification Commands**, **Color Table State and Proxy State**, **Color Table Lookup**, **Post Convolution Color Table Lookup**, and **Post Color Matrix Color Table Lookup**, as well as the query commands described in section 8.4.3.4.
- Convolution, including all commands and enumerants described in subsections **Convolution Filter Specification**, **Alternate Convolution Filter Specification Commands**, and **Convolution**, as well as the query commands described in section 8.4.3.8.
- Color matrix, including all commands and enumerants described in subsections **Color Matrix Specification** and **Color Matrix Transformation**, as well as the simple query commands described in section 8.4.3.11.
- Histogram and minmax, including all commands and enumerants described in subsections **Histogram Table Specification**, **Histogram State and Proxy State**, **Histogram**, **Minmax Table Specification**, and **Minmax**, as well as the query commands described in section 8.4.3.13 and section 8.4.3.16.

The imaging subset is supported only if the `EXTENSIONS` string includes the substring "`GL_ARB_imaging`" Querying `EXTENSIONS` is described in section 22.2.

Parameter Name	Type	Initial Value	Valid Range
MAP_COLOR	boolean	FALSE	TRUE/FALSE
MAP_STENCIL	boolean	FALSE	TRUE/FALSE
INDEX_SHIFT	integer	0	$(-\infty, \infty)$
INDEX_OFFSET	integer	0	$(-\infty, \infty)$
x_SCALE	float	1.0	$(-\infty, \infty)$
DEPTH_SCALE	float	1.0	$(-\infty, \infty)$
x_BIAS	float	0.0	$(-\infty, \infty)$
DEPTH_BIAS	float	0.0	$(-\infty, \infty)$
POST_CONVOLUTION_x_SCALE	float	1.0	$(-\infty, \infty)$
POST_CONVOLUTION_x_BIAS	float	0.0	$(-\infty, \infty)$
POST_COLOR_MATRIX_x_SCALE	float	1.0	$(-\infty, \infty)$
POST_COLOR_MATRIX_x_BIAS	float	0.0	$(-\infty, \infty)$

Table 8.2: **PixelTransfer** parameters. x is RED, GREEN, BLUE, or ALPHA.

If the imaging subset is not supported, the related pixel transfer operations are not performed; pixels are passed unchanged to the next operation.

8.4.3 Pixel Transfer Modes

Pixel transfer modes affect the operation of **DrawPixels** (section 18.1), **ReadPixels** (section 18.2), and **CopyPixels** (section 18.3) at the time when one of these commands is executed (which may differ from the time the command is issued). Some pixel transfer modes are set with

`void` **PixelTransfer**{**if**}(`enum` *param*, `T` *value*);

param is a symbolic constant indicating a parameter to be set, and *value* is the value to set it to. Table 8.2 summarizes the pixel transfer parameters that are set with **PixelTransfer**, their types, their initial values, and their allowable ranges.

Data conversions are performed as specified in section 2.2.1.

Errors

An INVALID_ENUM error is generated if *param* is not one of the parameter names in table 8.2.

Map Name	Address	Value	Init. Size	Init. Value
PIXEL_MAP_I_TO_I	color idx	color idx	1	0.0
PIXEL_MAP_S_TO_S	stencil idx	stencil idx	1	0
PIXEL_MAP_I_TO_R	color idx	R	1	0.0
PIXEL_MAP_I_TO_G	color idx	G	1	0.0
PIXEL_MAP_I_TO_B	color idx	B	1	0.0
PIXEL_MAP_I_TO_A	color idx	A	1	0.0
PIXEL_MAP_R_TO_R	R	R	1	0.0
PIXEL_MAP_G_TO_G	G	G	1	0.0
PIXEL_MAP_B_TO_B	B	B	1	0.0
PIXEL_MAP_A_TO_A	A	A	1	0.0

Table 8.3: **PixelMap** parameters.

An INVALID_VALUE error is generated if *value* is outside the range given for *param* in table 8.2.

The pixel map lookup tables are set with

```
void PixelMap{ui us f}v( enum map, sizei size, const
    T *values );
```

map is a symbolic map name, indicating the map to set, *size* indicates the size of the map, and *values* refers to an array of *size* map values.

The entries of a table may be specified using one of three types: single-precision floating-point, unsigned short integer, or unsigned integer, depending on which of the three versions of **PixelMap** is called. A table entry is converted to the appropriate type when it is specified. An entry giving a color component value is converted as described in equation 2.1 and then clamped to the range $[0, 1]$. An entry giving a color index value is converted from an unsigned short integer or unsigned integer to floating-point. An entry giving a stencil index is converted from single-precision floating-point to an integer by rounding to nearest. The various tables and their initial sizes and entries are summarized in table 8.3.

The maximum allowable *size* of each table is specified by the implementation-dependent value of MAX_PIXEL_MAP_TABLE, but must be at least 32 (a single maximum applies to all tables).

If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING), *values* is an offset into the pixel unpack buffer; otherwise, *values* is a pointer to client memory. All pixel storage and pixel transfer

modes are ignored when specifying a pixel map. n machine units are read where n is the *size* of the pixel map times the size of a `float`, `uint`, or `ushort` datum in basic machine units, depending on the respective **PixelMap** version.

Errors

An `INVALID_VALUE` error is generated if *size* is negative.

An `INVALID_ENUM` error is generated if *map* is not one of the map names in table 8.3.

An `INVALID_VALUE` error is generated if *size* is less than one or larger than the value of `MAX_PIXEL_MAP_TABLE`.

An `INVALID_VALUE` error is generated if *map* is `INDEX_SHIFT` or `INDEX_OFFSET` and *size* is not a power of two.

An `INVALID_OPERATION` error is generated if a pixel unpack buffer object is bound and $data + n$ is greater than the size of the pixel buffer.

An `INVALID_OPERATION` error is generated if a pixel unpack buffer object is bound and *values* is not evenly divisible by the number of basic machine units needed to store in memory a `float`, `uint`, or `ushort` datum depending on their respective **PixelMap** version.

8.4.3.1 Pixel Map Queries

The commands

```
void GetPixelMap{ui us f}v( enum map, T *data );
void GetnPixelMap{ui us f}v( enum map, sizei bufSize,
    T data );
```

return all values in the pixel map *map* in *data*. *map* must be a map name from table 8.3. **Get*PixelMapuiv** and **Get*PixelMapusv** convert floating-point pixel map values to integers according to the `UNSIGNED_INT` and `UNSIGNED_SHORT` entries, respectively, of table 18.2.

GetnPixelMap* do not write more than *bufSize* bytes into *data*.

If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), *data* is an offset into the pixel pack buffer; otherwise, *data* is a pointer to client memory. All pixel storage and pixel transfer modes are ignored when returning a pixel map. n machine units are written where n is the size of the pixel map times the size of `FLOAT`, `UNSIGNED_INT`, or `UNSIGNED_SHORT` respectively in basic machine units.

Errors

An INVALID_OPERATION error is generated if a pixel pack buffer object is bound and $data + n$ is greater than the size of the pixel buffer.

An INVALID_OPERATION error is generated if a pixel pack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory a FLOAT, UNSIGNED_INT, or UNSIGNED_SHORT respectively.

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetnPixelMap*** if the buffer size required to store the requested data is greater than *bufSize*.

The same errors generated by **PixelMap*v** for invalid values of *map* are generated by **GetPixelMap***.

8.4.3.2 Color Table Specification

Color lookup tables are specified with

```
void ColorTable( enum target, enum internalformat,
    sizei width, enum format, enum type, const
    void *data );
```

target must be one of the *regular* color table names listed in table 8.4 to define the table. A *proxy* table name is a special case discussed later in this section. *width*, *format*, *type*, and *data* specify an image in memory with the same meaning and allowed values as the corresponding arguments to **DrawPixels** (see section 18.1), with *height* taken to be 1. The maximum allowable *width* of a table is implementation-dependent, but must be at least 32. The *format*s COLOR_INDEX, DEPTH_COMPONENT, DEPTH_STENCIL, and STENCIL_INDEX and the *type* BITMAP are not allowed.

The specified image is taken from memory and processed just as if **DrawPixels** were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four COLOR_TABLE_SCALE parameters and biased by the four COLOR_TABLE_BIAS parameters. These parameters are set by calling **ColorTableParameterfv** as described below. If fragment color clamping is enabled or *internalformat* is fixed-point, components are clamped to $[0, 1]$. Otherwise, components are not modified.

Components are then selected from the resulting R, G, B, and A values to obtain a table with the *base internal format* specified by (or derived from) *internalformat*, in the same manner as for textures (section 8.5). *internalformat* must

Table Name	Type
COLOR_TABLE POST_CONVOLUTION_COLOR_TABLE POST_COLOR_MATRIX_COLOR_TABLE	regular
PROXY_COLOR_TABLE PROXY_POST_CONVOLUTION_COLOR_TABLE PROXY_POST_COLOR_MATRIX_COLOR_TABLE	proxy

Table 8.4: Color table names. Regular tables have associated image data. Proxy tables have no image data, and are used only to determine if an image can be loaded into the corresponding regular table.

be one of the formats in table 8.18 or tables 8.19- 8.21, with the exception of the RED, RG, DEPTH_COMPONENT, and DEPTH_STENCIL base and sized internal formats in those tables, all sized internal formats with non-fixed internal data types (see section 8), and sized internal format RGB9_E5.

The color lookup table is redefined to have *width* entries, each with the specified internal format. The table is formed with indices 0 through $width - 1$. Table location i is specified by the ith image pixel, counting from zero.

Errors

An INVALID_ENUM error is generated if *target* is not a regular color table name from table 8.4.

An INVALID_VALUE error is generated if *width* is not zero or a non-negative power of two.

A TABLE_TOO_LARGE error is generated if the specified color lookup table is too large for the implementation.

The scale and bias parameters for a table are specified by calling

```
void ColorTableParameter{if}v( enum target, enum pname,
    const T *params );
```

target must be a regular color table name. *pname* is one of COLOR_TABLE_SCALE or COLOR_TABLE_BIAS. *params* points to an array of four values: red, green, blue, and alpha, in that order.

Errors

An INVALID_ENUM error is generated if *target* is not a regular color table name from table 8.4.

An INVALID_ENUM error is generated if *pname* is not COLOR_TABLE_SCALE or COLOR_TABLE_BIAS.

Data conversions are performed as specified in section 2.2.1.

A GL implementation may vary its allocation of internal component resolution based on any **ColorTable** parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. Allocations must be invariant; the same allocation must be made each time a color table is specified with the same parameter values. These allocation rules also apply to proxy color tables, which are described later in this section.

8.4.3.3 Alternate Color Table Specification Commands

Color tables may also be specified using image data taken directly from the framebuffer, and portions of existing tables may be respecified.

The command

```
void CopyColorTable( enum target, enum internalformat,
    int x, int y, sizei width );
```

defines a color table in exactly the manner of **ColorTable**, except that table data are taken from the framebuffer, rather than from client memory. *target* must be a regular color table name. x, y, and *width* correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 18.3); they specify the image's *width* and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to COLOR and *height* set to 1, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for **ColorTable**, beginning with scaling by COLOR_TABLE_SCALE. Parameters *target*, *internalformat* and *width* are specified using the same values, with the same meanings, as the corresponding arguments of **ColorTable**. *format* is taken to be RGBA.

Errors

An INVALID_ENUM error is generated if *target* is not a regular color table name from table 8.4.

An INVALID_VALUE error is generated if *width* is negative.

An INVALID_FRAMEBUFFER_OPERATION error is generated if the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer complete (see section 9.4.2).

Two additional commands,

```
void ColorSubTable( enum target, sizei start, sizei count,
    enum format, enum type, const void *data );
void CopyColorSubTable( enum target, sizei start, int x,
    int y, sizei count );
```

respecify only a portion of an existing color table. No change is made to the *internalformat* or *width* parameters of the specified color table, nor is any change made to table entries outside the specified portion. *target* must be a regular color table name.

ColorSubTable arguments *format*, *type*, and *data* match the corresponding arguments to **ColorTable**, meaning that they are specified using the same values, and have the same meanings. Likewise, **CopyColorSubTable** arguments *x*, *y*, and *count* match the *x*, *y*, and *width* arguments of **CopyColorTable**. Both of the **ColorSubTable** commands interpret and process pixel groups in exactly the manner of their **ColorTable** counterparts, except that the assignment of R, G, B, and A pixel group values to the color table components is controlled by the *internalformat* of the table, not by an argument to the command.

Arguments *start* and *count* of **ColorSubTable** and **CopyColorSubTable** specify a subregion of the color table starting at index *start* and ending at index $start + count - 1$. Counting from zero, the nth pixel group is assigned to the table entry with index $count + n$.

Errors

An INVALID_ENUM error is generated if *target* is not a regular color table name from table 8.4.

An INVALID_FRAMEBUFFER_OPERATION error is generated by **CopyColorSubTable** if the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer complete (see section 9.4.2).

An INVALID_VALUE error is generated if *start* or *count* is negative, or if $start + count$ is greater than the width of the color table specified by *target*.

8.4.3.4 Color Table Query

The current contents of a color table are queried with the commands

format Name
RED
GREEN
BLUE
ALPHA
RGB
RGBA
BGR
BGRA
LUMINANCE
LUMINANCE_ALPHA

Table 8.5: Pixel data *format* parameter values accepted for the color table, convolution filter, histogram table, and minmax table query commands. These commands accept only a subset of the formats accepted by **GetTexImage**, but the specification and interpretation of pixels in those formats is identical to that described for the same formats in table 8.8.

```
void GetColorTable( enum target, enum format, enum type,
    void *table );
void GetnColorTable( enum target, enum format, enum type,
    sizei bufSize, void *table );
```

target must be one of the *regular* color table names listed in table 8.4. *format* must be a pixel format from table 8.5 and *type* must be a data type from table 8.6. The one-dimensional color table image is returned to pixel pack buffer or client memory starting at *table*. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **ReadPixels** are performed. Color components that are requested in the specified *format*, but which are not included in the internal format of the color lookup table, are returned as zero. The assignments of internal color components to the components requested by *format* are described in table 8.26.

Errors

An INVALID_ENUM error is generated if *target* is not one of the *regular* color table names from table 8.4.

An INVALID_ENUM error is generated if *format* is not one of the pixel

type Name
UNSIGNED_BYTE
BYTE
UNSIGNED_SHORT
SHORT
UNSIGNED_INT
INT
UNSIGNED_BYTE_3_3_2
UNSIGNED_BYTE_2_3_3_REV
UNSIGNED_SHORT_5_6_5
UNSIGNED_SHORT_5_6_5_REV
UNSIGNED_SHORT_4_4_4_4
UNSIGNED_SHORT_4_4_4_4_REV
UNSIGNED_SHORT_5_5_5_1
UNSIGNED_SHORT_1_5_5_5_REV
UNSIGNED_INT_8_8_8_8
UNSIGNED_INT_8_8_8_8_REV
UNSIGNED_INT_10_10_10_2
UNSIGNED_INT_2_10_10_10_REV

Table 8.6: Pixel data *type* parameter values accepted for the color table, convolution filter, histogram table, and minmax table query commands. These commands accept only a subset of the types accepted by **GetTexImage**, but the specification and interpretation of pixels in those types is identical to that described for the same types in table 8.7.

formats from table 8.5.

An INVALID_ENUM error is generated if *type* is not a data type from table 8.6.

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetnColorTable** if the buffer size required to store the requested data is greater than *bufSize*.

The command

```
void GetColorTableParameter{if}v( enum target,
    enum pname, T params );
```

is used for integer and floating-point query.

target must be one of the regular or proxy color table names listed in table 8.4. *pname* is one of COLOR_TABLE_SCALE, COLOR_TABLE_BIAS, COLOR_TABLE_FORMAT, COLOR_TABLE_WIDTH, COLOR_TABLE_RED_SIZE, COLOR_TABLE_GREEN_SIZE, COLOR_TABLE_BLUE_SIZE, COLOR_TABLE_ALPHA_SIZE, COLOR_TABLE_LUMINANCE_SIZE, or COLOR_TABLE_INTENSITY_SIZE. The value of the specified parameter is returned in *params*.

Errors

An INVALID_ENUM error is generated if *target* is not one of the regular or proxy color table names from table 8.4.

An INVALID_ENUM error is generated if *pname* is not one of the valid parameter names described above.

8.4.3.5 Color Table State and Proxy State

The state necessary for color tables can be divided into two categories. For each of the three tables, there is an array of values. Each array has associated with it a width, an integer describing the internal format of the table, six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the table, and two groups of four floating-point numbers to store the table scale and bias. Each initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the scale parameters is (1,1,1,1) and the initial value of the bias parameters is (0,0,0,0).

In addition to the color lookup tables, partially instantiated proxy color lookup tables are maintained. Each proxy table includes width and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions. Proxy tables do not include image data, nor do they include

scale and bias parameters. When **ColorTable** is executed with *target* specified as one of the proxy color table names listed in table 8.4, the proxy state values of the table are recomputed and updated. If the table is too large, no error is generated, but the proxy format, width and component resolutions are set to zero. If the color table would be accommodated by **ColorTable** called with *target* set to the corresponding regular table name (`COLOR_TABLE` is the regular name corresponding to `PROXY_COLOR_TABLE`, for example), the proxy state values are set exactly as though the regular table were being specified. Calling **ColorTable** with a proxy *target* has no effect on the image or state of any actual color table.

There is no image associated with any of the proxy targets. They cannot be used as color tables, and cannot be queried using **GetColorTable**.

8.4.3.6 Convolution Filter Specification

A two-dimensional convolution filter image is specified by calling

```
void ConvolutionFilter2D( enum target, enum internalformat,
    sizei width, sizei height, enum format, enum type,
    const void *data );
```

target must be `CONVOLUTION_2D`. *width*, *height*, *format*, *type*, and *data* specify an image in memory with the same meaning and allowed values as the corresponding parameters to **DrawPixels**. The *format*s `COLOR_INDEX`, `DEPTH_COMPONENT`, `DEPTH_STENCIL`, and `STENCIL_INDEX` and the *type* `BITMAP` are not allowed.

The specified image is extracted from memory and processed just as if **DrawPixels** were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four two-dimensional `CONVOLUTION_FILTER_SCALE` parameters and biased by the four two-dimensional `CONVOLUTION_FILTER_BIAS` parameters. These parameters are set by calling **ConvolutionParameterfv** as described below. No clamping takes place at any time during this process.

Components are then selected from the resulting R, G, B, and A values to obtain a table with the *base internal format* specified by (or derived from) *internalformat*, in the same manner as for textures (section 8.5). *internalformat* accepts the same values as the corresponding argument of **ColorTable**.

The red, green, blue, alpha, luminance, and/or intensity components of the pixels are stored in floating-point, rather than integer format. They form a two-dimensional image indexed with coordinates i, j such that i increases from left to right, starting at zero, and j increases from bottom to top, also starting at zero. Image location i, j is specified by the Nth pixel, counting from zero, where

$$N = i + j * width$$

Errors

An INVALID_VALUE error is generated if *width* or *height* is negative, or greater than the maximum supported width or height, which may be queried by calling **GetConvolutionParameteriv** with *target* CONVOLUTION_2D and *pname* MAX_CONVOLUTION_WIDTH or MAX_CONVOLUTION_HEIGHT, respectively.

The scale and bias parameters for a two-dimensional filter are specified by calling

```
void ConvolutionParameter{if}v( enum target, enum pname,
    const T *params );
```

with *target* CONVOLUTION_2D. *pname* is one of CONVOLUTION_FILTER_SCALE or CONVOLUTION_FILTER_BIAS. *params* points to an array of four values: red, green, blue, and alpha, in that order.

Data conversions are performed as specified in section 2.2.1.

A one-dimensional convolution filter is defined using

```
void ConvolutionFilter1D( enum target, enum internalformat,
    sizei width, enum format, enum type, const
    void *data );
```

target must be CONVOLUTION_1D. *internalformat*, *width*, *format*, and *type* have identical semantics and accept the same values as do their two-dimensional counterparts. *data* must point to a one-dimensional image, however.

The image is extracted from memory and processed as if **ConvolutionFilter2D** were called with a *height* of 1, except that it is scaled and biased by the one-dimensional CONVOLUTION_FILTER_SCALE and CONVOLUTION_FILTER_BIAS parameters. These parameters are specified exactly as the two-dimensional parameters, except that **ConvolutionParameterfv** is called with *target* CONVOLUTION_1D.

The image is formed with coordinates i such that i increases from left to right, starting at zero. Image location i is specified by the ith pixel, counting from zero.

Errors

An INVALID_VALUE error is generated if *width* is negative, or greater than the maximum supported value width, which may be queried by calling **GetConvolutionParameteriv** with *target* CONVOLUTION_1D and *pname* MAX_-

CONVOLUTION_WIDTH.

Special facilities are provided for the definition of two-dimensional *separable filters* – filters whose image can be represented as the product of two one-dimensional images, rather than as full two-dimensional images. A two-dimensional separable convolution filter is specified with

```
void SeparableFilter2D( enum target, enum internalformat,
    sizei width, sizei height, enum format, enum type,
    const void *row, const void *column );
```

target must be SEPARABLE_2D. *internalformat* specifies the formats of the table entries of the two one-dimensional images that will be retained. *row* points to a *width* pixel wide image of the specified *format* and *type*. *column* points to a *height* pixel high image, also of the specified *format* and *type*.

The two images are extracted from memory and processed as if **ConvolutionFilter1D** were called separately for each, except that each image is scaled and biased by the two-dimensional separable CONVOLUTION_FILTER_SCALE and CONVOLUTION_FILTER_BIAS parameters. These parameters are specified exactly as the one-dimensional and two-dimensional parameters, except that **ConvolutionParameteriv** is called with *target* SEPARABLE_2D.

Errors

An INVALID_VALUE error is generated if *width* or *height* is negative.

8.4.3.7 Alternate Convolution Filter Specification Commands

One and two-dimensional filters may also be specified using image data taken directly from the framebuffer.

The command

```
void CopyConvolutionFilter2D( enum target,
    enum internalformat, int x, int y, sizei width,
    sizei height );
```

defines a two-dimensional filter in exactly the manner of **ConvolutionFilter2D**, except that image data are taken from the framebuffer, rather than from client memory. *target* must be CONVOLUTION_2D. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 18.3); they specify the image's *width* and *height*, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as

if these arguments were passed to **CopyPixels** with argument *type* set to COLOR, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for **ConvolutionFilter2D**, beginning with scaling by CONVOLUTION_FILTER_SCALE. Parameters *target*, *internalformat*, *width*, and *height* are specified using the same values, with the same meanings, as the corresponding arguments of **ConvolutionFilter2D**. *format* is taken to be RGBA.

Errors

An INVALID_VALUE error is generated if *width* or *height* is negative.

An INVALID_FRAMEBUFFER_OPERATION error is generated if the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer complete (see section 9.4.2).

The command

```
void CopyConvolutionFilter1D( enum target,
    enum internalformat, int x, int y, sizei width );
```

defines a one-dimensional filter in exactly the manner of **ConvolutionFilter1D**, except that image data are taken from the framebuffer, rather than from client memory. *target* must be CONVOLUTION_1D. *x*, *y*, and *width* correspond precisely to the corresponding arguments of **CopyPixels** (see section 18.3); they specify the image's *width* and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to COLOR and *height* set to 1, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for **ConvolutionFilter1D**, beginning with scaling by CONVOLUTION_FILTER_SCALE. Parameters *target*, *internalformat*, and *width* are specified using the same values, with the same meanings, as the corresponding arguments of **ConvolutionFilter2D**. *format* is taken to be RGBA.

Errors

An INVALID_VALUE error is generated if *width* is negative.

An INVALID_FRAMEBUFFER_OPERATION error is generated if the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer complete (see section 9.4.2).

8.4.3.8 Convolution Query

The contents of a convolution filter image are queried with the commands

```
void GetConvolutionFilter( enum target, enum format,
    enum type, void *image );
void GetnConvolutionFilter( enum target, enum format,
    enum type, sizei bufSize, void *image );
```

target must be CONVOLUTION_1D or CONVOLUTION_2D. *format* must be a pixel format from table 8.5 and *type* must be a data type from table 8.6. The one- or two-dimensional image is returned to pixel pack buffer or client memory starting at *image*. Pixel processing and component mapping are identical to those of **GetTexImage**.

Errors

An INVALID_ENUM error is generated if *target* is not CONVOLUTION_1D or CONVOLUTION_2D.

An INVALID_ENUM error is generated if *format* is not a pixel format from table 8.5.

An INVALID_ENUM error is generated if *type* is not a data type from table 8.6.

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetnConvolutionFilter** if the buffer size required to store the requested data is greater than *bufSize*.

The contents of a separable filter image are queried with the commands

```
void GetSeparableFilter( enum target, enum format,
    enum type, void *row, void *column, void *span );
void GetnSeparableFilter( enum target, enum format,
    enum type, sizei rowBufSize, void *row,
    sizei columnBufSize, void *column, void *span );
```

target must be SEPARABLE_2D. *format* must be a pixel format from table 8.5 and *type* must be a data type from table 8.6. The row and column images are returned to pixel pack buffer or client memory starting at *row* and *column* respectively. *span* is unused. Pixel processing and component mapping are identical to those of **GetTexImage**.

Errors

An INVALID_ENUM error is generated if *target* is not SEPARABLE_2D.

An INVALID_ENUM error is generated if *format* is not a pixel format from table 8.5.

An INVALID_ENUM error is generated if *type* is not a data type from table 8.6.

An INVALID_VALUE error is generated if *rowBufSize* or *columnBufSize* is negative.

An INVALID_OPERATION error is generated by **GetnSeparableFilter** if the buffer size required to store the requested row and column data is greater than *rowBufSize* or *columnBufSize* respectively.

The commands

```
void GetConvolutionParameter{if}v( enum target,
    enum pname, T *params );
```

are used for integer and floating-point query. *target* must be CONVOLUTION_1D, CONVOLUTION_2D, or SEPARABLE_2D. *pname* is one of CONVOLUTION_BORDER_COLOR, CONVOLUTION_BORDER_MODE, CONVOLUTION_FILTER_SCALE, CONVOLUTION_FILTER_BIAS, CONVOLUTION_FORMAT, CONVOLUTION_WIDTH, CONVOLUTION_HEIGHT, MAX_CONVOLUTION_WIDTH, or MAX_CONVOLUTION_HEIGHT. The value of the specified parameter is returned in *params*.

8.4.3.9 Convolution Filter State

The required state for convolution filters includes a one-dimensional filter image, two one-dimensional filter images, for the separable filter, and a two-dimensional filter image. Each filter has associated with it a width and height (two-dimensional and separable only), an integer describing the internal format of the filter, and two groups of four floating-point numbers to store the filter scale and bias.

Each initial convolution filter is null (zero width and height, internal format RGBA, with zero-sized components). The initial value of all scale parameters is (1,1,1,1) and the initial value of all bias parameters is (0,0,0,0).

8.4.3.10 Color Matrix Specification

Setting the matrix mode to COLOR causes the matrix operations described in section 12.1.1 to apply to the top matrix on the color matrix stack. All matrix operations have the same effect on the color matrix as they do on the other matrices.

8.4.3.11 Color Matrix Query

The scale and bias variables are queried using **GetFloatv** with *pname* set to the appropriate variable name. The top matrix on the color matrix stack is returned by **GetFloatv** called with *pname* set to COLOR_MATRIX or TRANSPOSE_COLOR_MATRIX. The depth of the color matrix stack, and the maximum depth of the color matrix stack, are queried with **GetIntegerv**, setting *pname* to COLOR_MATRIX_STACK_DEPTH and MAX_COLOR_MATRIX_STACK_DEPTH respectively.

8.4.3.12 Histogram Table Specification

The histogram table is specified with

```
void Histogram( enum target, sizei width,
    enum internalformat, boolean sink );
```

target must be HISTOGRAM if a histogram table is to be specified. *target* value PROXY_HISTOGRAM is a special case discussed later in this section. *width* specifies the number of entries in the histogram table, and *internalformat* specifies the format of each table entry. The maximum allowable *width* of the histogram table is implementation-dependent, but must be at least 32. *sink* specifies whether pixel groups will be consumed by the histogram operation (TRUE) or passed on to the minmax operation (FALSE).

The specified histogram table is redefined to have *width* entries, each with the specified internal format. The entries are indexed 0 through $width - 1$. Each component in each entry is set to zero. The values in the previous histogram table, if any, are lost.

A GL implementation may vary its allocation of internal component resolution based on any **Histogram** parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. In particular, allocations must be invariant; the same allocation must be made each time a histogram is specified with the same parameter values. These allocation rules also apply to the proxy histogram, which is described later in this section.

Errors

internalformat accepts the same values as the corresponding argument of **ColorTable**, with the exception of the values 1, 2, 3, and 4.

An INVALID_VALUE error is generated if *width* is not zero or a non-negative power of two.

A TABLE_TOO_LARGE error is generated if the specified histogram table

is too large for the implementation.

8.4.3.13 Histogram Query

The contents of the histogram table are queried with the commands

```
void GetHistogram( enum target, boolean reset,
    enum format, enum type, void *values );
void GetnHistogram( enum target, boolean reset,
    enum format, enum type, sizei bufSize, void *values );
```

target must be HISTOGRAM. *format* must be a pixel format from table 8.5 and *type* must be a data type from table 8.6. The one-dimensional histogram table image is returned to pixel pack buffer or client memory starting at *values*. Pixel processing and component mapping are identical to those of **GetTexImage**, except that instead of applying the Final Conversion pixel storage mode, component values are simply clamped to the range of the target data type.

If *reset* is TRUE, then all counters of all elements of the histogram are reset to zero. Counters are reset whether returned or not.

No counters are modified if *reset* is FALSE.

Errors

An INVALID_ENUM error is generated if *target* is not HISTOGRAM.

An INVALID_ENUM error is generated if *format* is not a pixel format from table 8.5.

An INVALID_ENUM error is generated if *type* is not a data type from table 8.6.

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetnHistogram** if the buffer size required to store the requested data is greater than *bufSize*.

The command

```
void ResetHistogram( enum target );
```

resets all counters of all elements of the histogram table to zero. *target* must be HISTOGRAM.

It is not an error to reset or query the contents of a histogram table with zero entries.

The commands

```
void GetHistogramParameter{if}v( enum target,
    enum pname, T *params );
```

are used for integer and floating-point query. *target* must be HISTOGRAM or PROXY_HISTOGRAM. *pname* is one of HISTOGRAM_FORMAT, HISTOGRAM_WIDTH, HISTOGRAM_RED_SIZE, HISTOGRAM_GREEN_SIZE, HISTOGRAM_BLUE_SIZE, HISTOGRAM_ALPHA_SIZE, or HISTOGRAM_LUMINANCE_SIZE. *pname* may be HISTOGRAM_SINK only for *target* HISTOGRAM. The value of the specified parameter is returned in *params*.

8.4.3.14 Histogram State and Proxy State

The state necessary for histogram operation is an array of values, with which is associated a width, an integer describing the internal format of the histogram, five integer values describing the resolutions of each of the red, green, blue, alpha, and luminance components of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the flag is FALSE.

In addition to the histogram table, a partially instantiated proxy histogram table is maintained. It includes width, internal format, and red, green, blue, alpha, and luminance component resolutions. The proxy table does not include image data or the flag. When **Histogram** is executed with *target* set to PROXY_HISTOGRAM, the proxy state values are recomputed and updated. If the histogram array is too large, no error is generated, but the proxy format, width, and component resolutions are set to zero. If the histogram table would be accomodated by **Histogram** called with *target* set to HISTOGRAM, the proxy state values are set exactly as though the actual histogram table were being specified. Calling **Histogram** with *target* PROXY_HISTOGRAM has no effect on the actual histogram table.

There is no image associated with PROXY_HISTOGRAM. It cannot be used as a histogram, and its image cannot be queried using **GetHistogram**.

8.4.3.15 Minmax Table Specification

The minmax table is specified with

```
void Minmax( enum target, enum internalformat,
    boolean sink );
```

target must be MINMAX. *internalformat* specifies the format of the table entries. *sink* specifies whether pixel groups will be consumed by the minmax operation (TRUE) or passed on to final conversion (FALSE).

internalformat accepts the same values as the corresponding argument of **ColorTable**, with the exception of the values 1, 2, 3, and 4, as well as the INTENSITY base and sized internal formats. The resulting table always has 2 entries, each with values corresponding only to the components of the internal format.

The state necessary for minmax operation is a table containing two elements (the first element stores the minimum values, the second stores the maximum values), an integer describing the internal format of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial state is a minimum table entry set to the maximum representable value and a maximum table entry set to the minimum representable value. Internal format is set to RGBA and the initial value of the flag is FALSE.

8.4.3.16 Minmax Query

The contents of the minmax table are queried with the commands

```
void GetMinmax( enum target, boolean reset, enum format,
    enum type, void *values );
void GetnMinmax( enum target, boolean reset,
    enum format, enum type, sizei bufSize, void *values );
```

target must be MINMAX. *format* must be a pixel format from table 8.5 and *type* must be a data type from table 8.6. A one-dimensional image of width 2 is returned to pixel pack buffer or client memory starting at *values*. Pixel processing and component mapping are identical to those of **GetTexImage**.

If *reset* is TRUE, then each minimum value is reset to the maximum representable value, and each maximum value is reset to the minimum representable value. All values are reset, whether returned or not.

No values are modified if *reset* is FALSE.

Errors

An INVALID_ENUM error is generated if *target* is not MINMAX.

An INVALID_ENUM error is generated if *format* is not a pixel format from table 8.5.

An INVALID_ENUM error is generated if *type* is not a data type from table 8.6.

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetnMinmax** if the buffer size required to store the requested data is greater than *bufSize*.

The command

```
void ResetMinmax( enum target );
```

resets all minimum and maximum values of *target* to to their maximum and minimum representable values, respectively, *target* must be MINMAX.

The commands

```
void GetMinmaxParameter{if}v( enum target, enum pname,
    T *params );
```

are used for integer and floating-point query. *target* must be MINMAX. *pname* is MINMAX_FORMAT or MINMAX_SINK. The value of the specified parameter is returned in *params*.

8.4.4 Transfer of Pixel Rectangles

The process of transferring pixels encoded in buffer object or client memory is diagrammed in figure 8.1. We describe the stages of this process in the order in which they occur.

Commands accepting or returning pixel rectangles take the following arguments (as well as additional arguments specific to their function):

format is a symbolic constant indicating what the values in memory represent.

width and *height* are the width and height, respectively, of the pixel rectangle to be transferred.

data refers to the data to be drawn. These data are represented with one of several GL data types, specified by *type*. The correspondence between the *type* token values and the GL data types they indicate is given in table 8.7.

Not all combinations of *format* and *type* are valid.

Errors

An INVALID_OPERATION error is generated if *format* is one of the INTEGER component formats defined in table 8.8 and *type* is one of the floating-point types defined in table 8.7.

An INVALID_ENUM error is generated if *type* is BITMAP and *format* is not COLOR_INDEX or STENCIL_INDEX.

Some additional constraints on the combinations of *format* and *type* values that are accepted are discussed below. Additional restrictions may be imposed by specific commands.

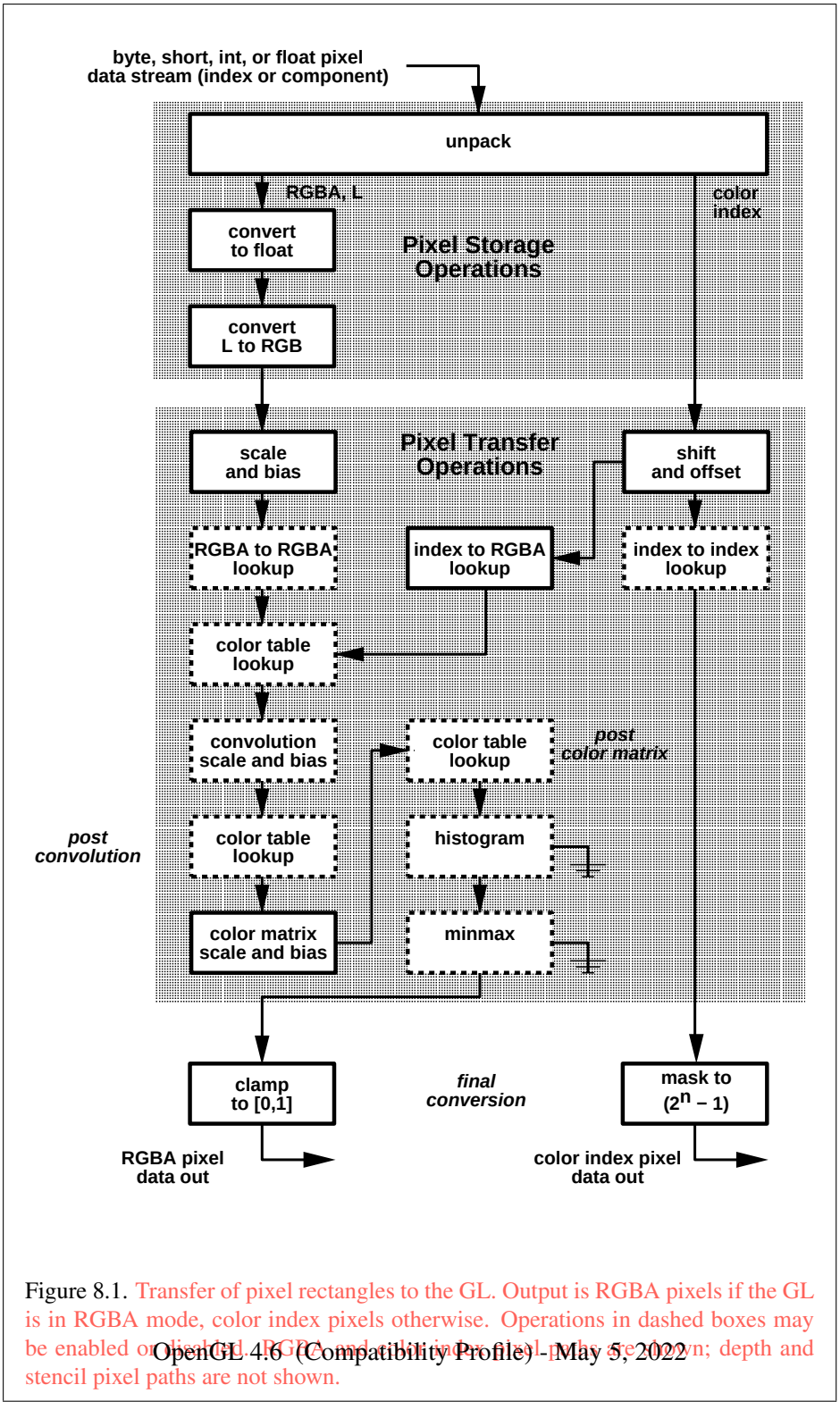


Figure 8.1. Transfer of pixel rectangles to the GL. Output is RGBA pixels if the GL is in RGBA mode, color index pixels otherwise. Operations in dashed boxes may be enabled or disabled. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation	Floating-Point
UNSIGNED_BYTE	ubyte	No	No
BITMAP	ubyte	Yes	No
BYTE	byte	No	No
UNSIGNED_SHORT	ushort	No	No
SHORT	short	No	No
UNSIGNED_INT	uint	No	No
INT	int	No	No
HALF_FLOAT	half	No	Yes
FLOAT	float	No	Yes
UNSIGNED_BYTE_3_3_2	ubyte	Yes	No
UNSIGNED_BYTE_2_3_3_REV	ubyte	Yes	No
UNSIGNED_SHORT_5_6_5	ushort	Yes	No
UNSIGNED_SHORT_5_6_5_REV	ushort	Yes	No
UNSIGNED_SHORT_4_4_4_4	ushort	Yes	No
UNSIGNED_SHORT_4_4_4_4_REV	ushort	Yes	No
UNSIGNED_SHORT_5_5_5_1	ushort	Yes	No
UNSIGNED_SHORT_1_5_5_5_REV	ushort	Yes	No
UNSIGNED_INT_8_8_8_8	uint	Yes	No
UNSIGNED_INT_8_8_8_8_REV	uint	Yes	No
UNSIGNED_INT_10_10_10_2	uint	Yes	No
UNSIGNED_INT_2_10_10_10_REV	uint	Yes	No
UNSIGNED_INT_24_8	uint	Yes	No
UNSIGNED_INT_10F_11F_11F_REV	uint	Yes	Yes
UNSIGNED_INT_5_9_9_9_REV	uint	Yes	Yes
FLOAT_32_UNSIGNED_INT_24_8_REV	n/a	Yes	No

Table 8.7: Pixel data *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described in section 8.4.4.2. Floating-point types are incompatible with INTEGER formats as described above.

8.4.4.1 Unpacking

Data are taken from the currently bound pixel unpack buffer or client memory as a sequence of signed or unsigned bytes (GL data types `byte` and `ubyte`), signed or unsigned short integers (GL data types `short` and `ushort`), signed or unsigned integers (GL data types `int` and `uint`), or floating-point values (GL data types `half` and `float`). These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a group. Table 8.8 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield floating-point or integer components.

If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), *data* is an offset into the pixel unpack buffer and the pixels are unpacked from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the pixels are unpacked from client memory relative to the pointer.

Errors

An `INVALID_OPERATION` error is generated if a pixel unpack buffer object is bound and unpacking the pixel data according to the process described below would access memory beyond the size of the pixel unpack buffer's memory size.

An `INVALID_OPERATION` error is generated if a pixel unpack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 8.7 for the *type* parameter (or not evenly divisible by 4 for *type* `FLOAT_32_UNSIGNED_INT_24_8_REV`, which does not have a corresponding GL data type).

By default the values of each GL data type are interpreted as they would be specified in the language of the client's GL binding. If `UNPACK_SWAP_BYTES` is enabled, however, then the values are interpreted with the bit orderings modified as per table 8.9. The modified bit orderings are defined only if the GL data type `ubyte` has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of *rows*, with the first element of the first group of the first row pointed to by *data*. If the value of `UNPACK_ROW_LENGTH` is zero, then the number of groups in a row is *width*; otherwise the number of groups is the value of `UNPACK_ROW_LENGTH`. If p indicates the location in memory of the first element

Format Name	Element Meaning and Order	Target Buffer
COLOR_INDEX	Color Index	Color Index
STENCIL_INDEX	Stencil Index	Stencil
DEPTH_COMPONENT	Depth	Depth
DEPTH_STENCIL	Depth and Stencil Index	Depth and Stencil
RED	R	Color
GREEN	G	Color
BLUE	B	Color
ALPHA	A	Color
RG	R, G	Color
RGB	R, G, B	Color
RGBA	R, G, B, A	Color
BGR	B, G, R	Color
BGRA	B, G, R, A	Color
LUMINANCE	Luminance	Color
LUMINANCE_ALPHA	Luminance, A	Color
RED_INTEGER	iR	Color
GREEN_INTEGER	iG	Color
BLUE_INTEGER	iB	Color
ALPHA_INTEGER	iA	Color
RG_INTEGER	iR, iG	Color
RGB_INTEGER	iR, iG, iB	Color
RGBA_INTEGER	iR, iG, iB, iA	Color
BGR_INTEGER	iB, iG, iR	Color
BGRA_INTEGER	iB, iG, iR, iA	Color

Table 8.8: Pixel data formats. The second column gives a description of and the number and order of elements in a group. Unless specified as an index, formats yield components. Components are floating-point unless prefixed with the letter 'i', which indicates they are integer.

Element Size	Default Bit Ordering	Modified Bit Ordering
8 bit	[7..0]	[7..0]
16 bit	[15..0]	[7..0][15..8]
32 bit	[31..0]	[7..0][15..8][23..16][31..24]

Table 8.9: Bit ordering modification of elements when UNPACK_SWAP_BYTES is enabled. These reorderings are defined only when GL data type ubyte has 8 bits, and then only for GL data types with 8, 16, or 32 bits. Bit 0 is the least significant.

of the first row, then the first element of the Nth row is indicated by

$$p + Nk \tag{8.1}$$

where N is the row number (counting from zero) and k is defined as

$$k = \begin{cases} nl & s \geq a, \\ \frac{a}{s}\left\lceil \frac{snl}{a} \right\rceil & s < a \end{cases} \tag{8.2}$$

where n is the number of elements in a group, l is the number of groups in the row, a is the value of UNPACK_ALIGNMENT, and s is the size, in units of GL ubytes, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then $k = nl$ for all values of a.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the *data* pointer is advanced by (UNPACK_SKIP_PIXELS)n + (UNPACK_SKIP_ROWS)k elements. Then *width* groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. *height* sets of *width* groups of values are obtained this way. See figure 8.2.

8.4.4.2 Special Interpretations

A *type* matching one of the types in table 8.10 is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. If *type* is FLOAT_32_UNSIGNED_INT_24_8_REV, the components of each group are contained within two 32-bit words; the first word contains the float component, and the second word contains a packed 24-bit unused field, followed by an 8-bit index. The number of components per

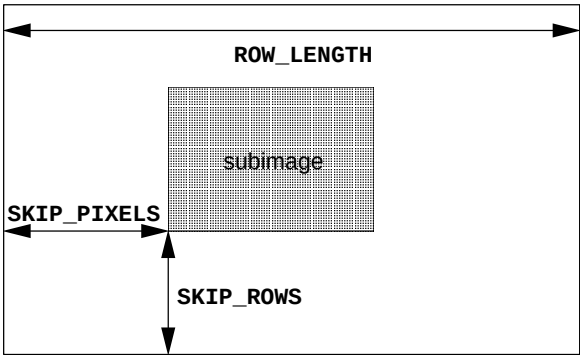


Figure 8.2. Selecting a subimage from an image. The indicated parameter names are prefixed by UNPACK_ for **DrawPixels** and by PACK_ for **ReadPixels**.

packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 8.10.

Errors

An INVALID_OPERATION error is generated by any command processing pixel rectangles if a mismatch occurs.

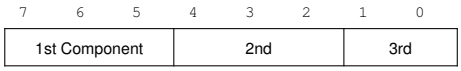
Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 8.11- 8.14. Each bitfield is interpreted as an unsigned integer value.

Components are normally packed with the first component in the most significant bits of the bitfield, and successive components occupying progressively less significant locations. Types whose token names end with _REV reverse the component packing order from least to most significant locations. In all cases, the most significant bit of each component is packed in the most significant bit location of its location in the bitfield.

type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_BYTE_3_3_2	ubyte	3	RGB, RGB_INTEGER
UNSIGNED_BYTE_2_3_3_REV	ubyte	3	RGB, RGB_INTEGER
UNSIGNED_SHORT_5_6_5	ushort	3	RGB, RGB_INTEGER
UNSIGNED_SHORT_5_6_5_REV	ushort	3	RGB, RGB_INTEGER
UNSIGNED_SHORT_4_4_4_4	ushort	4	RGBA, BGRA, RGBA_INTEGER, BGRA_INTEGER
UNSIGNED_SHORT_4_4_4_4_REV	ushort	4	RGBA, BGRA, RGBA_INTEGER, BGRA_INTEGER
UNSIGNED_SHORT_5_5_5_1	ushort	4	RGBA, BGRA, RGBA_INTEGER, BGRA_INTEGER
UNSIGNED_SHORT_1_5_5_5_REV	ushort	4	RGBA, BGRA, RGBA_INTEGER, BGRA_INTEGER
UNSIGNED_INT_8_8_8_8	uint	4	RGBA, BGRA, RGBA_INTEGER, BGRA_INTEGER
UNSIGNED_INT_8_8_8_8_REV	uint	4	RGBA, BGRA, RGBA_INTEGER, BGRA_INTEGER
UNSIGNED_INT_10_10_10_2	uint	4	RGBA, BGRA, RGBA_INTEGER, BGRA_INTEGER
UNSIGNED_INT_2_10_10_10_REV	uint	4	RGBA, BGRA, RGBA_INTEGER, BGRA_INTEGER
UNSIGNED_INT_24_8	uint	2	DEPTH_STENCIL
UNSIGNED_INT_10F_11F_11F_REV	uint	3	RGB
UNSIGNED_INT_5_9_9_9_REV	uint	4	RGB
FLOAT_32_UNSIGNED_INT_24_8_REV	n/a	2	DEPTH_STENCIL

Table 8.10: Packed pixel formats.

UNSIGNED_BYTE_3_3_2:



UNSIGNED_BYTE_2_3_3_REV:

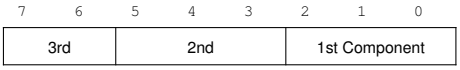
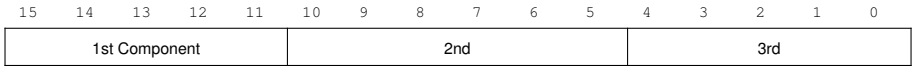
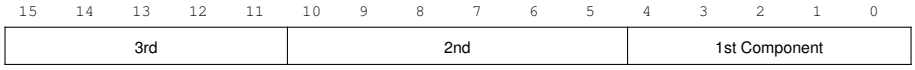


Table 8.11: UNSIGNED_BYTE formats. Bit numbers are indicated for each component.

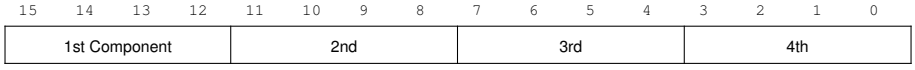
UNSIGNED_SHORT_5_6_5:



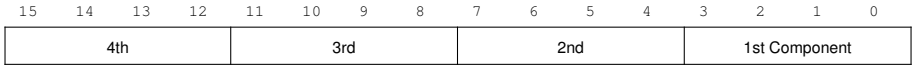
UNSIGNED_SHORT_5_6_5_REV:



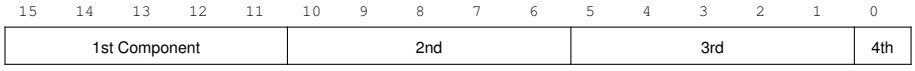
UNSIGNED_SHORT_4_4_4_4:



UNSIGNED_SHORT_4_4_4_4_REV:



UNSIGNED_SHORT_5_5_5_1:



UNSIGNED_SHORT_1_5_5_5_REV:

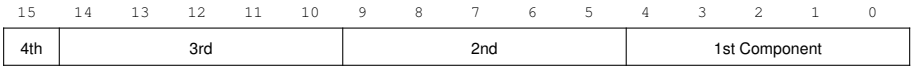
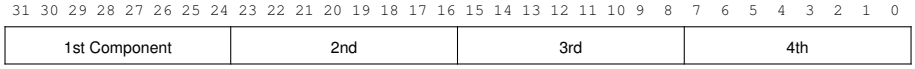
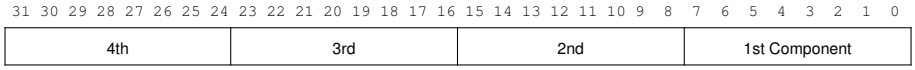


Table 8.12: UNSIGNED_SHORT formats

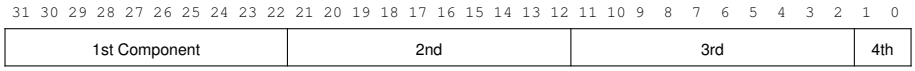
`UNSIGNED_INT_8_8_8_8`:



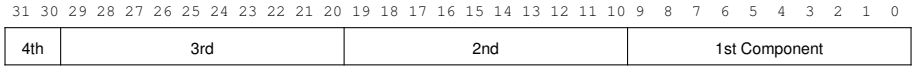
`UNSIGNED_INT_8_8_8_8_REV`:



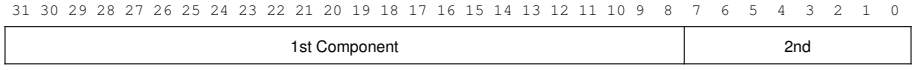
`UNSIGNED_INT_10_10_10_2`:



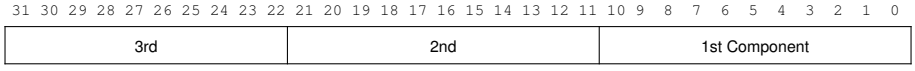
`UNSIGNED_INT_2_10_10_10_REV`:



`UNSIGNED_INT_24_8`:



`UNSIGNED_INT_10F_11F_11F_REV`:



`UNSIGNED_INT_5_9_9_9_REV`:

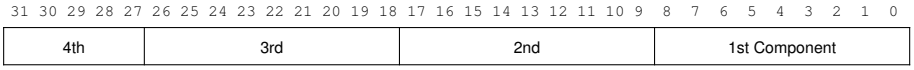


Table 8.13: `UNSIGNED_INT` formats

FLOAT_32_UNSIGNED_INT_24_8_REV:

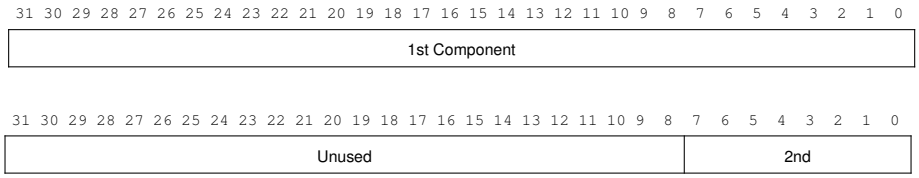


Table 8.14: FLOAT_UNSIGNED_INT formats

Format	First Component	Second Component	Third Component	Fourth Component
RGB	red	green	blue	
RGBA	red	green	blue	alpha
BGRA	blue	green	red	alpha
DEPTH_STENCIL	depth	stencil		

Table 8.15: Packed pixel field assignments.

The assignment of components to fields in the packed pixel is as described in table 8.15.

Byte swapping, if enabled, is performed before the components are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if "group" is substituted for "component" and the number of components per group is understood to be one.

A *type* of UNSIGNED_INT_10F_11F_11F_REV and *format* of RGB is a special case in which the data are a series of GL uint values. Each uint value specifies 3 packed components as shown in table 8.13. The 1st, 2nd, and 3rd components are called f_{red} (11 bits), f_{green} (11 bits), and f_{blue} (10 bits) respectively.

f_{red} and f_{green} are treated as unsigned 11-bit floating-point values and converted to floating-point red and green components respectively as described in section 2.3.4.3. f_{blue} is treated as an unsigned 10-bit floating-point value and converted to a floating-point blue component as described in section 2.3.4.4.

A *type* of UNSIGNED_INT_5_9_9_9_REV and *format* of RGB is a special case in which the data are a series of GL uint values. Each uint value specifies 4 packed components as shown in table 8.13. The 1st, 2nd, 3rd, and 4th components are called p_{red}, p_{green}, p_{blue}, and p_{exp} respectively and are treated as unsigned integers. These are then used to compute floating-point RGB components (ignoring the "Conversion to floating-point" section below in this case) as follows:

$$red = p_{red} 2^{p_{exp}-B-N}$$
$$green = p_{green} 2^{p_{exp}-B-N}$$
$$blue = p_{blue} 2^{p_{exp}-B-N}$$

where $B = 15$ (the exponent bias) and $N = 9$ (the number of mantissa bits).

8.4.4.3 Conversion to floating-point

This step applies only to groups of floating-point components. It is not performed on indices or integer components. For groups containing both components and indices, such as DEPTH_STENCIL, the indices are not converted.

Each element in a group is converted to a floating-point value. For unsigned or signed normalized fixed-point elements, equations 2.1 or 2.2, respectively, are used.

8.4.4.4 Conversion to RGB

This step is applied only if the *format* is LUMINANCE or LUMINANCE_ALPHA. If the *format* is LUMINANCE, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the *format* is LUMINANCE_ALPHA, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

8.4.4.5 Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to one for integer components or 1.0 for floating-point components. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0 for integer components or 0.0 for floating-point components.

8.4.4.6 Pixel Transfer Operations

This step is actually a sequence of steps. Because the pixel transfer operations are performed equivalently during the drawing, copying, and reading of pixels (see chapter 18), and during the specification of texture images (either from memory or from the framebuffer), they are described separately in section 8.4.5. After the operations described in that section are completed, groups are processed as described in the following sections.

8.4.5 Pixel Transfer Operations

The GL defines six kinds of pixel groups:

1. *Floating-point RGBA component:* Each group comprises four color components in floating-point format: red, green, blue, and alpha.
2. *Integer RGBA component:* Each group comprises four color components in integer format: red, green, blue, and alpha.
3. *Depth component:* Each group comprises a single depth component.
4. *Color index:* Each group comprises a single color index.
5. *Stencil index:* Each group comprises a single stencil index.
6. *Depth/stencil:* Each group comprises a single depth component and a single stencil index.

Each operation described in this section is applied sequentially to each pixel group in an image. Many operations are applied only to pixel groups of certain kinds; if an operation is not applicable to a given group, it is skipped. None of the operations defined in this section affect integer RGBA component pixel groups.

This step applies only to RGBA component and depth component groups, and to the depth components in depth/stencil groups. Each component is multiplied by an appropriate signed scale factor: `RED_SCALE` for an R component, `GREEN_SCALE` for a G component, `BLUE_SCALE` for a B component, and `ALPHA_SCALE` for an A component, or `DEPTH_SCALE` for a depth component. Then the result is added to the appropriate signed bias: `RED_BIAS`, `GREEN_BIAS`, `BLUE_BIAS`, `ALPHA_BIAS`, or `DEPTH_BIAS`.

8.4.5.1 Arithmetic on Indices

This step applies only to color index and stencil index groups, and to the stencil indices in depth/stencil groups. If the index is a floating-point value, it is converted to fixed-point, with an unspecified number of bits to the right of the binary point and at least $\lceil \log_2(\texttt{MAX_PIXEL_MAP_TABLE}) \rceil$ bits to the left of the binary point. Indices that are already integers remain so; any fraction bits in the resulting fixed-point value are zero.

The fixed-point index is then shifted by $|\texttt{INDEX_SHIFT}|$ bits, left if $\texttt{INDEX_SHIFT} > 0$ and right otherwise. In either case the shift is zero-filled. Then, the signed integer offset `INDEX_OFFSET` is added to the index.

8.4.5.2 RGBA to RGBA Lookup

This step applies only to RGBA component groups, and is skipped if MAP_COLOR is FALSE. First, each component is clamped to the range $[0, 1]$. There is a table associated with each of the R, G, B, and A component elements: PIXEL_MAP_R_TO_R for R, PIXEL_MAP_G_TO_G for G, PIXEL_MAP_B_TO_B for B, and PIXEL_MAP_A_TO_A for A. Each element is multiplied by an integer one less than the size of the corresponding table, and, for each element, an address is found by rounding this value to the nearest integer. For each element, the addressed value in the corresponding table replaces the element.

8.4.5.3 Color Index Lookup

This step applies only to color index groups. If the GL command that invokes the pixel transfer operation requires that RGBA component pixel groups be generated, then a conversion is performed at this step. RGBA component pixel groups are required if

- The groups will be rasterized, and the GL is in RGBA mode, or
- The groups will be loaded as an image into texture memory, or
- The groups will be returned to client memory with a format other than COLOR_INDEX.

If RGBA component groups are required, then the integer part of the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A. Each of these tables must have 2^n entries for some integer value of n (n may be different for each table). For each table, the index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The indexed value becomes an R, G, B, or A value, as appropriate. The group of four elements so obtained replaces the index, changing the group's type to RGBA component.

If RGBA component groups are not required, and if MAP_COLOR is enabled, then the index is looked up in the PIXEL_MAP_I_TO_I table (otherwise, the index is not looked up). Again, the table must have 2^n entries for some integer n. The index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The value in the table replaces the index. The floating-point table value is first rounded to a fixed-point value with unspecified precision. The group's type remains color index.

Base Internal Format	R	G	B	A
ALPHA				A_t
LUMINANCE	L_t	L_t	L_t	
LUMINANCE_ALPHA	L_t	L_t	L_t	A_t
INTENSITY	I_t	I_t	I_t	I_t
RGB	R_t	G_t	B_t	
RGBA	R_t	G_t	B_t	A_t

Table 8.16: Color table lookup. R_t, G_t, B_t, A_t, L_t, and I_t are color table values that are assigned to pixel components R, G, B, and A depending on the table format. When there is no assignment, the component value is left unchanged by lookup.

8.4.5.4 Stencil Index Lookup

This step applies only to stencil index groups, and to the stencil indices in depth/stencil groups. If MAP_STENCIL is enabled, then the index is looked up in the PIXEL_MAP_S_TO_S table (otherwise, the index is not looked up). The table must have 2^n entries for some integer n. The integer index is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The integer value in the table replaces the index.

8.4.5.5 Color Table Lookup

This step applies only to RGBA component groups. Color table lookup is only done if COLOR_TABLE is enabled. If a zero-width table is enabled, no lookup is performed.

The internal format of the table determines which components of the group will be replaced (see table 8.16). The components to be replaced are converted to indices by clamping to $[0, 1]$, multiplying by an integer one less than the width of the table, and rounding to the nearest integer. Components are replaced by the table entry at the index.

The required state is one bit indicating whether color table lookup is enabled or disabled. In the initial state, lookup is disabled.

8.4.5.6 Convolution

This step applies only to RGBA component groups. If CONVOLUTION_1D is enabled, the one-dimensional convolution filter is applied only to the one-

Base Filter Format	R	G	B	A
ALPHA	R_s	G_s	B_s	$A_s * A_f$
LUMINANCE	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	A_s
LUMINANCE_ALPHA	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	$A_s * A_f$
INTENSITY	$R_s * I_f$	$G_s * I_f$	$B_s * I_f$	$A_s * I_f$
RGB	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	A_s
RGBA	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	$A_s * A_f$

Table 8.17: Computation of filtered color components depending on filter image format. $C * F$ indicates the convolution of image component C with filter F.

dimensional texture images passed to **TexImage1D**, **TexSubImage1D**, **CopyTexImage1D**, and **CopyTexSubImage1D**. If CONVOLUTION_2D is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to **DrawPixels**, **CopyPixels**, **ReadPixels**, **TexImage2D**, **TexSubImage2D**, **CopyTexImage2D**, **CopyTexSubImage2D**, and **CopyTexSubImage3D**. If SEPARABLE_2D is enabled, and CONVOLUTION_2D is disabled, the separable two-dimensional convolution filter is instead applied these images.

The convolution operation is a sum of products of source image pixels and convolution filter pixels. Source image pixels always have four components: red, green, blue, and alpha, denoted in the equations below as R_s, G_s, B_s, and A_s. Filter pixels may be stored in one of five formats, with 1, 2, 3, or 4 components. These components are denoted as R_f, G_f, B_f, A_f, L_f, and I_f in the equations below. The result of the convolution operation is the 4-tuple R,G,B,A. Depending on the internal format of the filter, individual color components of each source image pixel are convolved with one filter component, or are passed unmodified. The rules for this are defined in table 8.17.

The convolution operation is defined differently for each of the three convolution filters. The variables W_f and H_f refer to the dimensions of the convolution filter. The variables W_s and H_s refer to the dimensions of the source pixel image.

The convolution equations are defined as follows, where C refers to the filtered result, C_f refers to the one- or two-dimensional convolution filter, and C_{row} and C_{column} refer to the two one-dimensional filters comprising the two-dimensional separable filter. C'_s depends on the source image color C_s and the convolution border mode as described below. C_r, the filtered output image, depends on all of these variables and is described separately for each border mode. The pixel indexing nomenclature is decribed in section 8.4.3.5.

One-dimensional filter:

$$C[i'] = \sum_{n=0}^{W_f-1} C'_s[i'+n] * C_f[n]$$

Two-dimensional filter:

$$C[i',j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C'_s[i'+n,j'+m] * C_f[n,m]$$

Two-dimensional separable filter:

$$C[i',j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C'_s[i'+n,j'+m] * C_{row}[n] * C_{column}[m]$$

If W_f of a one-dimensional filter is zero, then $C[i]$ is always set to zero. Likewise, if either W_f or H_f of a two-dimensional filter is zero, then $C[i,j]$ is always set to zero.

The convolution border mode for a specific convolution filter is specified by calling

`void` **ConvolutionParameter**{**if**}(`enum` *target*`, enum` *pname*`,`
`T` *param* `);`

where *target* is the name of the filter, *pname* is `CONVOLUTION_BORDER_MODE`, and *param* is one of `REDUCE`, `CONSTANT_BORDER` or `REPLICATE_BORDER`.

8.4.5.7 Border Mode `REDUCE`

The width and height of source images convolved with border mode `REDUCE` are reduced by $W_f - 1$ and $H_f - 1$, respectively. If this reduction would generate a resulting image with zero or negative width and/or height, the output is simply null, with no error generated. The coordinates of the image that results from a convolution with border mode `REDUCE` are zero through $W_s - W_f$ in width, and zero through $H_s - H_f$ in height. In cases where errors can result from the specification of invalid image dimensions, it is these resulting dimensions that are tested, not the dimensions of the source image. (A specific example is **TexImage1D** and **TexImage2D**, which specify constraints for image dimensions. Even if **TexImage1D** or **TexImage2D** is called with a `NULL` pixel pointer, the dimensions of the resulting texture image are those that would result from the convolution of the specified image).

When the border mode is REDUCE, C'_s equals the source image color C_s and C_r equals the filtered result C.

For the remaining border modes, define

$$C_w = \left\lfloor \frac{W_f}{2} \right\rfloor$$
$$C_h = \left\lfloor \frac{H_f}{2} \right\rfloor .$$

The coordinates (C_w, C_h) define the center of the convolution filter.

8.4.5.8 Border Mode CONSTANT_BORDER

If the convolution border mode is CONSTANT_BORDER, the output image has the same dimensions as the source image. The result of the convolution is the same as if the source image were surrounded by pixels with the same color as the current convolution border color. Whenever the convolution filter extends beyond one of the edges of the source image, the constant-color border pixels are used as input to the filter. The current convolution border color is set by calling **ConvolutionParameterfv** or **ConvolutionParameteriv** with *pname* set to CONVOLUTION_BORDER_COLOR and *params* containing four values that comprise the RGBA color to be used as the image border. Integer color components are interpreted linearly such that the largest positive integer maps to 1.0, and the smallest negative integer maps to -1.0. Floating-point color components are not clamped when they are specified.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where $C[i']$ is computed using the following equation for $C'_s[i']$:

$$C'_s[i'] = \begin{cases} C_s[i'], & 0 \le i' < W_s \\ C_c, & \text{otherwise} \end{cases}$$

and C_c is the convolution border color.

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i, j] = C[i - C_w, j - C_h]$$

where $C[i', j']$ is computed using the following equation for $C'_s[i', j']$:

$$C'_s[i', j'] = \begin{cases} C_s[i', j'], & 0 \le i' < W_s, 0 \le j' < H_s \\ C_c, & otherwise \end{cases}$$

8.4.5.9 Border Mode REPLICATE_BORDER

The convolution border mode REPLICATE_BORDER also produces an output image with the same dimensions as the source image. The behavior of this mode is identical to that of the CONSTANT_BORDER mode except for the treatment of pixel locations where the convolution filter extends beyond the edge of the source image. For these locations, it is as if the outermost one-pixel border of the source image was replicated. Conceptually, each pixel in the leftmost one-pixel column of the source image is replicated C_w times to provide additional image data along the left edge, each pixel in the rightmost one-pixel column is replicated C_w times to provide additional image data along the right edge, and each pixel value in the top and bottom one-pixel rows is replicated to create C_h rows of image data along the top and bottom edges. The pixel value at each corner is also replicated in order to provide data for the convolution operation at each corner of the source image.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where $C[i']$ is computed using the following equation for $C'_s[i']$:

$$C'_s[i'] = C_s[\text{clamp}(i', W_s)]$$

and the clamping function $\text{clamp}(val, max)$ is defined as

$$\text{clamp}(val, max) = \begin{cases} 0, & val < 0 \\ val, & 0 \leq val < max \\ max - 1, & val \geq max \end{cases}$$

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i, j] = C[i - C_w, j - C_h]$$

where $C[i', j']$ is computed using the following equation for $C'_s[i', j']$:

$$C'_s[i', j'] = C_s[\text{clamp}(i', W_s), \text{clamp}(j', H_s)]$$

If a convolution operation is performed, each component of the resulting image is scaled by the corresponding **PixelTransfer** parameters: POST_CONVOLUTION_RED_SCALE for an R component, POST_CONVOLUTION_GREEN_SCALE for a G component, POST_CONVOLUTION_BLUE_SCALE for a B component, and POST_CONVOLUTION_ALPHA_SCALE for an A component. The result is added to the

corresponding bias: POST_CONVOLUTION_RED_BIAS, POST_CONVOLUTION_GREEN_BIAS, POST_CONVOLUTION_BLUE_BIAS, or POST_CONVOLUTION_ALPHA_BIAS.

The required state is three bits indicating whether each of one-dimensional, two-dimensional, or separable two-dimensional convolution is enabled or disabled, an integer describing the current convolution border mode, and four floating-point values specifying the convolution border color. In the initial state, all convolution operations are disabled, the border mode is REDUCE, and the border color is $(0, 0, 0, 0)$.

8.4.5.10 Post Convolution Color Table Lookup

This step applies only to RGBA component groups. Post convolution color table lookup is enabled or disabled by calling **Enable** or **Disable** with *target* POST_CONVOLUTION_COLOR_TABLE. The post convolution table is defined by calling **ColorTable** with a *target* argument of POST_CONVOLUTION_COLOR_TABLE. In all other respects, operation is identical to color table lookup, as defined earlier in section 8.4.5.5.

The required state is one bit indicating whether post convolution table lookup is enabled or disabled. In the initial state, lookup is disabled.

8.4.5.11 Color Matrix Transformation

This step applies only to RGBA component groups. The components are transformed by the color matrix. Each transformed component is multiplied by an appropriate signed scale factor: POST_COLOR_MATRIX_RED_SCALE for an R component, POST_COLOR_MATRIX_GREEN_SCALE for a G component, POST_COLOR_MATRIX_BLUE_SCALE for a B component, and POST_COLOR_MATRIX_ALPHA_SCALE for an A component. The result is added to a signed bias: POST_COLOR_MATRIX_RED_BIAS, POST_COLOR_MATRIX_GREEN_BIAS, POST_COLOR_MATRIX_BLUE_BIAS, or POST_COLOR_MATRIX_ALPHA_BIAS. The resulting components replace each component of the original group.

That is, if M_c is the color matrix, a subscript of s represents the scale term for a component, and a subscript of b represents the bias term, then the components

$$\begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

are transformed to

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} = \begin{pmatrix} R_s & 0 & 0 & 0 \\ 0 & G_s & 0 & 0 \\ 0 & 0 & B_s & 0 \\ 0 & 0 & 0 & A_s \end{pmatrix} M_c \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} R_b \\ G_b \\ B_b \\ A_b \end{pmatrix}.$$

8.4.5.12 Post Color Matrix Color Table Lookup

This step applies only to RGBA component groups. Post color matrix color table lookup is enabled or disabled by calling **Enable** or **Disable** with *target* `POST_COLOR_MATRIX_COLOR_TABLE`. The post color matrix table is defined by calling **ColorTable** with a *target* argument of `POST_COLOR_MATRIX_COLOR_TABLE`. In all other respects, operation is identical to color table lookup, as defined in section 8.4.5.5.

The required state is one bit indicating whether post color matrix lookup is enabled or disabled. In the initial state, lookup is disabled.

8.4.5.13 Histogram

This step applies only to RGBA component groups. Histogram operation is enabled or disabled by calling **Enable** or **Disable** with *target* `HISTOGRAM`.

If the width of the table is non-zero, then indices R_i, G_i, B_i, and A_i are derived from the red, green, blue, and alpha components of each pixel group (without modifying these components) by clamping each component to $[0, 1]$, multiplying by one less than the width of the histogram table, and rounding to the nearest integer. If the format of the `HISTOGRAM` table includes red or luminance, the red or luminance component of histogram entry R_i is incremented by one. If the format of the `HISTOGRAM` table includes green, the green component of histogram entry G_i is incremented by one. The blue and alpha components of histogram entries B_i and A_i are incremented in the same way. If a histogram entry component is incremented beyond its maximum value, its value becomes undefined; this is not an error.

If the **Histogram** *sink* parameter is `FALSE`, histogram operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the histogram operation is completed. Because histogram precedes minmax, no minmax operation is performed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

8.4.5.14 Minmax

This step applies only to RGBA component groups. Minmax operation is enabled or disabled by calling **Enable** or **Disable** with *target* `MINMAX`.

If the format of the minmax table includes red or luminance, the red component value replaces the red or luminance value in the minimum table element if and only if it is less than that component. Likewise, if the format includes red or luminance and the red component of the group is greater than the red or luminance value in the maximum element, the red group component replaces the red or luminance maximum component. If the format of the table includes green, the green group component conditionally replaces the green minimum and/or maximum if it is smaller or larger, respectively. The blue and alpha group components are similarly tested and replaced, if the table format includes blue and/or alpha. The internal type of the minimum and maximum component values is floating-point, with at least the same representable range as a floating-point number used to represent colors (section 2.3.4). There are no semantics defined for the treatment of group component values that are outside the representable range.

If the **Minmax** *sink* parameter is `FALSE`, minmax operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the minmax operation is completed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

8.5 Texture Image Specification

The command

```
void TexImage3D( enum target, int level, int internalformat,
    sizei width, sizei height, sizei depth, int border,
    enum format, enum type, const void *data );
```

is used to specify a three-dimensional texture image. *target* must be one of `TEXTURE_3D` for a three-dimensional texture, `TEXTURE_2D_ARRAY` for a two-dimensional array texture, or `TEXTURE_CUBE_MAP_ARRAY` for a cube map array texture. Additionally, *target* may be either `PROXY_TEXTURE_3D` for a three-dimensional proxy texture, `PROXY_TEXTURE_2D_ARRAY` for a two-dimensional proxy array texture, or `PROXY_TEXTURE_CUBE_MAP_ARRAY` for a cube map array texture, as discussed in section 8.22. *format*, *type*, and *data* specify the format of

the image data, the type of those data, and a reference to the image data in the currently bound pixel unpack buffer or client memory, as described in section 8.4.4.

The groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a two-dimensional image, whose size and organization are specified by the *width* and *height* parameters to **TexImage3D**. The values of UNPACK_ROW_LENGTH and UNPACK_ALIGNMENT control the row-to-row spacing in these images as described in section 8.4.4. If the value of the integer parameter UNPACK_IMAGE_HEIGHT is not positive, then the number of rows in each two-dimensional image is *height*; otherwise the number of rows is UNPACK_IMAGE_HEIGHT. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a three-dimensional image relies on the integer parameter UNPACK_SKIP_IMAGES. If UNPACK_SKIP_IMAGES is positive, the pointer is advanced by UNPACK_SKIP_IMAGES times the number of elements in one two-dimensional image before obtaining the first group from memory. Then *depth* two-dimensional images are processed, each having a subimage extracted as described in section 8.4.4.

The selected groups are transferred to the GL as described in section 8.4.4 and then clamped to the representable range of the internal format as follows:

- If the *internalformat* of the texture is signed or unsigned integer, components are clamped to $[-2^{n-1}, 2^{n-1} - 1]$ or $[0, 2^n - 1]$, respectively, where n is the number of bits per component.
- For color component groups, if the *internalformat* of the texture is signed or unsigned normalized fixed-point:
  - If the type of the data is a floating-point type (as defined in table 8.7), it is clamped to $[-1, 1]$ or $[0, 1]$, respectively.
  - Otherwise, it is clamped to to $[-2^{n-1}, 2^{n-1} - 1]$ or $[0, 2^n - 1]$, respectively, where n is the number of bits in the normalized representation.
- For depth component groups, the depth value is clamped to $[0, 1]$.
- Stencil index values are masked by $2^n - 1$, where n is the number of stencil bits in the internal format resolution (see below).
- Otherwise, values are not modified.

If the base internal format is DEPTH_STENCIL and *format* is not DEPTH_STENCIL, then the values of the stencil index texture components are undefined.

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
ALPHA	A	A
DEPTH_COMPONENT	Depth	D
DEPTH_STENCIL	Depth,Stencil	D,S
INTENSITY	R	I
LUMINANCE	R	L
LUMINANCE_ALPHA	R,A	L,A
RED	R	R
RG	R,G	R,G
RGB	R,G,B	R,G,B
RGBA	R,G,B,A	R,G,B,A
STENCIL_INDEX	Stencil	S

Table 8.18: Conversion from RGBA, depth, and stencil pixel components to internal texture, table, or filter components. See section 16.1 for a description of the texture components R, G, B, A, L, I, D, and S.

Components are then selected from the resulting R, G, B, A, depth, or stencil values to obtain a texture with the *base internal format* specified by (or derived from) *internalformat*. Table 8.18 summarizes the mapping of R, G, B, A, depth, or stencil values to texture components, as a function of the base internal format of the texture image. *internalformat* may be specified as one of the internal format symbolic constants listed in table 8.18, as one of the *sized internal format* symbolic constants listed in tables 8.19- 8.21, as one of the generic compressed internal format symbolic constants listed in table 8.22, or as one of the specific compressed internal format symbolic constants (if listed in table 8.22). *internalformat* may (for backwards compatibility with the 1.0 version of the GL) also take on the integer values 1, 2, 3, and 4, which are equivalent to symbolic constants LUMINANCE, LUMINANCE_ALPHA, RGB, and RGBA respectively.

Textures with a base internal format of DEPTH_COMPONENT, DEPTH_STENCIL, or STENCIL_INDEX are supported by texture image specification commands only if *target* is TEXTURE_1D, TEXTURE_2D, TEXTURE_2D_MULTISAMPLE, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_2D_MULTISAMPLE_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP, TEXTURE_CUBE_MAP_ARRAY, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_2D_MULTISAMPLE, PROXY_TEXTURE_1D_ARRAY, PROXY_TEXTURE_2D_ARRAY, PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY, PROXY_TEXTURE_RECTANGLE, PROXY_TEXTURE_CUBE_MAP, or PROXY_-

TEXTURE_CUBE_MAP_ARRAY.

An INVALID_OPERATION error is generated if these formats are used in conjunction with any other *target*.

Textures with a base internal format of DEPTH_COMPONENT or DEPTH_STENCIL require either depth component data or depth/stencil component data. Textures with other base internal formats require RGBA component data.

Textures with integer internal formats (see tables 8.19- 8.20) require integer data.

In addition to the specific compressed internal formats listed in table 8.22, the GL provides a mechanism to query token values for specific compressed internal formats, suitable for general-purpose[2] usage. Formats with restrictions that need to be specifically understood prior to use will not be returned by this query. The number of specific compressed internal formats is obtained by querying the value of NUM_COMPRESSED_TEXTURE_FORMATS. The set of specific compressed internal formats is obtained by querying COMPRESSED_TEXTURE_FORMATS with **GetIntegerv**, returning an array containing that number of values.

Generic compressed internal formats are never used directly as the internal formats of texture images. If *internalformat* is one of the six generic compressed internal formats, its value is replaced by the symbolic constant for a specific compressed internal format of the GL's choosing with the same base internal format. If no specific compressed format is available, *internalformat* is instead replaced by the corresponding base internal format. If *internalformat* is given as or mapped to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures or borders), *internalformat* is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

The *internal component resolution* is the number of bits allocated to each value in a texture image. If *internalformat* is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing, referred to as the *effective internal format*. The effective internal format chosen may change depending only on the values of *format* and *type*, and affects format compatibility for commands such as **TextureView** (see section 8.18) and **CopyImageSubData** (see section 18.3.3). If a sized internal format is specified, the mapping of the R, G, B, A, depth, and stencil values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 8.18; the type (unsigned int, float, etc.) is assigned the same

[2]These queries have been deprecated in OpenGL 4.2, because the vagueness of the term "general-purpose" makes it possible for implementations to choose to return **no** formats from the query.

type specified by *internalformat*; and the memory allocation per texture component is assigned by the GL to match the allocations listed in tables 8.19- 8.21 as closely as possible. (The definition of closely is left up to the implementation. However, a non-zero number of bits must be allocated for each component whose *desired* allocation in tables 8.19- 8.21 is non-zero, and zero bits must be allocated for all other components).

8.5.1 Required Texture Formats

Implementations are required to support at least one allocation of internal component resolution for each type (unsigned int, float, etc.) for each base internal format.

In addition, implementations are required to support the following sized and compressed internal formats. Requesting one of these sized internal formats for any texture type will allocate at least the internal component sizes, and exactly the component types shown for that format in the corresponding table:

- Color formats which are checked in the "Req. tex." column of table 8.19.
- All of the specific compressed texture formats in table 8.22.
- Depth, depth+stencil, and stencil formats which are checked in the "Req. format" column of table 8.21.

8.5.2 Encoding of Special Internal Formats

If *internalformat* is `R11F_G11F_B10F`, the red, green, and blue bits are converted to unsigned 11-bit, unsigned 11-bit, and unsigned 10-bit floating-point values as described in sections 2.3.4.3 and 2.3.4.4.

If *internalformat* is `RGB9_E5`, the red, green, and blue bits are converted to a shared exponent format according to the following procedure:

Components red, $green$, and $blue$ are first clamped (in the process, mapping NaN to zero) as follows:

$$\begin{aligned} red_c &= \max(0, min(sharedexp_{max}, red)) \\ green_c &= \max(0, min(sharedexp_{max}, green)) \\ blue_c &= \max(0, min(sharedexp_{max}, blue)) \end{aligned}$$

where

$$sharedexp_{max} = \frac{(2^N - 1)}{2^N} 2^{E_{max} - B}.$$

N is the number of mantissa bits per component (9), B is the exponent bias (15), and E_{max} is the maximum allowed biased exponent value (31).

The largest clamped component, max_c, is determined:

$$max_c = \max(red_c, green_c, blue_c)$$

A preliminary shared exponent exp_p is computed:

$$exp_p = \max(-B - 1, \lfloor \log_2(max_c) \rfloor) + 1 + B$$

A refined shared exponent exp_s is computed:

$$max_s = \left\lfloor \frac{max_c}{2^{exp_p - B - N}} + \frac{1}{2} \right\rfloor$$

$$exp_s = \begin{cases} exp_p, & 0 \leq max_s < 2^N \\ exp_p + 1, & max_s = 2^N \end{cases}$$

Finally, three integer values in the range 0 to $2^N - 1$ are computed:

$$red_s = \left\lfloor \frac{red_c}{2^{exp_s - B - N}} + \frac{1}{2} \right\rfloor$$
$$green_s = \left\lfloor \frac{green_c}{2^{exp_s - B - N}} + \frac{1}{2} \right\rfloor$$
$$blue_s = \left\lfloor \frac{blue_c}{2^{exp_s - B - N}} + \frac{1}{2} \right\rfloor$$

The resulting red_s, $green_s$, $blue_s$, and exp_s are stored in the red, green, blue, and shared bits respectively of the texture image.

An implementation accepting pixel data of *type* `UNSIGNED_INT_5_9_9_9_REV` with *format* `RGB` is allowed to store the components "as is" if the implementation can determine the current pixel transfer state acts as an identity transform on the components.

Sized Internal Format	Base Internal Format	Bits/component S are shared bits R	G	B	A	S	CR	Req. rend.	Req. tex.
ALPHA4	ALPHA				4		✓		
ALPHA8	ALPHA				8		✓	✓	✓
Sized internal color formats continued on next page									

Sized internal color formats continued from previous page									
Sized Internal Format	Base Internal Format	Bits/component S are shared bits					CR	Req. rend.	Req. tex.
		R	*G*	*B*	*A*	S			
ALPHA12	ALPHA				12		✓		
ALPHA16	ALPHA				16		✓		
R8	RED	8					✓	✓	✓
R8_SNORM	RED	s8					✓		✓
R16	RED	16					✓	✓	✓
R16_SNORM	RED	s16					✓		✓
RG8	RG	8	8				✓	✓	✓
RG8_SNORM	RG	s8	s8				✓		✓
RG16	RG	16	16				✓	✓	✓
RG16_SNORM	RG	s16	s16				✓		✓
R3_G3_B2	RGB	3	3	2			✓		✓
RGB4	RGB	4	4	4			✓		✓
RGB5	RGB	5	5	5			✓		✓
RGB565	RGB	5	6	5			✓	✓	✓
RGB8	RGB	8	8	8			✓		✓
RGB8_SNORM	RGB	s8	s8	s8			✓		✓
RGB10	RGB	10	10	10			✓		✓
RGB12	RGB	12	12	12			✓		✓
RGB16	RGB	16	16	16			✓		✓
RGB16_SNORM	RGB	s16	s16	s16			✓		✓
RGBA2	RGBA	2	2	2	2		✓		✓
RGBA4	RGBA	4	4	4	4		✓	✓	✓
RGB5_A1	RGBA	5	5	5	1		✓	✓	✓
RGBA8	RGBA	8	8	8	8		✓	✓	✓
RGBA8_SNORM	RGBA	s8	s8	s8	s8		✓		✓
RGB10_A2	RGBA	10	10	10	2		✓	✓	✓
RGB10_A2UI	RGBA	ui10	ui10	ui10	ui2		✓	✓	✓
RGBA12	RGBA	12	12	12	12		✓		✓
RGBA16	RGBA	16	16	16	16		✓	✓	✓
RGBA16_SNORM	RGBA	s16	s16	s16	s16		✓		✓
SRGB8	RGB	8	8	8			✓		✓
SRGB8_ALPHA8	RGBA	8	8	8	8		✓	✓	✓
R16F	RED	f16					✓	✓	✓
Sized internal color formats continued on next page									

Sized internal color formats continued from previous page									
Sized Internal Format	Base Internal Format	Bits/component S are shared bits					CR	Req. rend.	Req. tex.
		R	*G*	*B*	*A*	S			
RG16F	RG	f16	f16				✓	✓	✓
RGB16F	RGB	f16	f16	f16			✓		✓
RGBA16F	RGBA	f16	f16	f16	f16		✓	✓	✓
R32F	RED	f32					✓	✓	✓
RG32F	RG	f32	f32				✓	✓	✓
RGB32F	RGB	f32	f32	f32			✓		✓
RGBA32F	RGBA	f32	f32	f32	f32		✓	✓	✓
R11F_G11F_B10F	RGB	f11	f11	f10			✓	✓	✓
RGB9_E5	RGB	9	9	9		5			✓
R8I	RED	i8					✓	✓	✓
R8UI	RED	ui8					✓	✓	✓
R16I	RED	i16					✓	✓	✓
R16UI	RED	ui16					✓	✓	✓
R32I	RED	i32					✓	✓	✓
R32UI	RED	ui32					✓	✓	✓
RG8I	RG	i8	i8				✓	✓	✓
RG8UI	RG	ui8	ui8				✓	✓	✓
RG16I	RG	i16	i16				✓	✓	✓
RG16UI	RG	ui16	ui16				✓	✓	✓
RG32I	RG	i32	i32				✓	✓	✓
RG32UI	RG	ui32	ui32				✓	✓	✓
RGB8I	RGB	i8	i8	i8			✓		✓
RGB8UI	RGB	ui8	ui8	ui8			✓		✓
RGB16I	RGB	i16	i16	i16			✓		✓
RGB16UI	RGB	ui16	ui16	ui16			✓		✓
RGB32I	RGB	i32	i32	i32			✓		✓
RGB32UI	RGB	ui32	ui32	ui32			✓		✓
RGBA8I	RGBA	i8	i8	i8	i8		✓	✓	✓
RGBA8UI	RGBA	ui8	ui8	ui8	ui8		✓	✓	✓
RGBA16I	RGBA	i16	i16	i16	i16		✓	✓	✓
RGBA16UI	RGBA	ui16	ui16	ui16	ui16		✓	✓	✓
RGBA32I	RGBA	i32	i32	i32	i32		✓	✓	✓
RGBA32UI	RGBA	ui32	ui32	ui32	ui32		✓	✓	✓

Table 8.19: Correspondence of sized internal color formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type of the entry. *i* is signed integer, *ui* is unsigned integer, *s* is signed normalized fixed-point, and no prefix is unsigned normalized fixed-point. The "CR", "Req. tex.", and "Req. rend." columns are described in sections 9.4, 8.5.1, and 9.2.5, respectively.

Sized Internal Format	Base Internal Format	*A* bits	*L* bits	*I* bits
LUMINANCE4	LUMINANCE		4	
LUMINANCE8	LUMINANCE		8	
LUMINANCE12	LUMINANCE		12	
LUMINANCE16	LUMINANCE		16	
LUMINANCE4_ALPHA4	LUMINANCE_ALPHA	4	4	
LUMINANCE6_ALPHA2	LUMINANCE_ALPHA	2	6	
LUMINANCE8_ALPHA8	LUMINANCE_ALPHA	8	8	
LUMINANCE12_ALPHA4	LUMINANCE_ALPHA	4	12	
LUMINANCE12_ALPHA12	LUMINANCE_ALPHA	12	12	
LUMINANCE16_ALPHA16	LUMINANCE_ALPHA	16	16	
INTENSITY4	INTENSITY			4
INTENSITY8	INTENSITY			8
INTENSITY12	INTENSITY			12
INTENSITY16	INTENSITY			16
SLUMINANCE	LUMINANCE		8	
SLUMINANCE8_ALPHA8	LUMINANCE_ALPHA	8	8	

Table 8.20: Correspondence of sized internal luminance and intensity formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating-point, *i* is signed integer, *ui* is unsigned integer, and no prefix is fixed-point.

If a compressed internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 8.18. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any **TexImage3D**, **TexImage2D** (see below), or **TexImage1D** (see below) parameter (except *target*), but the allocation and chosen compressed image format must not be a function of any other state and cannot be changed once they are established. In addition, the choice of a compressed

Sized Internal Format	Base Internal Format	D bits	S bits	Req. format
DEPTH_COMPONENT16	DEPTH_COMPONENT	16		✓
DEPTH_COMPONENT24	DEPTH_COMPONENT	24		✓
DEPTH_COMPONENT32	DEPTH_COMPONENT	32		
DEPTH_COMPONENT32F	DEPTH_COMPONENT	f32		✓
DEPTH24_STENCIL8	DEPTH_STENCIL	24	ui8	✓
DEPTH32F_STENCIL8	DEPTH_STENCIL	f32	ui8	✓
STENCIL_INDEX1	STENCIL_INDEX		ui1	
STENCIL_INDEX4	STENCIL_INDEX		ui4	
STENCIL_INDEX8	STENCIL_INDEX		ui8	✓
STENCIL_INDEX16	STENCIL_INDEX		ui16	

Table 8.21: Correspondence of sized internal depth and stencil formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating-point, *i* is signed integer, *ui* is unsigned integer, and no prefix is fixed-point. The "Req. format" column is described in section 8.5.1.

image format may not be affected by the *data* parameter. Allocations must be invariant; the same allocation and compressed image format must be chosen each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 8.22.

8.5.3 Texture Image Structure

The texture image itself (referred to by *data*) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top forming a single two-dimensional image slice; and *depth* slices are stacked from back to front. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a texel as described by table 8.18. Counting from zero, each resulting nth texel is assigned internal integer coordinates (i, j, k), where

$$i = (n \bmod width) - w_b$$
$$j = \left(\left\lfloor \frac{n}{width} \right\rfloor \bmod height\right) - h_b$$

Compressed Internal Format	Base Internal Format	Type	Border Type	Copyable
COMPRESSED_ALPHA	ALPHA	Generic	unorm	✓
COMPRESSED_LUMINANCE	LUMINANCE	Generic	unorm	✓
COMPRESSED_LUMINANCE_ALPHA	LUMINANCE_ALPHA	Generic	unorm	✓
COMPRESSED_INTENSITY	INTENSITY	Generic	unorm	✓
COMPRESSED_RED	RED	Generic	unorm	✓
COMPRESSED_RG	RG	Generic	unorm	✓
COMPRESSED_RGB	RGB	Generic	unorm	✓
COMPRESSED_RGBA	RGBA	Generic	unorm	✓
COMPRESSED_SRGB	RGB	Generic	unorm	✓
COMPRESSED_SRGB_ALPHA	RGBA	Generic	unorm	✓
COMPRESSED_SLUMINANCE	LUMINANCE	Generic	unorm	✓
COMPRESSED_SLUMINANCE_ALPHA	LUMINANCE_ALPHA	Generic	unorm	✓
COMPRESSED_RED_RGTC1	RED	Specific	unorm	✓
COMPRESSED_SIGNED_RED_RGTC1	RED	Specific	snorm	✓
COMPRESSED_RG_RGTC2	RG	Specific	unorm	✓
COMPRESSED_SIGNED_RG_RGTC2	RG	Specific	snorm	✓
COMPRESSED_RGBA_BPTC_UNORM	RGBA	Specific	unorm	✓
COMPRESSED_SRGB_ALPHA_BPTC_UNORM	RGBA	Specific	unorm	✓
COMPRESSED_RGB_BPTC_SIGNED_FLOAT	RGB	Specific	float	✓
COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT	RGB	Specific	float	✓
COMPRESSED_RGB8_ETC2	RGB	Specific	unorm	
COMPRESSED_SRGB8_ETC2	RGB	Specific	unorm	
COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2	RGB	Specific	unorm	
COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2	RGB	Specific	unorm	
COMPRESSED_RGBA8_ETC2_EAC	RGBA	Specific	unorm	
COMPRESSED_SRGB8_ALPHA8_ETC2_EAC	RGBA	Specific	unorm	
COMPRESSED_R11_EAC	RED	Specific	unorm	
COMPRESSED_SIGNED_R11_EAC	RED	Specific	snorm	
COMPRESSED_RG11_EAC	RG	Specific	unorm	
COMPRESSED_SIGNED_RG11_EAC	RG	Specific	snorm	

Table 8.22: Generic and specific compressed internal formats. Specific formats are described in appendix D. The "Border Type" field determines how border colors are clamped, as described in section 8.14.2. The "Copyable" field specifies if the format may be used with non-compressed texture image commands.

$$k = \left(\left\lfloor \frac{n}{width \times height} \right\rfloor \bmod depth \right) - d_b$$

and w_b, h_b, and d_b are the specified border width, height, and depth. w_b and h_b are the specified *border* value; d_b is the specified *border* value if *target* is `TEXTURE_3D`, or zero if *target* is `TEXTURE_2D_ARRAY` or `TEXTURE_CUBE_MAP_ARRAY`. Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of k.

When *target* is `TEXTURE_CUBE_MAP_ARRAY`, specifying a cube map array texture, k refers to a *layer-face*. The layer is given by

$$layer = \left\lfloor \frac{k}{6} \right\rfloor,$$

and the face is given by

$$face = k \bmod 6.$$

The face number corresponds to the cube map faces as shown in table 9.3.

If the internal data type of the texture image is signed or unsigned normalized fixed-point, each color component is converted using equation 2.4 or 2.3, respectively. If the internal type is floating-point or integer, components are clamped to the representable range of the corresponding internal component, but are not converted.

The *level* argument to **TexImage3D** is an integer *level-of-detail* number. Levels of detail are discussed in section 8.14.3. The main texture image has a level-of-detail number of zero. *level* must be zero or more.

border is a border width; the maximum border width b_t is 1. The significance of borders is described below. *border* affects the dimensions of the texture image: let

$$\begin{aligned} w_t &= w_s - 2w_b \\ h_t &= h_s - 2h_b \\ d_t &= d_s - 2d_b \end{aligned} \tag{8.3}$$

where w_s, h_s, and d_s are the specified image *width*, *height*, and *depth*, and w_t, h_t, and d_t are the dimensions of the texture image internal to the border.

An image with zero width, height, or depth indicates the null texture. If the null texture is specified for the level-of-detail specified by texture parameter `TEXTURE_BASE_LEVEL` (see section 8.10), it is as if texturing were disabled.

The maximum allowable size, in any relevant dimension, of a texture image is an implementation-dependent function of the texture target, the level-of-detail,

and the internal format of the image. For most texture types, it must satisfy the relationship

$$maxsize \geq 2^{k-level}+2b_t \tag{8.4}$$

for images of level-of-detail ($level$) 0 through k, where b_t is the maximum border width, and k is a texture target-dependent maximum level-of-detail. The maximum size may be zero for any images where $level > k$.

The maximum allowable width, height, or depth of a texture image for a three-dimensional texture is determined by equation 8.4, where k is log_2 of the value of `MAX_3D_TEXTURE_SIZE`.

In a similar fashion, the maximum allowable width, or height for two-dimensional texture types, of a texture image for a one- or two-dimensional, one- or two-dimensional array, two-dimensional multisample, or two-dimensional multisample array texture is determined by equation 8.4, where k is log_2 of the value of `MAX_TEXTURE_SIZE`.

The maximum allowable width or height of a cube map or cube map array texture image must be the same, and is determined by equation 8.4, where k is log_2 of the value of `MAX_CUBE_MAP_TEXTURE_SIZE`.

The maximum number of layers for one- and two-dimensional array textures (height or depth, respectively), or the maximum number of layer-faces for cube map array textures (depth), must be at least the value of `MAX_ARRAY_TEXTURE_LAYERS` for all levels.

The maximum allowable width or height of a rectangle texture image must each be at least the value of the implementation-dependent constant `MAX_RECTANGLE_TEXTURE_SIZE`.

As described in section 8.17, these implementation-dependent limits may be configured to reject textures at level one or greater unless a mipmap complete set of texture images consistent with the specified sizes can be supported.

Regardless of the values of these implementation-dependent constants, an implementation may not succeed in creating a texture of the maximum sizes due to resource limits, resulting in memory exhaustion.

Errors

An `INVALID_ENUM` error is generated if *target* is not one of the valid targets listed for each **TexImage*D** command.

An `INVALID_VALUE` error is generated if *level* is negative.

An `INVALID_VALUE` error is generated if *width*, *height*, or *depth* (if each argument is present) is negative.

An INVALID_VALUE error is generated if *width*, *height*, or *depth* (if each argument is present) exceed the corresponding *target*-dependent maximum size, as described above.

An INVALID_VALUE error is generated if w_t, h_t, or d_t are negative.

An INVALID_VALUE error is generated by **TexImage3D** if *target* is TEXTURE_CUBE_MAP_ARRAY or PROXY_TEXTURE_CUBE_MAP_ARRAY, and *width* and *height* are not equal, or if *depth* is not a multiple of six, indicating $6N$ layer-faces in the cube map array.

An INVALID_VALUE error is generated by **TexImage2D** if *target* is one of the cube map face targets from table 8.27, and *width* and *height* are not equal.

An INVALID_VALUE error is generated by **TexImage2D** if *target* is TEXTURE_RECTANGLE and *level* is non-zero.

An INVALID_VALUE error is generated if *border* is negative or greater than b_t.

An INVALID_VALUE error is generated by **TexImage2D** if *target* is TEXTURE_RECTANGLE and *border* is non-zero.

An INVALID_VALUE error is generated if *internalformat* is not one of the valid values described above.

An INVALID_OPERATION error is generated if *internalformat* is one of the formats in table 8.22 and the "Copyable" field in the table is not ✓.

An INVALID_OPERATION error is generated if the internal format is integer and *format* is not one of the integer formats listed in table 8.8, or if the internal format is not integer and *format* is an integer format.

An INVALID_OPERATION error is generated by **TexImage3D** if *internalformat* is one of the EAC, ETC2, or RGTC compressed formats and either *border* is non-zero, or *target* is not TEXTURE_2D_ARRAY.

An INVALID_OPERATION error is generated by **TexImage2D** if *internalformat* is one of the EAC, ETC2, or RGTC compressed formats and either *border* is non-zero, or *target* is not TEXTURE_2D or one of the cube map face targets from table 8.27.

An INVALID_ENUM error is generated by **CompressedTexImage1D** if *internalformat* is one of the specific compressed formats. OpenGL defines no specific one-dimensional compressed formats, but such formats may be provided by extensions.

An INVALID_OPERATION error is generated if one of the base internal format and *format* is DEPTH_COMPONENT or DEPTH_STENCIL, and the other is neither of these values.

An INVALID_OPERATION error is generated if either the base internal format or *format* is STENCIL_INDEX and the other is not also STENCIL_–

INDEX.

An INVALID_OPERATION error is generated if a pixel unpack buffer object is bound and storing texture data would access memory beyond the end of the pixel unpack buffer.

The command

```
void TexImage2D( enum target, int level, int internalformat,
    sizei width, sizei height, int border, enum format,
    enum type, const void *data );
```

is used to specify a two-dimensional texture image. *target* must be one of TEXTURE_2D for a two-dimensional texture, TEXTURE_1D_ARRAY for a one-dimensional array texture, TEXTURE_RECTANGLE for a rectangle texture, or one of the cube map face targets from table 8.27 for a cube map texture. Additionally, *target* may be either PROXY_TEXTURE_2D for a two-dimensional proxy texture, PROXY_TEXTURE_1D_ARRAY for a one-dimensional proxy array texture, PROXY_TEXTURE_RECTANGLE for a rectangle proxy texture, or PROXY_TEXTURE_CUBE_MAP for a cube map proxy texture in the special case discussed in section 8.22. The other parameters match the corresponding parameters of **TexImage3D**.

For the purposes of decoding the texture image, **TexImage2D** is equivalent to calling **TexImage3D** with corresponding arguments and *depth* of 1, except that

- The border depth, d_b, is zero, and the *depth* of the image is always one regardless of the value of *border*.
- The border height, h_b, is zero if *target* is TEXTURE_1D_ARRAY, and *border* otherwise.
- Convolution will be performed on the image (possibly changing its *width* and *height*) if SEPARABLE_2D or CONVOLUTION_2D is enabled.
- UNPACK_SKIP_IMAGES is ignored.

A two-dimensional or rectangle texture consists of a single two-dimensional texture image. A cube map texture is a set of six two-dimensional texture images. The six cube map texture face targets from table 8.27 form a single cube map texture. These targets each update the corresponding cube map face two-dimensional texture image. Note that the cube map face targets are used when specifying, updating, or querying one of a cube map's six two-dimensional images, but when enabling cube map texturing or binding to a cube map texture object (that is when

the cube map is accessed as a whole as opposed to a particular two-dimensional image), the TEXTURE_CUBE_MAP target is specified.

Finally, the command

```
void TexImage1D( enum target, int level,
    int internalformat, sizei width, int border,
    enum format, enum type, const void *data );
```

is used to specify a one-dimensional texture image. *target* must be either TEXTURE_1D, or PROXY_TEXTURE_1D in the special case discussed in section 8.22.

For the purposes of decoding the texture image, **TexImage1D** is equivalent to calling **TexImage2D** with corresponding arguments and *height* of 1, except that

- The border height and depth (h_b and d_b) are always zero, regardless of the value of *border*.
- Convolution will be performed on the image (possibly changing its *width*) only if CONVOLUTION_1D is enabled.

The image indicated to the GL by the image pointer is decoded and copied into the GL's internal memory. This copying effectively places the decoded image inside a border of the maximum allowable width b_t whether or not a border has been specified (see figure 8.3). If no border, or a border smaller than the maximum allowable width has been specified, then the image is still stored as if it were surrounded by a border of the maximum possible width. Any excess border (which surrounds the specified image, including any border) is assigned unspecified values. A two-dimensional texture has a border only at its left, right, top, and bottom ends, and a one-dimensional texture has a border only at its left and right ends.

We shall refer to the (possibly border augmented) decoded image as the *texture image*. A three-dimensional texture image has width, height, and depth w_s, h_s, and d_s as defined in section 8.5.3. A two-dimensional texture image has depth $d_s = 1$, with height h_s and width w_s as above. A rectangle texture image must have zero border width, so w_s and h_s equal the specified *width* and *height*, respectively, while $d_s = 1$. A one-dimensional texture image has depth $d_s = 1$, height $h_s = 1$, and width w_s as above.

An element (i, j, k) of the texture image is called a *texel* (for a two-dimensional texture or one-dimensional array texture, k is irrelevant; for a one-dimensional texture, j and k are both irrelevant). The *texture value* used in texturing a fragment is determined by that fragment's associated (s, t, r) coordinates in fixed-function fragment shading, or by sampling the texture in a shader, but may not correspond

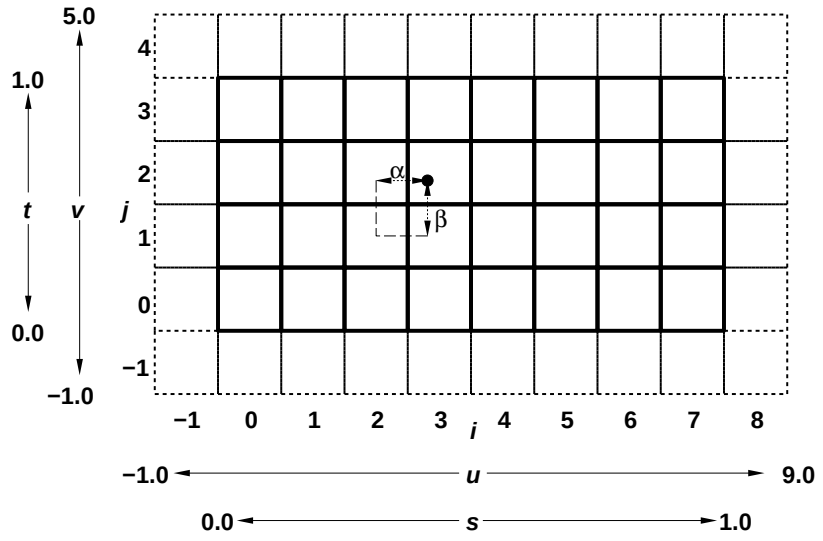


Figure 8.3. A texture image and the coordinates used to access it. This is a two-dimensional texture with *width* 8 and *height* 4. A one-dimensional texture would consist of a single horizontal strip. α and β, values used in blending adjacent texels to obtain a texture value are also shown.

to any actual texel. See figure 8.3. If target is `TEXTURE_CUBE_MAP_ARRAY`, the texture value is determined by (s, t, r, q) coordinates where s, t, and r are defined to be the same as for `TEXTURE_CUBE_MAP` and q is defined as the index of a specific cube map in the cube map array.

If the *data* argument of **TexImage1D**, **TexImage2D**, or **TexImage3D** is `NULL`, and the pixel unpack buffer object is zero, a one-, two-, or three-dimensional texture image is created with the specified *target*, *level*, *internalformat*, *border*, *width*, *height*, and *depth*, but with unspecified image contents. In this case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the *data* pointer were valid. Otherwise if the pixel unpack buffer object is non-zero, the *data* argument is treated normally to refer to the beginning of the pixel unpack buffer object's data.

8.6 Alternate Texture Image Specification Commands

Two-dimensional and one-dimensional texture images may also be specified using image data taken directly from the framebuffer, and rectangular subregions of existing texture images may be respecified.

The command

```
void CopyTexImage2D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    sizei height, int border );
```

defines a two-dimensional texture image in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. *target* must be one of `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_RECTANGLE`, or one of the cube map face targets from table 8.27. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments to **ReadPixels** (refer to section 18.2); they specify the image's *width* and *height*, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** (see section 18.3) with argument *type* set to `COLOR`, `DEPTH`, `DEPTH_STENCIL`, or `STENCIL_INDEX`, depending on *internalformat*, stopping after pixel transfer processing is complete. RGBA data is taken from the current color buffer, while depth component and stencil index data are taken from the depth and stencil buffers, respectively.

Subsequent processing is identical to that described for **TexImage2D**, beginning with clamping of the R, G, B, A, or depth values, and masking of the stencil index values from the resulting pixel groups. Parameters *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the corresponding arguments of **TexImage2D**, except that *internalformat* may not be specified as 1, 2, 3, or 4.

The constraints on *width*, *height*, and *border* are exactly those for the corresponding arguments of **TexImage2D**.

Errors

An `INVALID_ENUM` error is generated if *target* is not `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_RECTANGLE`, or one of the cube map face targets from table 8.27.

An `INVALID_ENUM` error is generated if an invalid value is specified for *internalformat*.

An INVALID_OPERATION error is generated if *internalformat* is one of the formats in table 8.22 and the "Copyable" field in the table is not ✓.

An INVALID_VALUE error is generated if the *target* parameter to **CopyTexImage2D** is one of the six cube map two-dimensional image targets, and *width* and *height* are not equal.

An INVALID_OPERATION error is generated under any of the following conditions:

- if depth component data is required and no depth buffer is present
- if stencil index data is required and no stencil buffer is present
- if integer RGBA data is required and the format of the current color buffer is not integer
- if floating- or fixed-point RGBA data is required and the format of the current color buffer is integer
- if the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the read buffer (see section 18.2.1) is LINEAR (see section 9.2.3) and *internalformat* is one of the sRGB formats in table 8.32
- if the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the read buffer is SRGB and *internalformat* is not one of the sRGB formats in table 8.32.

An INVALID_VALUE error is generated if *width* or *height* is negative.

The command

```
void CopyTexImage1D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    int border );
```

defines a one-dimensional texture image in exactly the manner of **TexImage1D**, except that the image data are taken from the framebuffer, rather than from client memory. Currently, *target* must be TEXTURE_1D. For the purposes of decoding the texture image, **CopyTexImage1D** is equivalent to calling **CopyTexImage2D** with corresponding arguments and *height* of 1, except that the *height* of the image is always 1, regardless of the value of *border*. *level*, *internalformat* and *border* are specified using the same values, with the same meanings, as the corresponding

arguments of **TexImage1D**, except that *internalformat* may not be specified as 1, 2, 3, or 4. The constraints on *width* and *border* are exactly those of the corresponding arguments of **TexImage1D**.

To respecify only a rectangular subregion of the texture image of a texture object, use the commands

`void` **TexSubImage3D**(`enum` *target*, `int` *level*, `int` *xoffset*, `int` *yoffset*, `int` *zoffset*, `sizei` *width*, `sizei` *height*, `sizei` *depth*, `enum` *format*, `enum` *type*, `const void` **data*);

`void` **TexSubImage2D**(`enum` *target*, `int` *level*, `int` *xoffset*, `int` *yoffset*, `sizei` *width*, `sizei` *height*, `enum` *format*, `enum` *type*, `const void` **data*);

`void` **TexSubImage1D**(`enum` *target*, `int` *level*, `int` *xoffset*, `sizei` *width*, `enum` *format*, `enum` *type*, `const void` **data*);

`void` **CopyTexSubImage3D**(`enum` *target*, `int` *level*, `int` *xoffset*, `int` *yoffset*, `int` *zoffset*, `int` *x*, `int` *y*, `sizei` *width*, `sizei` *height*);

`void` **CopyTexSubImage2D**(`enum` *target*, `int` *level*, `int` *xoffset*, `int` *yoffset*, `int` *x*, `int` *y*, `sizei` *width*, `sizei` *height*);

`void` **CopyTexSubImage1D**(`enum` *target*, `int` *level*, `int` *xoffset*, `int` *x*, `int` *y*, `sizei` *width*);

`void` **TextureSubImage3D**(`uint` *texture*, `int` *level*, `int` *xoffset*, `int` *yoffset*, `int` *zoffset*, `sizei` *width*, `sizei` *height*, `sizei` *depth*, `enum` *format*, `enum` *type*, `const void` **pixels*);

`void` **TextureSubImage2D**(`uint` *texture*, `int` *level*, `int` *xoffset*, `int` *yoffset*, `sizei` *width*, `sizei` *height*, `enum` *format*, `enum` *type*, `const void` **pixels*);

`void` **TextureSubImage1D**(`uint` *texture*, `int` *level*, `int` *xoffset*, `sizei` *width*, `enum` *format*, `enum` *type*, `const void` **pixels*);

`void` **CopyTextureSubImage3D**(`uint` *texture*, `int` *level*, `int` *xoffset*, `int` *yoffset*, `int` *zoffset*, `int` *x*, `int` *y*, `sizei` *width*, `sizei` *height*);

`void` **CopyTextureSubImage2D**(`uint` *texture*, `int` *level*, `int` *xoffset*, `int` *yoffset*, `int` *x*, `int` *y*, `sizei` *width*, `sizei` *height*);

Command Names	Valid *target*s or effective *texture* targets
TexSubImage1D, **CopyTexSubImage1D**, **TextureSubImage1D**, **CopyTextureSubImage1D**	TEXTURE_1D
TexSubImage2D, **CopyTexSubImage2D**,	TEXTURE_2D, TEXTURE_1D_ARRAY, TEXTURE_RECTANGLE or one of the cube map face targets from table 8.27
TextureSubImage2D, **CopyTextureSubImage2D**	TEXTURE_2D, TEXTURE_1D_ARRAY or TEXTURE_RECTANGLE
TexSubImage3D, **CopyTexSubImage3D**,	TEXTURE_3D, TEXTURE_2D_ARRAY or TEXTURE_CUBE_MAP_ARRAY
TextureSubImage3D, **CopyTextureSubImage3D**	TEXTURE_3D, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY or TEXTURE_CUBE_MAP

Table 8.23: Valid texture *target* parameters or effective *texture* targets for texture subimage commands.

```
void CopyTextureSubImage1D( uint texture, int level,
    int xoffset, int x, int y, sizei width );
```

For ***TexSubImage***, the texture object is that bound to *target*, For ***TextureSubImage***, *texture* is the name of the texture object. *target* or the effective target of *texture* (the value of TEXTURE_TARGET; see section 8.11.2) must match each command as shown in table 8.23.

No change is made to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texture image, nor is any change made to texel values outside the specified subregion.

The *level* parameter of each command specifies the level of the texture image that is modified.

TexSubImage*D and **TextureSubImage*D** arguments *width*, *height*, *depth*,

format, *type*, and *data* match the corresponding arguments to the corresponding[3] **TexImage*D** command (where those arguments exist), meaning that they accept the same values, and have the same meanings. The exception is that a `NULL` *data* pointer does not represent unspecified image contents.

CopyTex*SubImage3D and **CopyTex*SubImage2D** arguments *x*, *y*, *width*, and *height* match the corresponding arguments to **CopyTexImage2D**. **CopyTex*SubImage1D** arguments *x*, *y*, and *width* match the corresponding arguments to **CopyTexImage1D**.

Each of these commands interprets and processes pixel groups in exactly the manner of its **TexImage** counterpart, except that the assignment of R, G, B, A, depth, and stencil index pixel group values to the texture components is controlled by the *internalformat* of the texture image, not by an argument to the command. The same constraints and errors apply to the *format* argument of these commands and the *internalformat* of the texture image being respecified as apply to the *format* and *internalformat* arguments of their **TexImage** counterparts.

Arguments *xoffset*, *yoffset*, and *zoffset* of **Tex*SubImage3D** and **CopyTex*SubImage3D** specify the lower left back texel coordinates of a *width*-wide by *height*-high by *depth*-deep rectangular subregion of the texture image. For cube map array textures, *zoffset* is the first layer-face to update, and *depth* is the number of layer-faces to update. For **TextureSubImage3D** and **CopyTextureSubImage3D** only, *texture* may be a cube map texture. In this case, *zoffset* is interpreted as specifying the cube map face for the corresponding layer in table 9.3 and *depth* is the number of successive faces to update.

The *depth* argument associated with **CopyTex*SubImage3D** is always 1, because framebuffer memory is two-dimensional - only a portion of a single (s, t) slice of a three-dimensional texture is replaced by these commands.

The subregion must lie within the bounds of the texture image. If w_s, h_s, and d_s are the specified width, height, and depth of the texture image (including border width) and w_b, h_b, and d_b are the border width, height, and depth (as described in section 8.5.3); x, y, and z are the specified *xoffset*, *yoffset*, and *zoffset* values; and w, h, and d are the specified *width*, *height*, and *depth* values; then it is an error if any of the relationships in equations 8.5-8.7 are satisfied.

$$x < -w_b, \qquad x + w > w_s - w_b \tag{8.5}$$

$$y < -h_b, \qquad y + h > h_s - h_b \tag{8.6}$$

$$z < -d_b, \qquad z + d > d_s - d_b \tag{8.7}$$

[3] For example, both **TexSubImage2D** and **TextureSubImage2D** correspond to **TexImage2D** for purposes of this paragraph.

Negative values of *xoffset*, *yoffset*, and *zoffset* correspond to the coordinates of border texels, addressed as in figure 8.3.

Counting from zero, the nth pixel group is assigned to the texel with internal integer coordinates $[i, j, k]$, where

$$i = x + (n \bmod w)$$
$$j = y + \left(\left\lfloor \frac{n}{w} \right\rfloor \bmod h\right)$$
$$k = z + \left(\left\lfloor \frac{n}{w \times h} \right\rfloor \bmod d\right)$$

Arguments *xoffset* and *yoffset* of **Tex*SubImage2D** and **CopyTex*SubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texture image.

The subregion must lie within the bounds of the texture image, as described above for **TexSubImage3D**. It is an error if any of the relationships in equations 8.5-8.6 are satisfied.

Counting from zero, the nth pixel group is assigned to the texel with internal integer coordinates $[i, j]$, where

$$i = x + (n \bmod w)$$
$$j = y + (\left\lfloor \frac{n}{w} \right\rfloor \bmod h)$$

The *xoffset* argument of **Tex*SubImage1D** and **CopyTex*SubImage1D** specifies the left texel coordinate of a *width*-wide subregion of the texture image.

The subregion must lie within the bounds of the texture image, as described above for **TexSubImage3D**. It is an error if any of the relationships in equation 8.5 are satisfied.

Counting from zero, the nth pixel group is assigned to the texel with internal integer coordinates $[i]$, where

$$i = x + (n \bmod w)$$

Texture images with compressed internal formats may be stored in such a way that it is not possible to modify an image with subimage commands without having to decompress and recompress the texture image. Even if the image were modified in this manner, it may not be possible to preserve the contents of some of the texels outside the region being modified. To avoid these complications, the GL does not support arbitrary modifications to texture images with compressed internal formats. Calling any of the **Tex*SubImage*** or **CopyTex*SubImage*** commands

will generate an INVALID_OPERATION error if *xoffset*, *yoffset*, or *zoffset* is not equal to $-b_s$ (border width). In addition, the contents of any texel outside the region modified by such a call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

If the internal format of the texture image being modified is one of the specific compressed formats described in table 8.22, the texture is stored using the corresponding compressed texture image encoding (see appendix D). Since such images are easily edited along 4×4 texel boundaries, the limitations on subimage location and size are relaxed for **Tex*SubImage2D**, **Tex*SubImage3D**, **CopyTex*SubImage2D**, and **CopyTex*SubImage3D**. These commands will generate an INVALID_OPERATION error if one of the following conditions occurs:

- *width* is not a multiple of four, $width + xoffset$ is not equal to the value of TEXTURE_WIDTH, and either *xoffset* or *yoffset* is non-zero.
- *height* is not a multiple of four, $height + yoffset$ is not equal to the value of TEXTURE_HEIGHT, and either *xoffset* or *yoffset* is non-zero.
- *xoffset* or *yoffset* is not a multiple of four.

The contents of any 4×4 block of texels of such a compressed texture image that does not intersect the area being modified are preserved during valid **Tex*SubImage*** and **Copy*TexSubImage*** calls.

Errors

An INVALID_ENUM error is generated by ***TexSubImage*** if *target* does not match the command, as shown in table 8.23.

An INVALID_OPERATION error is generated by ***TextureSubImage*** if *texture* is not the name of an existing texture object.

An INVALID_OPERATION error is generated by ***TextureSubImage*** if the effective target of *texture* does not match the command, as shown in table 8.23.

An INVALID_OPERATION error is generated by:

- ***TextureSubImage3D** if the effective *target* is TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY

and the texture object is not cube complete (for TEXTURE_CUBE_MAP or cube array complete (for TEXTURE_CUBE_MAP_ARRAY).

An INVALID_VALUE error is generated if *width*, *height*, or *depth* (if each argument is present) is negative.

An INVALID_VALUE error is generated if *level* is negative or greater than the log_2 of the maximum texture width, height, or depth.

An INVALID_VALUE error is generated if the effective *target* is TEXTURE_RECTANGLE and *level* is not zero.

An INVALID_VALUE error is generated if the specified subregion does not lie within the bounds of the texture image, as described above.

An INVALID_FRAMEBUFFER_OPERATION error is generated by **CopyTexImage*D**, **CopyTexSubImage*D** and **CopyTextureSubImage*D** if the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer complete (see section 9.4.2)

An INVALID_OPERATION error is generated by **CopyTexImage*D**, **CopyTexSubImage*D** and **CopyTextureSubImage*D** if

- the read buffer is NONE, or
- the value of READ_FRAMEBUFFER_BINDING is non-zero, and
  - the read buffer selects an attachment that has no image attached, or
  - the effective value of SAMPLE_BUFFERS for the read framebuffer (see section 9.2.3.1) is one.

An INVALID_OPERATION error is generated by ***TexSubImage2D** and ***TextureSubImage2D** if *internalformat* is one of the EAC, ETC2, or RGTC formats and the effective *target* is TEXTURE_RECTANGLE.

8.6.1 Texture Copying Feedback Loops

Calling any of the **CopyTex*SubImage*** or **CopyTexImage*** commands will result in undefined behavior if the destination texture image level is also bound to to the selected read buffer (see section 18.2.1) of the read framebuffer. This situation is discussed in more detail in the description of feedback loops in section 9.3.2.

8.7 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format, including the formats defined in appendix D as well as any additional formats defined by extensions.

The commands

```
void CompressedTexImage1D( enum target, int level,
    enum internalformat, sizei width, int border,
    sizei imageSize, const void *data );
void CompressedTexImage2D( enum target, int level,
    enum internalformat, sizei width, sizei height,
    int border, sizei imageSize, const void *data );
void CompressedTexImage3D( enum target, int level,
    enum internalformat, sizei width, sizei height,
    sizei depth, int border, sizei imageSize, const
    void *data );
```

define one-, two-, and three-dimensional texture images, respectively, with incoming data stored in a specific compressed image format. The *target*, *level*, *internalformat*, *width*, *height*, *depth*, and *border* parameters have the same meaning as in **TexImage1D**, **TexImage2D**, and **TexImage3D**, except that compressed rectangle texture formats are not supported. *data* refers to compressed image data stored in the specific compressed image format corresponding to *internalformat*. If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING), *data* is an offset into the pixel unpack buffer and the compressed data is read from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the compressed data is read from client memory relative to the pointer.

The compressed image will be decoded according to the specification defining the *internalformat* token. Compressed texture images are treated as an array of *imageSize* ubytes relative to *data*.

If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

All pixel transfer modes are ignored when decoding a compressed texture image.

If the compressed data are arranged into fixed-size blocks of texels, the pixel storage modes can be used to select a sub-rectangle from a larger containing rectangle. These pixel storage modes operate in the same way as they do for **TexImage*D** and as described in section 8.4.4. In the remainder of this section, denote by $blocksize$, b_w, b_h, and b_d the values of pixel storage modes UNPACK_COMPRESSED_BLOCK_SIZE, UNPACK_COMPRESSED_BLOCK_WIDTH, UNPACK_COMPRESSED_BLOCK_HEIGHT, and UNPACK_COMPRESSED_BLOCK_DEPTH respectively. $blocksize$ is the compressed block size in bytes; b_w, b_h, and b_d are the compressed block width, height, and depth in pixels.

By default the pixel storage modes UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, UNPACK_SKIP_PIXELS, UNPACK_IMAGE_HEIGHT and UNPACK_SKIP_IMAGES are ignored for compressed images. To enable UNPACK_SKIP_PIXELS and UNPACK_ROW_LENGTH, $blocksize$ and b_w must both be non-zero. To also enable UNPACK_SKIP_ROWS and UNPACK_IMAGE_HEIGHT, b_h must be non-zero. And to also enable UNPACK_SKIP_IMAGES, b_d must be non-zero. All parameters must be consistent with the compressed format to produce the desired results.

Errors

An INVALID_ENUM error is generated if the *target* parameter to any of the **CompressedTexImage**n**D** commands is TEXTURE_RECTANGLE or PROXY_TEXTURE_RECTANGLE.

An INVALID_ENUM error is generated if *internalformat* is not a supported specific compressed internal format from table 8.22. In particular, this error will be generated for any of the generic compressed internal formats.

An INVALID_VALUE error is generated if *width*, *height*, *depth*, or *imageSize* is negative.

An INVALID_OPERATION error is generated if a pixel unpack buffer object is bound and $data + imageSize$ is greater than the size of the pixel buffer.

An INVALID_VALUE error is generated if the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image.

An INVALID_OPERATION error is generated if any of the following conditions is violated when selecting a sub-rectangle from a compressed image:

- the value of UNPACK_SKIP_PIXELS must be a multiple of b_w;
- the value of UNPACK_SKIP_ROWS must be a multiple of b_h for **CompressedTexImage2D** and **CompressedTexImage3D**;
- the value of UNPACK_SKIP_IMAGES must be a multiple of b_d for **CompressedTexImage3D**.

An INVALID_VALUE error is generated if *imageSize* does not match the following requirements when pixel storage modes are active:

- For **CompressedTexImage1D** the *imageSize* parameter must be equal to

$$blocksize \times \left\lceil \frac{width}{b_w} \right\rceil$$

- For **CompressedTexImage2D** the *imageSize* parameter must be equal to

$$blocksize \times \left\lceil \frac{width}{b_w} \right\rceil \times \left\lceil \frac{height}{b_h} \right\rceil$$

- For **CompressedTexImage3D** the *imageSize* parameter must be equal to

$$blocksize \times \left\lceil \frac{width}{b_w} \right\rceil \times \left\lceil \frac{height}{b_h} \right\rceil \times \left\lceil \frac{depth}{b_d} \right\rceil$$

Based on the definition of unpacking from section 8.4.4 for uncompressed images, unpacking compressed images can be defined where:

- n, the number of elements in a group, is 1.
- s, the size of an element, is $blocksize$.
- l, the number of groups in a row, is

$$l = \begin{cases} \left\lceil \frac{row_length}{b_w} \right\rceil, & row_length > 0 \\ \left\lceil \frac{width}{b_w} \right\rceil, & otherwise \end{cases}$$

  where row_length is the value of `UNPACK_ROW_LENGTH`.
- a, the value of `UNPACK_ALIGNMENT`, is ignored.
- $k = n \times l$ as is defined for uncompressed images.

Before obtaining the first compressed image block from memory, the *data* pointer is advanced by

$$\frac{\texttt{UNPACK_SKIP_PIXELS}}{b_w} \times n + \frac{\texttt{UNPACK_SKIP_ROWS}}{b_h} \times k$$

elements. Then $\left\lceil \frac{width}{b_w} \right\rceil$ blocks are obtained from contiguous blocks in memory (without advancing the pointer), after which the pointer is advanced by k elements.

$\left\lceil \frac{height}{b_h} \right\rceil$ sets of $\left\lceil \frac{width}{b_w} \right\rceil$ blocks are obtained this way. For three-dimensional compressed images the pointer is advanced by $\frac{\texttt{UNPACK_SKIP_IMAGES}}{b_d}$ times the number of elements in one two-dimensional image before obtaining the first group from memory. Then after *height* rows are obtained, if UNPACK_IMAGE_HEIGHT is positive, the pointer skips over the remaining $\left\lceil \frac{\texttt{UNPACK_IMAGE_HEIGHT} - height}{b_h} \right\rceil$ rows, before starting the next two-dimensional image.

Specific compressed internal formats may impose additional format-specific restrictions. For example, a format might be supported only for two-dimensional textures, or might not allow non-zero *border* values. Any such restrictions will be documented in the extension specification defining the compressed internal format.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to **CompressedTexImage1D**, **CompressedTexImage2D**, or **CompressedTexImage3D** will not generate an error if the following restrictions are satisfied:

- *data* points to a compressed texture image returned by **GetCompressedTexImage** (section 8.11).
- *target*, *level*, and *internalformat* match the *target*, *level* and *format* parameters provided to the **GetCompressedTexImage** call returning *data*.
- *width*, *height*, *depth*, *border*, *internalformat*, and *imageSize* match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, TEXTURE_BORDER, TEXTURE_INTERNAL_FORMAT, and TEXTURE_COMPRESSED_IMAGE_SIZE for image level *level* in effect at the time of the **GetCompressedTexImage** call returning *data*.

This guarantee applies not just to images returned by **GetCompressedTexImage**, but also to any other properly encoded compressed texture image of the same size and format.

If *internalformat* is one of the specific compressed formats described in table 8.22, the compressed image data is stored using the corresponding texture image encoding (see appendix D). The corresponding compression algorithms support only two-dimensional images without borders, though three-dimensional images can be compressed as multiple slices of compressed two-dimensional BPTC images.

Errors

An INVALID_OPERATION error is generated if any format-specific restrictions imposed by specific compressed internal formats are violated by the compressed image specification calls or parameters.

An INVALID_ENUM error is generated by **CompressedTexImage1D** if *internalformat* is one of the specific compressed formats. OpenGL defines no specific one-dimensional compressed formats, but such formats may be provided by extensions.

An INVALID_OPERATION error is generated by **CompressedTexImage2D** if *internalformat* is one of the EAC, ETC2, or RGTC formats and *border* is non-zero.

An INVALID_OPERATION error is generated by **CompressedTexImage3D** if *internalformat* is one of the EAC, ETC2, or RGTC formats and either *border* is non-zero, or *target* is not TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY.

An INVALID_OPERATION error is generated by **CompressedTexImage2D** and **CompressedTexImage3D** if *internalformat* is one of the BPTC formats and *border* is non-zero.

If the *data* argument of **CompressedTexImage1D**, **CompressedTexImage2D**, or **CompressedTexImage3D** is NULL, and the pixel unpack buffer object is zero, a texture image with unspecified image contents is created, just as when a NULL pointer is passed to **TexImage1D**, **TexImage2D**, or **TexImage3D**.

To respecify only a rectangular subregion of the texture image of a texture object, with incoming data stored in a specific compressed image format, use the commands

void **CompressedTexSubImage1D**(enum *target*, int *level*, int *xoffset*, sizei *width*, enum *format*, sizei *imageSize*, const void *data);

void **CompressedTexSubImage2D**(enum *target*, int *level*, int *xoffset*, int *yoffset*, sizei *width*, sizei *height*, enum *format*, sizei *imageSize*, const void *data);

void **CompressedTexSubImage3D**(enum *target*, int *level*, int *xoffset*, int *yoffset*, int *zoffset*, sizei *width*, sizei *height*, sizei *depth*, enum *format*, sizei *imageSize*, const void *data);

void **CompressedTextureSubImage1D**(uint *texture*, int *level*, int *xoffset*, sizei *width*, enum *format*, sizei *imageSize*, const void *data);

```
void CompressedTextureSubImage2D( uint texture,
    int level, int xoffset, int yoffset, sizei width,
    sizei height, enum format, sizei imageSize, const
    void *data );
void CompressedTextureSubImage3D( uint texture,
    int level, int xoffset, int yoffset, int zoffset,
    sizei width, sizei height, sizei depth, enum format,
    sizei imageSize, const void *data );
```

The *target*, *texture*, *level*, *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* parameters have the same meaning as in the corresponding commands from section 8.6 without the **Compressed** prefix (where those parameters are present). *data* points to compressed image data stored in the compressed image format corresponding to *format*.

The image pointed to by *data* and the *imageSize* parameter are interpreted as though they were provided to **CompressedTexImage1D**, **CompressedTexImage2D**, and **CompressedTexImage3D**.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to **CompressedTex*Image*** will not generate an error if the following restrictions are satisfied:

- *data* points to a compressed texture image returned by **GetCompressedTexImage** (section 8.11).
- *target*, *level*, and *format* match the *target*, *level* and *format* parameters provided to the **GetCompressedTexImage** call returning *data*.
- *width*, *height*, *depth*, *format*, and *imageSize* match the values of `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_DEPTH`, `TEXTURE_INTERNAL_FORMAT`, and `TEXTURE_COMPRESSED_IMAGE_SIZE` for image level *level* in effect at the time of the **GetCompressedTexImage** call returning *data*.
- *width*, *height*, *depth*, and *format* match the values of `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_DEPTH`, and `TEXTURE_INTERNAL_FORMAT` currently in effect for image level *level*.
- *xoffset*, *yoffset*, and *zoffset* are all $-b$, where b is the value of `TEXTURE_BORDER` currently in effect for image level *level*.

This guarantee applies not just to images returned by **GetCompressedTexImage**, but also to any other properly encoded compressed texture image of the same size.

If the internal format of the image being modified is one of the specific compressed formats described in table 8.22, the texture is stored using the corresponding texture image encoding (see appendix D).

Since these specific compressed formats are easily edited along 4×4 texel boundaries, the limitations on subimage location and size are relaxed for **CompressedTex*SubImage2D** and **CompressedTex*SubImage3D**.

The contents of any 4×4 block of texels of a compressed texture image in these specific compressed formats that do not intersect the area being modified are preserved during **CompressedTex*SubImage*** calls.

Errors

An `INVALID_ENUM` error is generated by **CompressedTexSubImage*D** if *target* is `TEXTURE_RECTANGLE` or `PROXY_TEXTURE_RECTANGLE`.

An `INVALID_OPERATION` error is generated by **CompressedTexture-SubImage*D** if *texture* is not the name of an existing texture object.

An `INVALID_OPERATION` error is generated by **CompressedTexture-SubImage*D** if the effective *target* is `TEXTURE_RECTANGLE`.

An `INVALID_ENUM` error is generated if *format* is one of the generic compressed internal formats.

An `INVALID_OPERATION` error is generated if *format* does not match the internal format of the texture image being modified, since these commands do not provide for image format conversion.

An `INVALID_VALUE` error is generated if *width*, *height*, *depth*, or *imageSize* is negative.

An `INVALID_VALUE` error is generated if *imageSize* is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data).

An `INVALID_OPERATION` error is generated if any format-specific restrictions are violated, as with **CompressedTex*Image** commands. Any such restrictions will be documented in the specification defining the compressed internal format.

An `INVALID_OPERATION` error is generated if *xoffset*, *yoffset*, or *zoffset* are not equal to $-b_s$ (border width), or if *width*, *height*, and *depth* do not match the corresponding dimensions of the texture level. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

An `INVALID_ENUM` error is generated by **CompressedTex*SubImage1D**

if the internal format of the texture is one of the specific compressed formats.

An INVALID_OPERATION error is generated by **CompressedTex*SubImage2D** if the internal format of the texture is one of the EAC, ETC2, or RGTC formats and *border* is non-zero.

An INVALID_OPERATION error is generated by **CompressedTex*SubImage3D** if the internal format of the texture is one of the EAC, ETC2, or RGTC formats and either *border* is non-zero, or the effective target for the texture is not TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY.

An INVALID_OPERATION error is generated if the internal format of the texture is one of the BPTC formats and *border* is non-zero.

An INVALID_OPERATION error is generated if any of the following conditions occurs:

- *width* is not a multiple of four, and $width + xoffset$ is not equal to the value of TEXTURE_WIDTH.
- *height* is not a multiple of four, and $height + yoffset$ is not equal to the value of TEXTURE_HEIGHT.
- *xoffset* or *yoffset* is not a multiple of four.

8.8 Multisample Textures

In addition to the texture types described in previous sections, two additional types of textures are supported. A multisample texture is similar to a two-dimensional or two-dimensional array texture, except it contains multiple samples per texel. Multisample textures do not have multiple image levels.

The commands

```
void TexImage2DMultisample( enum target, sizei samples,
    enum internalformat, sizei width, sizei height,
    boolean fixedsamplelocations );
void TexImage3DMultisample( enum target, sizei samples,
    enum internalformat, sizei width, sizei height,
    sizei depth, boolean fixedsamplelocations );
```

establish the data storage, format, dimensions, and number of samples of a multisample texture's image. For **TexImage2DMultisample**, *target* must be TEXTURE_2D_MULTISAMPLE or PROXY_TEXTURE_2D_MULTISAMPLE and for **TexImage3DMultisample** *target* must be TEXTURE_2D_MULTISAMPLE_ARRAY

or PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY. *width* and *height* are the dimensions in texels of the texture.

samples represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled textures, the actual number of samples allocated for the texture image is implementation-dependent. However, the resulting value for TEXTURE_SAMPLES is guaranteed to be greater than or equal to *samples* and no more than the next larger sample count supported by the implementation.

If *fixedsamplelocations* is TRUE, the image will use identical sample locations and the same number of samples for all texels in the image, and the sample locations will not depend on the internal format or size of the image.

Upon success, **TexImage*Multisample** deletes any existing image for *target* and the contents of texels are undefined. TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_SAMPLES, TEXTURE_INTERNAL_FORMAT and TEXTURE_FIXED_SAMPLE_LOCATIONS are set to *width*, *height*, the actual number of samples allocated, *internalformat*, and *fixedsamplelocations* respectively.

When a multisample texture is accessed in a shader, the access takes one vector of integers describing which texel to fetch and an integer corresponding to the sample numbers described in section 14.3.1 determining which sample within the texel to fetch. No standard sampling instructions are allowed on the multisample texture targets, and no filtering is performed by the fetch. Fetching a sample number less than zero, or greater than or equal to the number of samples in the texture, produces undefined results.

Errors

An INVALID_ENUM error is generated if *target* is not an accepted multisample target as described above.

An INVALID_VALUE error is generated if *width*, *height*, or *depth* is negative.

An INVALID_VALUE error is generated if *samples* is zero.

An INVALID_VALUE error is generated if *width* or *height* is greater than the value of MAX_TEXTURE_SIZE.

An INVALID_VALUE error is generated by **TexImage3DMultisample** if *depth* is greater than the value of MAX_ARRAY_TEXTURE_LAYERS.

An INVALID_ENUM error is generated if *internalformat* is not color-renderable, depth-renderable, or stencil-renderable (as defined in section 9.4).

An INVALID_OPERATION error is generated if *samples* is greater than the maximum number of samples supported for this *target* and *internalformat*.

The maximum number of samples supported can be determined by calling **GetInternalformativ** with a *pname* of SAMPLES (see section 22.3).

An INVALID_OPERATION error is generated if the value of TEXTURE_IMMUTABLE_FORMAT for the texture currently bound to *target* on the active texture unit is TRUE.

8.9 Buffer Textures

In addition to one-, two-, and three-dimensional, one- and two-dimensional array, and cube map textures described in previous sections, one additional type of texture is supported. A buffer texture is similar to a one-dimensional texture. However, unlike other texture types, the texture image is not stored as part of the texture. Instead, a buffer object is attached to a buffer texture and the texture image is taken from that buffer object's data store. When the contents of a buffer object's data store are modified, those changes are reflected in the contents of any buffer texture to which the buffer object is attached. Buffer textures do not have multiple image levels; only a single data store is available.

The commands

```
void TexBufferRange( enum target, enum internalformat,
    uint buffer, intptr offset, sizeiptr size );
void TextureBufferRange( uint texture, enum internalformat,
    uint buffer, intptr offset, sizeiptr size );
```

attach the range of the storage for the buffer object named *buffer* for *size* basic machine units, starting at *offset* (also in basic machine units) to a buffer texture.

For **TexBufferRange**, the buffer texture is that currently bound to *target*. For **TextureBufferRange**, *texture* is the name of the buffer texture. *target* or the effective target of *texture* must be TEXTURE_BUFFER.

If *buffer* is zero, then any buffer object attached to the buffer texture is detached, the values *offset* and *size* are ignored and the state for *offset* and *size* for the buffer texture are reset to zero. *internalformat* specifies the storage format for the texture image found in the range of the attached buffer object, and must be one of the sized internal formats found in table 8.24.

Errors

An INVALID_OPERATION error is generated by **TextureBufferRange** if *texture* is not the name of an existing texture object.

An INVALID_ENUM error is generated by **TexBufferRange** if *target* is not

TEXTURE_BUFFER.

An INVALID_OPERATION error is generated by **TextureBufferRange** if the effective *target* is not TEXTURE_BUFFER.

An INVALID_ENUM error is generated if *internalformat* is not one of the sized internal formats in table 8.24.

An INVALID_OPERATION error is generated if *buffer* is non-zero and is not the name of a buffer object.

An INVALID_VALUE error is generated if *offset* is negative, if *size* is less than or equal to zero, or if $offset + size$ is greater than the value of BUFFER_SIZE for the buffer bound to *target*.

An INVALID_VALUE error is generated if *offset* is not an integer multiple of the value of TEXTURE_BUFFER_OFFSET_ALIGNMENT.

The commands

```
void TexBuffer( enum target, enum internalformat,
    uint buffer );
void TextureBuffer( uint texture, enum internalformat,
    uint buffer );
```

are respectively equivalent to

TexBufferRange(*target*, *internalformat*, *buffer*, 0, size);

and

TextureBufferRange(*texture*, *internalformat*, *buffer*, 0, size);

where size is the value of BUFFER_SIZE for *buffer*.

When a range of the storage of a buffer object is attached to a buffer texture, the range of the buffer's data store is taken as the texture's texture image. The number of texels in the buffer texture's texture image is given by

$$\left\lfloor \frac{size}{components \times sizeof(base_type)} \right\rfloor .$$

where $components$ and $base_type$ are the element count and base type for elements, as specified in table 8.24.

The number of texels in the texture image is then clamped to an implementation-dependent limit, the value of MAX_TEXTURE_BUFFER_SIZE. When a buffer texture is accessed in a shader, the results of a texel fetch are undefined if the specified texel coordinate is negative, or greater than or equal to the clamped number of texels in the texture image.

When a buffer texture is accessed in a shader, an integer is provided to indicate the texel coordinate being accessed. If no buffer object is bound to the buffer texture, the results of the texel access are undefined. Otherwise, the attached buffer object's data store is interpreted as an array of elements of the GL data type corresponding to *internalformat*. Each texel consists of one to four elements that are mapped to texture components (R, G, B, and A). Element m of the texel numbered n is taken from element $n \times components + m$ of the attached buffer object's data store. Elements and texels are both numbered starting with zero. For texture formats with signed or unsigned normalized fixed-point components, the extracted values are converted to floating-point using equations 2.2 or 2.1, respectively. The components of the texture are then converted to a (R, G, B, A) vector according to table 8.24, and returned to the shader as a four-component result vector with components of the appropriate data type for the texture's internal format. The base data type, component count, normalized component information, and mapping of data store elements to texture components is specified in table 8.24.

Sized Internal Format	Base Type	Components	Norm	Component			
				0	1	2	3
R8	ubyte	1	Yes	R	0	0	1
R16	ushort	1	Yes	R	0	0	1
R16F	half	1	No	R	0	0	1
R32F	float	1	No	R	0	0	1
R8I	byte	1	No	R	0	0	1
R16I	short	1	No	R	0	0	1
R32I	int	1	No	R	0	0	1
R8UI	ubyte	1	No	R	0	0	1
R16UI	ushort	1	No	R	0	0	1
R32UI	uint	1	No	R	0	0	1
RG8	ubyte	2	Yes	R	G	0	1
RG16	ushort	2	Yes	R	G	0	1
RG16F	half	2	No	R	G	0	1
RG32F	float	2	No	R	G	0	1
RG8I	byte	2	No	R	G	0	1
RG16I	short	2	No	R	G	0	1
RG32I	int	2	No	R	G	0	1
RG8UI	ubyte	2	No	R	G	0	1
RG16UI	ushort	2	No	R	G	0	1

(Continued on next page)

Internal formats for buffer textures (continued)							
Sized Internal Format	Base Type	Components	Norm	Component			
				0	1	2	3
RG32UI	uint	2	No	R	G	0	1
RGB32F	float	3	No	R	G	B	1
RGB32I	int	3	No	R	G	B	1
RGB32UI	uint	3	No	R	G	B	1
RGBA8	ubyte	4	Yes	R	G	B	A
RGBA16	ushort	4	Yes	R	G	B	A
RGBA16F	half	4	No	R	G	B	A
RGBA32F	float	4	No	R	G	B	A
RGBA8I	byte	4	No	R	G	B	A
RGBA16I	short	4	No	R	G	B	A
RGBA32I	int	4	No	R	G	B	A
RGBA8UI	ubyte	4	No	R	G	B	A
RGBA16UI	ushort	4	No	R	G	B	A
RGBA32UI	uint	4	No	R	G	B	A

Table 8.24: Internal formats for buffer textures. For each format, the data type of each element is indicated in the "Base Type" column and the element count is in the "Components" column. The "Norm" column indicates whether components should be treated as normalized floating-point values. The "Component 0, 1, 2, and 3" columns indicate the mapping of each element of a texel to texture components.

In addition to attaching buffer objects to textures, buffer objects can be bound to the buffer object target named TEXTURE_BUFFER, in order to specify, modify, or read the buffer object's data store. The buffer object bound to TEXTURE_BUFFER has no effect on rendering. A buffer object is bound to TEXTURE_BUFFER by calling **BindBuffer** with *target* set to TEXTURE_BUFFER, as described in section 6.

8.10 Texture Parameters

Texture parameters control how the texture image of a texture object is treated when specified or changed, and when applied to a fragment. Each parameter is set with the commands

```
void TexParameter{if}( enum target, enum pname, T param );
void TexParameter{if}v( enum target, enum pname, const
    T *params );
void TexParameterI{i ui}v( enum target, enum pname, const
    T *params );
void TextureParameter{if}( uint texture, enum pname,
    T param );
void TextureParameter{if}v( uint texture, enum pname,
    const T *params );
void TextureParameterI{i ui}v( uint texture, enum pname,
    const T *params );
```

For **TexParameter***, the texture object is that bound to *target*. For **TextureParameter***, *texture* is the name of the texture object. *target* or the effective target of *texture* must be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY. TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_2D_MULTISAMPLE, or TEXTURE_2D_MULTISAMPLE_ARRAY.

pname is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 8.25. In the scalar forms of the command, *param* is a value to which to set a single-valued parameter; in the vector forms, *params* is an array of parameters whose type depends on the parameter being set.

Data conversions are performed as specified in section 2.2.1, with these exceptions:

- If the value for TEXTURE_PRIORITY is specified with **TexParameteri** or **TexParameteriv**, it is treated as a signed normalized fixed-point value and converted to floating-point using equation 2.2, followed by clamping the value to lie in $[0, 1]$.
- If the values for TEXTURE_BORDER_COLOR are specified with **TexParameterIiv** or **TexParameterIuiv**, they are unmodified and stored with an internal data type of integer. If specified with **TexParameteriv**, they are converted to floating-point using equation 2.2. Otherwise, the values are unmodified and stored as floating-point.

If *pname* is TEXTURE_SWIZZLE_RGBA, *params* is an array of four enums which respectively set the TEXTURE_SWIZZLE_R, TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A parameters simultaneously.

Name	Type	Legal Values
DEPTH_TEXTURE_MODE	enum	RED, LUMINANCE, INTENSITY, ALPHA
DEPTH_STENCIL_TEXTURE_MODE	enum	DEPTH_COMPONENT, STENCIL_INDEX
GENERATE_MIPMAP	boolean	TRUE or FALSE
TEXTURE_BASE_LEVEL	int	any non-negative integer
TEXTURE_BORDER_COLOR	4 floats, ints, or uints	any 4 values
TEXTURE_COMPARE_MODE	enum	NONE, COMPARE_REF_TO_TEXTURE
TEXTURE_COMPARE_FUNC	enum	LEQUAL, GEQUAL, LESS, GREATER, EQUAL, NOTEQUAL, ALWAYS, NEVER
TEXTURE_LOD_BIAS	float	any value
TEXTURE_MAG_FILTER	enum	NEAREST, LINEAR
TEXTURE_MAX_ANISOTROPY	float	greater or equal to 1.0
TEXTURE_MAX_LEVEL	int	any non-negative integer
TEXTURE_MAX_LOD	float	any value
TEXTURE_MIN_FILTER	enum	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR,
TEXTURE_MIN_LOD	float	any value
TEXTURE_PRIORITY	float	any value in $[0, 1]$
TEXTURE_SWIZZLE_R	enum	RED, GREEN, BLUE, ALPHA, ZERO, ONE
TEXTURE_SWIZZLE_G	enum	RED, GREEN, BLUE, ALPHA, ZERO, ONE
TEXTURE_SWIZZLE_B	enum	RED, GREEN, BLUE, ALPHA, ZERO, ONE
TEXTURE_SWIZZLE_A	enum	RED, GREEN, BLUE, ALPHA, ZERO, ONE
TEXTURE_SWIZZLE_RGBA	4 enums	RED, GREEN, BLUE, ALPHA, ZERO, ONE
Texture parameters continued on next page		

Texture parameters continued from previous page		
Name	Type	Legal Values
TEXTURE_WRAP_S	enum	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT, MIRROR_CLAMP_TO_EDGE
TEXTURE_WRAP_T	enum	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT, MIRROR_CLAMP_TO_EDGE
TEXTURE_WRAP_R	enum	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT, MIRROR_CLAMP_TO_EDGE

Table 8.25: Texture parameters and their values.

In the remainder of chapter 8, denote by lod_{min}, lod_{max}, $level_{base}$, and $level_{max}$ the values of the texture parameters TEXTURE_MIN_LOD, TEXTURE_MAX_LOD, TEXTURE_BASE_LEVEL, and TEXTURE_MAX_LEVEL respectively. If the texture was created with **TextureView**, then the TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL parameters are interpreted relative to the view and not relative to the original data store.

Texture parameters for a cube map texture apply to the cube map as a whole; the six distinct two-dimensional texture images use the texture parameters of the cube map itself.

If the value of texture parameter GENERATE_MIPMAP is TRUE, specifying or changing texture images may have side effects, which are discussed in the **Automatic Mipmap Generation** discussion of section 8.14.

Errors

An INVALID_ENUM error is generated by **TexParameter*** if *target* is not one of the valid targets listed above.

An INVALID_OPERATION error is generated by **TextureParameter*** if the effective *target* is not one of the valid targets listed above.

An INVALID_ENUM error is generated if *pname* is not one of the parameter names in table 8.25.

An INVALID_OPERATION error is generated by **TextureParameter*** if *texture* is not the name of an existing texture object.

An INVALID_ENUM error is generated if the type of the parameter specified by *pname* is enum, and any value specified by *param* or *params* is not among the legal values shown in table 8.25.

An INVALID_VALUE error is generated if *pname* is TEXTURE_BASE_LEVEL or TEXTURE_MAX_LEVEL, and any value specified by *param* or *params* is negative.

An INVALID_VALUE error is generated if *pname* is TEXTURE_MAX_ANISOTROPY, and any value specified by *param* or *params* is less than 1.0.

An INVALID_ENUM error is generated if **Tex*Parameter{if}** is called for a non-scalar parameter (*pname* TEXTURE_BORDER_COLOR or TEXTURE_SWIZZLE_RGBA).

An INVALID_ENUM error is generated by **TexParameter*** if *target* is either TEXTURE_2D_MULTISAMPLE or TEXTURE_2D_MULTISAMPLE_ARRAY, and *pname* is any sampler state from table 23.23.

An INVALID_OPERATION error is generated by **TextureParameter*** if the effective *target* is either TEXTURE_2D_MULTISAMPLE or TEXTURE_2D_MULTISAMPLE_ARRAY, and *pname* is any sampler state from table 23.23.

An INVALID_OPERATION error is generated if the effective *target* is TEXTURE_2D_MULTISAMPLE, TEXTURE_2D_MULTISAMPLE_ARRAY or TEXTURE_RECTANGLE, and *pname* TEXTURE_BASE_LEVEL is set to a value other than zero.

An INVALID_ENUM error is generated if the effective *target* is TEXTURE_RECTANGLE and either of *pname*s TEXTURE_WRAP_S or TEXTURE_WRAP_T is set to either MIRROR_CLAMP_TO_EDGE, MIRRORED_REPEAT or REPEAT.

An INVALID_ENUM error is generated if the effective *target* is TEXTURE_RECTANGLE and *pname* TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR (no mipmap filtering is permitted).

8.11 Texture Queries

8.11.1 Active Texture

Queries of most texture state variables are qualified by the value of ACTIVE_TEXTURE to determine which server texture state vector is queried.

Client texture state variables such as texture coordinate array pointers are qualified by the value of CLIENT_ACTIVE_TEXTURE. Tables 23.3, 23.4, 23.10, 23.17, 23.24, and 23.66 indicate those state variables which are qualified by ACTIVE_TEXTURE or CLIENT_ACTIVE_TEXTURE during state queries.

Errors

An INVALID_OPERATION error is generated by queries of texture state variables corresponding to texture coordinate processing units (namely, **TexGen** state and enables, and matrices) if the value of ACTIVE_TEXTURE is greater than or equal to the value of MAX_TEXTURE_COORDS.

An INVALID_OPERATION error is generated by all other texture state queries if the value of ACTIVE_TEXTURE is greater than or equal to the value of MAX_COMBINED_TEXTURE_IMAGE_UNITS.

8.11.2 Texture Parameter Queries

Parameters of a texture object may be queried with the commands

```
void GetTexParameter{if}v( enum target, enum pname,
    T *params );
void GetTexParameterI{i ui}v( enum target, enum pname,
    T *params );
void GetTextureParameter{if}v( uint texture, enum pname,
    T *data );
void GetTextureParameterI{i ui}v( uint texture,
    enum pname, T *data );
```

For **GetTexParameter***, the texture object is that bound to *target*. For **GetTextureParameter***, *texture* is the name of the texture object.

The value of texture parameter *pname* for the texture is returned in *params*.

target or the effective target of *texture* must be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP, TEXTURE_CUBE_MAP_ARRAY,

TEXTURE_2D_MULTISAMPLE, or TEXTURE_2D_MULTISAMPLE_ARRAY, indicating the currently bound one-, two-, or three-dimensional, one- or two-dimensional array, rectangle, cube map, cube map array, two-dimensional multisample, or two-dimensional multisample array texture object.

pname must be one of IMAGE_FORMAT_COMPATIBILITY_TYPE, TEXTURE_IMMUTABLE_FORMAT, TEXTURE_IMMUTABLE_LEVELS, TEXTURE_TARGET, TEXTURE_VIEW_MIN_LEVEL, TEXTURE_VIEW_NUM_LEVELS, TEXTURE_VIEW_MIN_LAYER, TEXTURE_VIEW_NUM_LAYERS, TEXTURE_RESIDENT, or one of the symbolic values in table 8.25.

Querying *pname* TEXTURE_BORDER_COLOR with **GetTex*ParameterIiv** or **GetTex*ParameterIuiv** returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 2.2.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.

Querying *pname* TEXTURE_TARGET returns the *effective target* of the texture object. For **GetTexParameter***, this is the *target* parameter. For **GetTextureParameter***, it is the target to which the texture was initially bound when it was created, or the value of the *target* parameter to the call to **CreateTextures** which created the texture.

Errors

An INVALID_OPERATION error is generated by **GetTextureParameter*** if *texture* is not the name of an existing texture object.

An INVALID_ENUM error is generated by **GetTexParameter*** if *target* is not one of the texture targets described above.

An INVALID_OPERATION error is generated by **GetTextureParameter*** if the effective *target* is not one of the texture targets described above.

An INVALID_ENUM error is generated if *pname* is not one of the texture parameters described above.

8.11.3 Texture Level Parameter Queries

Parameters of a specified level-of-detail of a texture object may be queried with the commands

```
void GetTexLevelParameter{if}v( enum target, int level,
    enum pname, T *params );
void GetTextureLevelParameter{if}v( uint texture,
    int level, enum pname, T *params );
```

For **GetTexLevelParameter***, the texture object is that bound to *target*. For **GetTextureLevelParameter***, *texture* is the name of the texture object.

The value of texture parameter *pname* for level-of-detail *level* of the texture is returned in *params*. *pname* must be one of the symbolic values in tables 23.21-23.22.

The effective target of the texture object must be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_RECTANGLE, TEXTURE_BUFFER, TEXTURE_2D_MULTISAMPLE, TEXTURE_2D_MULTISAMPLE_ARRAY, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, PROXY_TEXTURE_1D_ARRAY, PROXY_TEXTURE_2D_ARRAY, PROXY_TEXTURE_CUBE_MAP_ARRAY, PROXY_TEXTURE_RECTANGLE, PROXY_TEXTURE_CUBE_MAP, PROXY_TEXTURE_2D_MULTISAMPLE, or PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY, indicating the one-, two-, or three-dimensional texture, one- or two-dimensional array texture, cube map array texture, rectangle texture, buffer texture, two-dimensional multisample texture, two-dimensional multisample array texture; or the one-, two-, three-dimensional, one- or two-dimensional array, cube map array, rectangle, cube map, two-dimensional multisample, or two-dimensional multisample array proxy state vector.

For **GetTexLevelParameter*** only, *target* may also be one of the cube map face targets from table 8.27, indicating one of the six distinct two-dimensional images making up the cube map texture object. Note that TEXTURE_CUBE_MAP is not a valid *target* parameter for **GetTexLevelParameter***.

For **GetTextureLevelParameter*** only, *texture* may also be a cube map texture object. In this case the query is always performed for face zero (the TEXTURE_CUBE_MAP_POSITIVE_X face), since there is no way to specify another face.

level determines which level-of-detail's state is returned. The maximum value of *level* depends on the texture *target*:

- For *target*s TEXTURE_CUBE_MAP and TEXTURE_CUBE_MAP_ARRAY, the maximum value is $\log_2$ of the value of MAX_CUBE_MAP_TEXTURE_SIZE.
- For *target* TEXTURE_3D, the maximum value is $\log_2$ of the value of MAX_3D_TEXTURE_SIZE.
- For *target*s TEXTURE_BUFFER, TEXTURE_RECTANGLE, TEXTURE_2D_MULTISAMPLE, and TEXTURE_2D_MULTISAMPLE_ARRAY, which do not support mipmaps, the maximum value is zero.
- For all other texture *target*s supported by **GetTex*LevelParameter***, the maximum value is $\log_2$ of the value of MAX_TEXTURE_SIZE.

For texture images with uncompressed internal formats, queries of *pname* TEXTURE_RED_TYPE, TEXTURE_GREEN_TYPE, TEXTURE_BLUE_TYPE, TEXTURE_ALPHA_TYPE, TEXTURE_LUMINANCE_TYPE, TEXTURE_INTENSITY_TYPE, and TEXTURE_DEPTH_TYPE return the data type used to store the component. Types NONE, SIGNED_NORMALIZED, UNSIGNED_NORMALIZED, FLOAT, INT, and UNSIGNED_INT respectively indicate missing, signed normalized fixed-point, unsigned normalized fixed-point, floating-point, signed unnormalized integer, and unsigned unnormalized integer components. Queries of *pname* TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, TEXTURE_INTENSITY_SIZE, TEXTURE_DEPTH_SIZE, TEXTURE_STENCIL_SIZE, and TEXTURE_SHARED_SIZE return the actual resolutions of the stored texture image components, not the resolutions specified when the image was defined.

For texture images with compressed internal formats, the types returned specify how components are interpreted after decompression, while the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. Since the quality of the implementation's compression algorithm is likely data-dependent, the returned component sizes should be treated only as rough approximations.

Querying *pname* TEXTURE_COMPRESSED_IMAGE_SIZE returns the size (in ubytes) of the compressed texture image that would be returned by **GetCompressedTexImage** (section 8.11). *target* must be a compressed texture target.

Queries of *pname* TEXTURE_SAMPLES and TEXTURE_FIXED_SAMPLE_LOCATIONS on multisample textures return the number of samples and whether texture sample fixed locations are enabled respectively. For non-multisample textures, the default values in tables 23.21- 23.22 are returned.

Queries of *pname* TEXTURE_BORDER, TEXTURE_INTERNAL_FORMAT, TEXTURE_WIDTH, TEXTURE_HEIGHT, and TEXTURE_DEPTH return the border width, internal format, width, height, and depth, respectively, as specified when the texture image was created. The internal format of the texture image may also be queried as TEXTURE_COMPONENTS, for compatibility with GL version 1.0.

Errors

An INVALID_OPERATION error is generated by **GetTextureLevelParameter*** if *texture* is not the name of an existing texture object.

An INVALID_ENUM error is generated by **GetTexLevelParameter*** if *tar-*

get is not one of the targets described above as valid for the corresponding command.

An INVALID_OPERATION error is generated by **GetTextureLevelParameter*** if the effective *target* is not one of the targets described above as valid for the corresponding command.

An INVALID_ENUM error is generated if *pname* is not one of the symbolic values in tables 23.21- 23.22.

An INVALID_VALUE error is generated if *level* is negative or larger than the maximum allowable level-of-detail for the effective texture target as described above.

An INVALID_OPERATION error is generated if *pname* is TEXTURE_COMPRESSED_IMAGE_SIZE and the effective texture target is a proxy target, or has an uncompressed internal format.

8.11.4 Texture Image Queries

Texture images may be obtained from a texture object with the commands

```
void GetTexImage( enum target, int level, enum format,
    enum type, void *pixels );
void GetTextureImage( uint texture, int level, enum format,
    enum type, sizei bufSize, void *pixels );
void GetnTexImage( enum target, int level, enum format,
    enum type, sizei bufSize, void *pixels );
```

For **Get*TexImage**, *target* specifies the target to which the texture object is bound. *target* must be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY or TEXTURE_RECTANGLE, indicating a one-, two- or three-dimensional, one- or two-dimensional array, cube map array or rectangle texture, respectively, or one of the targets from table 8.27, indicates the corresponding face of a cube map texture.

For **GetTextureImage**, *texture* is the name of the texture object. In addition to the types of textures accepted by the **Get*TexImage** commands, **GetTextureImage** also accepts cube map texture objects (with effective *target* TEXTURE_CUBE_MAP).

level is a level-of-detail number, *format* is a pixel format from table 8.8, and *type* is a pixel type from table 8.7.

If present, *bufSize* is the size of the buffer to receive the retrieved pixel data.

GetnTexImage and **GetTextureImage** do not write more than *bufSize* bytes into *pixels*.

These commands obtain component groups from a texture image with the indicated level-of-detail. If *format* is a color format then the components are assigned among R, G, B, and A according to table 8.26, starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for three-dimensional textures. One- and two-dimensional array and cube map array textures are treated as two-, three-, and three-dimensional images, respectively, where the layers are treated as rows or images. Cube map textures are treated as three-dimensional images with a depth of 6, where the cube map faces are ordered as image layers as shown in table 9.3.

If *format* is DEPTH_COMPONENT, DEPTH_STENCIL, or STENCIL_INDEX, then each depth component and/or stencil index is assigned with the same ordering of rows and images.

These groups are then packed and placed in client or pixel buffer object memory. If a pixel pack buffer is bound (as indicated by a non-zero value of PIXEL_PACK_BUFFER_BINDING), *pixels* is an offset into the pixel pack buffer; otherwise, *pixels* is a pointer to client memory. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **ReadPixels** are applied, as described in table 18.1 and section 18.2.9.

For three-dimensional, two-dimensional array, cube map array, and cube map textures pixel storage operations are applied as if the image were two-dimensional, except that the additional pixel storage state values PACK_IMAGE_HEIGHT and PACK_SKIP_IMAGES are applied. The correspondence of texels to memory locations is as defined for **TexImage3D** in section 8.5.

The row length, number of rows, image depth, and number of images are determined by the size of the texture image (including any borders).

Errors

An INVALID_OPERATION error is generated by **GetTextureImage** if *texture* is not the name of an existing texture object.

An INVALID_ENUM error is generated by **GetTexImage** and **GetnTexImage** if *target* is not one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_RECTANGLE, or one of the targets from table 8.27.

An INVALID_OPERATION error is generated by **GetTextureImage** if the effective *target* is not one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_RECTANGLE, or TEXTURE_CUBE_MAP (for **GetTextureIm-**

age only).

An INVALID_OPERATION error is generated by **GetTextureImage** if the effective *target* is TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY, and the texture object is not cube complete or cube array complete, respectively.

An INVALID_VALUE error is generated if *level* is negative or larger than the maximum allowable level.

An INVALID_ENUM error is generated if *format* is COLOR_INDEX.

An INVALID_VALUE error is generated if *level* is non-zero and the effective *target* is TEXTURE_RECTANGLE.

An INVALID_OPERATION error is generated if any of the following mismatches between *format* and the internal format of the texture image exist:

- *format* is a color format (one of the formats in table 8.8 whose target is the color buffer) and the base internal format of the texture image is not a color format.
- *format* is DEPTH_COMPONENT and the base internal format is not DEPTH_COMPONENT or DEPTH_STENCIL.
- *format* is DEPTH_STENCIL and the base internal format is not DEPTH_STENCIL.
- *format* is STENCIL_INDEX and the base internal format is not STENCIL_INDEX or DEPTH_STENCIL.
- *format* is one of the integer formats in table 8.8 and the internal format of the texture image is not integer, or *format* is not one of the integer formats in table 8.8 and the internal format is integer.

An INVALID_OPERATION error is generated if a pixel pack buffer object is bound and packing the texture image into the buffer's memory would exceed the size of the buffer.

An INVALID_OPERATION error is generated if a pixel pack buffer object is bound and *pixels* is not evenly divisible by the number of basic machine units needed to store in memory the GL data type corresponding to *type* (see table 8.7).

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetTextureImage** and **GetnTexImage** if the buffer size required to store the requested data is greater than *bufSize*.

Sub-regions of a texture image may be obtained from a texture object with the

Base Internal Format	R	G	B	A
ALPHA	0	0	0	A_i
LUMINANCE (or 1)	L_i	0	0	1
LUMINANCE_ALPHA (or 2)	L_i	0	0	A_i
INTENSITY	I_i	0	0	1
RED	R_i	0	0	1
RG	R_i	G_i	0	1
RGB (or 3)	R_i	G_i	B_i	1
RGBA (or 4)	R_i	G_i	B_i	A_i

Table 8.26: Texture, table and filter return values. R_i, G_i, B_i, A_i, L_i, and I_i are components of the internal format that are assigned to pixel values R, G, B, and A. If a requested pixel value is not present in the internal format, the specified constant value is used.

command

```
void GetTextureSubImage( uint texture, int level,
    int xoffset, int yoffset, int zoffset, sizei width,
    sizei height, sizei depth, enum format, enum type,
    sizei bufSize, void *pixels );
```

texture is the name of the texture object, and must not be a buffer or multisample texture. The effective *target* is the value of TEXTURE_TARGET for *texture*. *level*, *format*, *type* and *pixels* have the same meaning as for **GetTexImage**. *bufSize* is the size of the buffer to receive the retrieved pixel data.

For cube map textures, the behavior is as though **GetTextureImage** were called, but only texels from the requested cube map faces (selected by *zoffset* and *depth*, as described below) were returned.

xoffset, *yoffset* and *zoffset* indicate the position of the subregion to return. *width*, *height* and *depth* indicate the size of the region to return. These parameters have the same meaning as for **TexSubImage3D**, though for one- and two-dimensional textures there are extra restrictions, described in the errors section below.

For one-dimensional array textures, *yoffset* is interpreted as the first layer to access and *height* is the number of layers to access. For two-dimensional array textures, *zoffset* is interpreted as the first layer to access and *depth* is the number of layers to access. Cube map textures are treated as an array of six slices in the z-dimension, where the value of *zoffset* is interpreted as specifying the cube map face for the corresponding layer in table 9.3 and *depth* is the number of faces to access.

For cube map array textures, *zoffset* is the first layer-face to access, and *depth* is the number of layer-faces to access. Each layer-face is translated into an array layer and a cube map face as described for layer-face numbers in section 8.5.3.

Component groups from the specified sub-region are packed and placed into memory as described for **GetTextureImage**, starting with the texel at (xoffset, yoffset, zoffset).

Errors

An `INVALID_OPERATION` error is generated if *texture* is not the name of an existing texture object.

An `INVALID_OPERATION` error is generated if *texture* is the name of a buffer or multisample texture.

An `INVALID_OPERATION` error is generated if the effective *target* is `TEXTURE_CUBE_MAP` or `TEXTURE_CUBE_MAP_ARRAY`, and the texture object is not cube complete or cube array complete, respectively.

An `INVALID_VALUE` error is generated if *xoffset*, *yoffset* or *zoffset* are negative.

An `INVALID_VALUE` error is generated if *width*, *height*, or *depth* is negative.

An `INVALID_VALUE` error is generated if *xoffset* + *width* is greater than the texture's width, *yoffset* + *height* is greater than the texture's height, or *zoffset* + *depth* is greater than the texture's depth.

An `INVALID_VALUE` error is generated if the effective *target* is `TEXTURE_1D` and either *yoffset* is not zero, or *height* is not one.

An `INVALID_VALUE` error is generated if the effective *target* is `TEXTURE_1D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D` or `TEXTURE_RECTANGLE`, and either *zoffset* is not zero, or *depth* is not one.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

An `INVALID_OPERATION` error is generated if the buffer size required to store the requested data is greater than *bufSize*.

Texture images stored in compressed form may be obtained with the commands

`void` **GetCompressedTexImage**(`enum` *target*, `int` *level*, `void` **pixels*);
`void` **GetCompressedTextureImage**(`uint` *texture*, `int` *level*, `sizei` *bufSize*, `void` **pixels*);
`void` **GetnCompressedTexImage**(`enum` *target*, `int` *level*, `sizei` *bufSize*, `void` **pixels*);

For **Get*CompressedTexImage**, the texture object is that which is bound to *target*. For **GetCompressedTextureImage**, *texture* is the name of the texture object, and the effective *target* is the value of TEXTURE_TARGET for *texture*.

target, *level*, *bufSize*, and *pixels* are interpreted in the same manner as the corresponding parameters of **GetTexImage**, **GetTextureImage**, and **GetnTexImage**.

When called, **GetCompressedTexImage** writes n ubytes of compressed image data to the pixel pack buffer or client memory pointed to by *pixels*, while **GetCompressedTextureImage** and **GetnCompressedTexImage** write $\min(n, bufSize)$ ubytes. n is the value of TEXTURE_COMPRESSED_IMAGE_SIZE for the texture image The compressed image data is formatted according to the definition of the texture's internal format.

All pixel transfer modes are ignored when returning a compressed texture image.

By default the pixel storage modes PACK_ROW_LENGTH, PACK_SKIP_ROWS, PACK_SKIP_PIXELS, PACK_IMAGE_HEIGHT and PACK_SKIP_IMAGES are ignored for compressed images. To enable PACK_SKIP_PIXELS and PACK_ROW_LENGTH, the values of PACK_COMPRESSED_BLOCK_SIZE and PACK_COMPRESSED_BLOCK_WIDTH must both be non-zero. To also enable PACK_SKIP_ROWS and PACK_IMAGE_HEIGHT, the value of PACK_COMPRESSED_BLOCK_HEIGHT must be non-zero. And to also enable PACK_SKIP_IMAGES, the value of PACK_COMPRESSED_BLOCK_DEPTH must be non-zero. All parameters must be consistent with the compressed format to produce the desired results. When the pixel storage modes are active, the correspondence of texels to memory locations is as defined for **CompressedTexImage3D** in section 8.7.

Errors

An INVALID_OPERATION error is generated by **GetCompressedTextureImage** if *texture* is not the name of an existing texture object.

An INVALID_OPERATION error is generated by **GetCompressedTextureImage** if the effective *target* is TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY, and the texture object is not cube complete or cube array complete, respectively.

An INVALID_VALUE error is generated if *level* is negative, or greater than the maximum allowable level.

An INVALID_OPERATION error is generated if the texture image is stored with an uncompressed internal format.

An INVALID_OPERATION error is generated if a pixel pack buffer object is bound and packing the texture image into the buffer's memory would exceed

the size of the buffer.

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetCompressedTextureImage** and **GetnCompressedTexImage** if the buffer size required to store the requested data is greater than *bufSize*.

If the compressed data are arranged into fixed-size blocks of texels, the command

```
void GetCompressedTextureSubImage( uint texture,
    int level, int xoffset, int yoffset, int zoffset,
    sizei width, sizei height, sizei depth, sizei bufSize,
    void *pixels );
```

can be used to obtain a sub-region of a compressed texture image instead of the whole image. *texture* is the name of the texture object, and must not be a buffer or multisample texture. The effective *target* is the value of TEXTURE_TARGET for *texture*. *level* and *pixels* have the same meaning as the corresponding arguments of **CompressedTexSubImage3D**. *bufSize* indicates the size of the buffer to receive the retrieved pixel data.

For cube map textures, the behavior is as though **GetCompressedTexImage** were called once for each requested face (selected by *zoffset* and *depth*, as described below) with *target* corresponding to the requested texture cube map face as indicated by table 9.3. *pixels* is offset appropriately for each successive image.

xoffset, *yoffset* and *zoffset* indicate the position of the subregion to return. *width*, *height* and *depth* indicate the size of the region to return. These arguments have the same meaning as for **CompressedTexSubImage3D**, though there are extra restrictions, described in the errors section below.

The mapping between the *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* parameters and the faces, layers, and layer-faces for cube map, array, and cube map array textures is the same as for **GetTextureSubImage**.

The *xoffset*, *yoffset*, *zoffset* offsets and *width*, *height* and *depth* sizes must be multiples of the values of PACK_COMPRESSED_BLOCK_WIDTH, PACK_COMPRESSED_BLOCK_HEIGHT, and PACK_COMPRESSED_BLOCK_DEPTH respectively, unless an offset is zero and the corresponding size is the same as the texture size in that dimension.

Pixel storage modes are treated as for **GetCompressedTexSubImage**. The texel at (xoffset, yoffset, zoffset) will be stored at the location indicated by *pixels* and the current pixel packing parameters.

Errors

In addition to the same errors generated by **GetTextureSubImage** with corresponding parameters:

An INVALID_VALUE error is generated if *xoffset*, *yoffset* or *zoffset* is not a multiple of the compressed block width, height or depth respectively.

An INVALID_VALUE error is generated if *width*, *height* or *depth* is not a multiple of the compressed block width, height or depth respectively, unless the offset is zero and the size equals the texture image size.

An INVALID_OPERATION error is generated if the texture compression format is not based on fixed-size blocks.

8.12 Depth Component Textures

Depth textures and the depth components of depth/stencil textures can be treated as RED , LUMINANCE, INTENSITY or ALPHA textures during texture filtering and application (see section 8.23). The initial state for depth and depth/stencil textures treats them as LUMINANCE textures except in a forward-compatible context, where the initial state instead treats them as RED textures.

8.13 Cube Map Texture Selection

When a cube map texture is sampled, the $\begin{pmatrix} s & t & r \end{pmatrix}$ texture coordinates are treated as a direction vector $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$ emanating from the center of a cube. The q coordinate is ignored. At texture application time, the interpolated per-fragment direction vector selects one of the cube map face's two-dimensional images based on the largest magnitude coordinate direction (the major axis direction). If two or more coordinates have the identical magnitude, the implementation may define the rule to disambiguate this situation. The rule must be deterministic and depend only on $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$. The target column in table 8.27 explains how the major axis direction maps to the two-dimensional image of a particular cube map target.

Using the s_c, t_c, and m_a determined by the major axis direction as specified in table 8.27, an updated $\begin{pmatrix} s & t \end{pmatrix}$ is calculated as follows:

$$s = \frac{1}{2}\left(\frac{s_c}{|m_a|} + 1\right)$$

$$t = \frac{1}{2}\left(\frac{t_c}{|m_a|} + 1\right)$$

Major Axis Direction	Target	s_c	t_c	m_a
$+r_x$	TEXTURE_CUBE_MAP_POSITIVE_X	$-r_z$	$-r_y$	r_x
$-r_x$	TEXTURE_CUBE_MAP_NEGATIVE_X	r_z	$-r_y$	r_x
$+r_y$	TEXTURE_CUBE_MAP_POSITIVE_Y	r_x	r_z	r_y
$-r_y$	TEXTURE_CUBE_MAP_NEGATIVE_Y	r_x	$-r_z$	r_y
$+r_z$	TEXTURE_CUBE_MAP_POSITIVE_Z	r_x	$-r_y$	r_z
$-r_z$	TEXTURE_CUBE_MAP_NEGATIVE_Z	$-r_x$	$-r_y$	r_z

Table 8.27: Selection of cube map images based on major axis direction of texture coordinates.

8.13.1 Seamless Cube Map Filtering

Seamless cube map filtering is enabled or disabled by calling **Enable** or **Disable** with *target* TEXTURE_CUBE_MAP_SEAMLESS.

When seamless cube map filtering is disabled, the new $\begin{pmatrix} s & t \end{pmatrix}$ is used to find a texture value in the determined face's two-dimensional image using the rules given in sections 8.14 through 8.15.

When seamless cube map filtering is enabled, the rules for texel selection in sections 8.14 through 8.15 are modified so that texture wrap modes are ignored. Instead,

- If NEAREST filtering is done within a miplevel, always apply wrap mode CLAMP_TO_EDGE.
- If LINEAR filtering is done within a miplevel, always apply wrap mode CLAMP_TO_BORDER. Then,
  - If a texture sample location would lie in the texture border in either u or v, instead select the corresponding texel from the appropriate neighboring face.
  - If a texture sample location would lie in the texture border in *both* u and v (in one of the corners of the cube), there is no unique neighboring face from which to extract one texel. The recommended method to generate this texel is to average the values of the three available samples. However, implementations are free to construct this fourth texel in another way, so long as, when the three available samples have the same value, this texel also has that value.

The required state is one bit indicating whether seamless cube map filtering is enabled or disabled. Initially, it is disabled.

8.14 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to *magnify* or *minify* the texture image.

When the texture's value of `TEXTURE_MAX_ANISOTROPY` is equal to 1.0, the GL uses an isotropic texture filtering approach as described in this section and section 8.15. However, when the texture's value of `TEXTURE_MAX_ANISOTROPY` is greater than 1.0, the GL should use a texture filtering scheme that accounts for a degree of anisotropy up to the smaller of the texture's value of `TEXTURE_MAX_ANISOTROPY` or the implementation-defined value of `MAX_TEXTURE_MAX_ANISOTROPY`.

The particular scheme for anisotropic texture filtering is implementation dependent. Additionally, implementations are free to consider the current texture minification and magnification modes to control the specifics of the anisotropic filtering scheme used.

The anisotropic texture filtering scheme may only access mipmap levels if the minification filter is one that requires mipmaps. Additionally, when a minification filter is specified, the anisotropic texture filtering scheme may only access texture mipmap levels between the texture's values for `TEXTURE_BASE_LEVEL` and `TEXTURE_MAX_LEVEL`, inclusive. Implementations should also respect the values of `TEXTURE_MAX_LOD` and `TEXTURE_MIN_LOD` to the extent the anisotropic texture filtering scheme permits this.

The following describes one particular approach to implementing anisotropic texture filtering for the two-dimensional texturing case:

Anisotropic texture filtering substantially changes section 8.14. Previously a single scale factor P was determined based on the pixel's projection into texture space. Now two scale factors, ρ_x and ρ_y, are computed:

$$\rho_x = \sqrt{\frac{\partial u}{\partial x}^2 + \frac{\partial v}{\partial x}^2}$$
$$\rho_y = \sqrt{\frac{\partial u}{\partial y}^2 + \frac{\partial v}{\partial y}^2}$$
$$\rho_{max} = \max(\rho_x, \rho_y)$$
$$\rho_{min} = \min(\rho_x, \rho_y)$$
$$N = \min(\left\lceil \frac{\rho_{max}}{\rho_{min}} \right\rceil, maxAniso)$$
$$\lambda' = log_2(\frac{\rho_{max}}{N})$$

where $maxAniso$ is the smaller of the texture's value of `TEXTURE_MAX_ANISOTROPY` or the implementation-defined value of `MAX_TEXTURE_MAX_ANISOTROPY`.

It is acceptable for an implementation to round the sampling rate, N, up to the nearest supported sampling rate. For example, an implementation may only support power-of-two sampling rates.

It is also acceptable for an implementation to approximate the ideal functions ρ_x and ρ_y with functions f_x and f_y subject to the following constraints:

1. f_x is continuous and monotonically increasing in $\left|\frac{\partial u}{\partial x}\right|$ and $\left|\frac{\partial v}{\partial x}\right|$.
2. f_y is continuous and monotonically increasing in $\left|\frac{\partial u}{\partial y}\right|$ and $\left|\frac{\partial v}{\partial y}\right|$.
3. $\max(\left|\frac{\partial u}{\partial x}\right|, \left|\frac{\partial v}{\partial x}\right|) \leq f_x \leq \sqrt{2}\left(\left|\frac{\partial u}{\partial x}\right| + \left|\frac{\partial v}{\partial x}\right|\right)$.
4. $\max(\left|\frac{\partial u}{\partial y}\right|, \left|\frac{\partial v}{\partial y}\right|) \leq f_y \leq \sqrt{2}\left(\left|\frac{\partial u}{\partial y}\right| + \left|\frac{\partial v}{\partial y}\right|\right)$.

Instead of a single sample, τ, at (u, v, λ), N locations in the mipmap at level-of-detail λ are sampled within the texture footprint of the pixel.

This sum τ_{aniso} is defined using the single sample τ. When the texture's value of `TEXTURE_MAX_ANISOTROPHY` is greater than 1.0, use τ_{aniso} instead of τ to determine the fragment's texture value.

$$\tau_{aniso} = \begin{cases} \frac{1}{N}\sum_{i=1}^{N} \tau(u(x - \frac{1}{2} + \frac{i}{N+1}, y), v(x - \frac{1}{2} + \frac{i}{N+1}, y)), & P_x > P_y \\ \frac{1}{N}\sum_{i=1}^{N} \tau(u(x, y - \frac{1}{2} + \frac{i}{N+1}), v(x, y - \frac{1}{2} + \frac{i}{N+1})), & P_y \geq P_x \end{cases}$$

It is acceptable to approximate the u and v functions with equally spaced samples in texture space at level-of-details λ:

$$\tau_{aniso} = \begin{cases} \frac{1}{N}\sum_{i=1}^{N} \tau(u(x, y) + \frac{\partial u}{\partial x}(\frac{i}{N+1} - \frac{1}{2}), v(x, y) + \frac{\partial v}{\partial x}(\frac{i}{N+1} - \frac{1}{2})), & P_x > P_y \\ \frac{1}{N}\sum_{i=1}^{N} \tau(u(x, y) + \frac{\partial u}{\partial y}(\frac{i}{N+1} - \frac{1}{2}), v(x, y) + \frac{\partial v}{\partial y}(\frac{i}{N+1} - \frac{1}{2})), & P_y \geq P_x \end{cases}$$

8.14.1 Scale Factor and Level-of-Detail

The choice is governed by a scale factor $\rho(x, y)$ and the *level-of-detail* parameter $\lambda(x, y)$, defined as

$$\lambda_{base}(x, y) = \log_2[\rho(x, y)] \tag{8.8}$$

$$\lambda'(x, y) = \lambda_{base}(x, y) + clamp(bias_{texobj} + bias_{texunit} + bias_{shader}) \tag{8.9}$$

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & lod_{min} \leq \lambda' \leq lod_{max} \\ lod_{min}, & \lambda' < lod_{min} \\ undefined, & lod_{min} > lod_{max} \end{cases} \tag{8.10}$$

$bias_{texobj}$ is the value of TEXTURE_LOD_BIAS for the bound texture object (as described in section 8.10). $bias_{texunit}$ is the value of TEXTURE_LOD_BIAS for the current texture unit (as described in section 16.1). $bias_{shader}$ is the value of the optional bias parameter in the texture lookup functions available to fragment shaders. If the texture access is performed in a fragment shader without a provided bias, or outside a fragment shader, then $bias_{shader}$ is zero. The sum of these values is clamped to the range $[-bias_{max}, bias_{max}]$ where $bias_{max}$ is the value of the implementation defined constant MAX_TEXTURE_LOD_BIAS.

Different implementations have chosen to perform clamping on intermediate and final terms in computing λ' differently. Care should be taken that intermediate terms do not exceed the implementation-dependent range as different results may otherwise occur.

If $\lambda(x, y)$ is less than or equal to zero, the texture is said to be *magnified*; if it is greater, the texture is *minified*. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 8.15.

The initial values of lod_{min} and lod_{max} are chosen so as to never clamp the normal range of λ.

Let $s(x, y)$ be the function that associates an s texture coordinate with each set of window coordinates (x, y) that lie within a primitive; define $t(x, y)$ and $r(x, y)$

analogously. Let

$$
\begin{aligned}
u(x,y) &= \begin{cases} s(x,y) + \delta_u, & \text{rectangle texture} \\ w_t \times s(x,y) + \delta_u, & \text{otherwise} \end{cases} \\
v(x,y) &= \begin{cases} t(x,y) + \delta_v, & \text{rectangle texture} \\ h_t \times t(x,y) + \delta_v, & \text{otherwise} \end{cases} \\
w(x,y) &= d_t \times r(x,y) + \delta_w
\end{aligned}
\tag{8.11}
$$

where w_t, h_t, and d_t are as defined in section 8.5.3 for the texture image whose level is $level_{base}$. For a one-dimensional or one-dimensional array texture, define $v(x,y) = 0$ and $w(x,y) = 0$; for a two-dimensional, two-dimensional array, rectangle, cube map, or cube map array texture, define $w(x,y) = 0$.

$(\delta_u, \delta_v, \delta_w)$ are the texel offsets specified in the OpenGL Shading Language texture lookup functions that support offsets. If the texture function used does not support offsets, or for fixed-function texture accesses, all three shader offsets are taken to be zero.

If the value of any non-ignored component of the offset vector operand is outside implementation-dependent limits, the results of the texture lookup are undefined. For all instructions except `textureGather`, the limits are the values of `MIN_PROGRAM_TEXEL_OFFSET` and `MAX_PROGRAM_TEXEL_OFFSET`. For the `textureGather` instruction, the limits are the values of `MIN_PROGRAM_TEXTURE_GATHER_OFFSET` and `MAX_PROGRAM_TEXTURE_GATHER_OFFSET`.

A point sampled in screen space has an elliptical footprint in texture space. The ideal scale factor ρ should be the major axis of this ellipse.

The derivatives $\frac{\partial u}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial x}$, $\frac{\partial v}{\partial y}$, $\frac{\partial w}{\partial x}$, and $\frac{\partial w}{\partial y}$ are used to calculate the value of ρ for a fragment with window coordinates (x, y).

Computing the ellipse major axis using the derivatives in x and y can be impractical to implement. Therefore, an implementation may approximate the ideal ρ with a function $f(x, y)$ subject to these conditions:

1. $f(x, y)$ is continuous and monotonically increasing in each of $\left|\frac{\partial u}{\partial x}\right|$, $\left|\frac{\partial u}{\partial y}\right|$, $\left|\frac{\partial v}{\partial x}\right|$, $\left|\frac{\partial v}{\partial y}\right|$, $\left|\frac{\partial w}{\partial x}\right|$, and $\left|\frac{\partial w}{\partial y}\right|$

2. $\max(\left|\frac{\partial u}{\partial x}\right|, \left|\frac{\partial u}{\partial y}\right|, \left|\frac{\partial v}{\partial x}\right|, \left|\frac{\partial v}{\partial y}\right|, \left|\frac{\partial w}{\partial x}\right|, \left|\frac{\partial w}{\partial y}\right|) \leq f(x,y) \leq \sqrt{2}\max(\left|\frac{\partial u}{\partial x}\right| + \left|\frac{\partial v}{\partial x}\right| + \left|\frac{\partial w}{\partial x}\right|, \left|\frac{\partial u}{\partial y}\right| + \left|\frac{\partial v}{\partial y}\right| + \left|\frac{\partial w}{\partial y}\right|)$

For a polygon, or for a point sprite with texture coordinate replacement enabled, ρ is given at a fragment with window coordinates (x, y) by

$$\rho = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2}\right\} \tag{8.12}$$

where $\partial u/\partial x$ indicates the derivative of u with respect to window x, and similarly for the other derivatives.

For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2} \Big/ l, \tag{8.13}$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$ with (x_1, y_1) and (x_2, y_2) being the segment's window coordinate endpoints and $l = \sqrt{\Delta x^2 + \Delta y^2}$.

For a point, point sprite without texture coordinate replacement enabled, pixel rectangle, or bitmap, $\rho = 1$.

While it is generally agreed that equations 8.12 and 8.13 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal ρ with a function $f(x, y)$ subject to these conditions:

1. $f(x, y)$ is continuous and monotonically increasing in each of $|\partial u/\partial x|$, $|\partial u/\partial y|$, $|\partial v/\partial x|$, $|\partial v/\partial y|$, $|\partial w/\partial x|$, and $|\partial w/\partial y|$

2. Let

$$m_u = \max\left\{\left|\frac{\partial u}{\partial x}\right|, \left|\frac{\partial u}{\partial y}\right|\right\}$$

$$m_v = \max\left\{\left|\frac{\partial v}{\partial x}\right|, \left|\frac{\partial v}{\partial y}\right|\right\}$$

$$m_w = \max\left\{\left|\frac{\partial w}{\partial x}\right|, \left|\frac{\partial w}{\partial y}\right|\right\}.$$

Then $\max\{m_u, m_v, m_w\} \leq f(x, y) \leq m_u + m_v + m_w$.

8.14.2 Coordinate Wrapping and Texel Selection

After generating $u(x, y)$, $v(x, y)$, and $w(x, y)$, they may be clamped and wrapped before sampling the texture, depending on the corresponding texture wrap modes.

Let

$$u'(x,y) = \begin{cases} clamp(u(x,y), 0, w_t), & \texttt{TEXTURE_WRAP_S} \text{ is } \texttt{CLAMP} \\ u(x,y), & \text{otherwise} \end{cases}$$
$$v'(x,y) = \begin{cases} clamp(v(x,y), 0, h_t), & \texttt{TEXTURE_WRAP_T} \text{ is } \texttt{CLAMP} \\ v(x,y), & \text{otherwise} \end{cases}$$
$$w'(x,y) = \begin{cases} clamp(w(x,y), 0, d_t), & \texttt{TEXTURE_WRAP_R} \text{ is } \texttt{CLAMP} \\ w(x,y), & \text{otherwise} \end{cases}$$

where $clamp(a, b, c)$ returns b if $a < b$, c if $a > c$, and a otherwise.

The value assigned to TEXTURE_MIN_FILTER is used to determine how the texture value for a fragment is selected.

When the value of TEXTURE_MIN_FILTER is NEAREST, the texel in the texture image of level $level_{base}$ that is nearest (in Manhattan distance) to (u', v', w') is obtained. Let (i, j, k) be integers such that

$$i = wrap(\lfloor u'(x,y) \rfloor)$$
$$j = wrap(\lfloor v'(x,y) \rfloor)$$
$$k = wrap(\lfloor w'(x,y) \rfloor)$$

and the value returned by $wrap()$ is defined in table 8.28. For a three-dimensional texture, the texel at location (i, j, k) becomes the texture value. For two-dimensional, two-dimensional array, rectangle, or cube map textures, k is irrelevant, and the texel at location (i, j) becomes the texture value. For one-dimensional texture or one-dimensional array textures, j and k are irrelevant, and the texel at location i becomes the texture value.

For one- and two-dimensional array textures, the texel is obtained from image layer l, where

$$l = \begin{cases} clamp(RNE(t), 0, h_t - 1), & \text{for one-dimensional array textures} \\ clamp(RNE(r), 0, d_t - 1), & \text{for two-dimensional array textures} \end{cases}$$ [4]

and $RNE()$ is the round-to-nearest-even operation defined by IEEE arithmetic.

Wrap mode	Result of $wrap(coord)$
CLAMP	$\begin{cases} clamp(coord, 0, size-1), & \text{for NEAREST filtering} \\ clamp(coord, -1, size), & \text{for LINEAR filtering} \end{cases}$
CLAMP_TO_EDGE	$clamp(coord, 0, size-1)$
CLAMP_TO_BORDER	$clamp(coord, -1, size)$
REPEAT	$coord \bmod size$
MIRRORED_REPEAT	$(size-1) - mirror(coord \bmod (2 \times size)) - size)$
MIRROR_CLAMP_TO_EDGE	$clamp(mirror(coord), 0, size-1)$

Table 8.28: Texel location wrap mode application. $mirror(a)$ returns a if $a \geq 0$, and $-(1+a)$ otherwise. The values of $mode$ and $size$ are TEXTURE_WRAP_S and w_t, TEXTURE_WRAP_T and h_t, and TEXTURE_WRAP_R and d_t when wrapping i, j, or k coordinates, respectively.

If the selected (i, j, k), (i, j), or i location refers to a border texel that satisfies any of the conditions

$$\begin{aligned} i &< -b_s & \qquad i &\geq w_t + b_s \\ j &< -b_s & \qquad j &\geq h_t + b_s \\ k &< -b_s & \qquad k &\geq d_t + b_s \end{aligned}$$

then the border values defined by TEXTURE_BORDER_COLOR are used in place of the non-existent texel. If the texture contains color components, the values of TEXTURE_BORDER_COLOR are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 8.18. The internal data type of the border values must be consistent with the type returned by the texture as described in chapter 8, or the result is undefined. If border values are out-of-range with respect to the texture's internal format, the result is undefined. If the texture contains depth components, the first component of TEXTURE_BORDER_COLOR is interpreted as a depth value.

When the value of TEXTURE_MIN_FILTER is LINEAR, a $2 \times 2 \times 2$ cube of texels in the texture image of level $level_{base}$ is selected. Let

[4] Implementations may instead round the texture layer using the nearly equivalent computation $\lfloor value + \frac{1}{2} \rfloor$.

$$
\begin{aligned}
i_0 &= wrap(\lfloor u' - \frac{1}{2} \rfloor) \\
j_0 &= wrap(\lfloor v' - \frac{1}{2} \rfloor) \\
k_0 &= wrap(\lfloor w' - \frac{1}{2} \rfloor) \\
i_1 &= wrap(\lfloor u' - \frac{1}{2} \rfloor + 1) \\
j_1 &= wrap(\lfloor v' - \frac{1}{2} \rfloor + 1) \\
k_1 &= wrap(\lfloor w' - \frac{1}{2} \rfloor + 1) \\
\alpha &= frac(u' - \frac{1}{2}) \\
\beta &= frac(v' - \frac{1}{2}) \\
\gamma &= frac(w' - \frac{1}{2})
\end{aligned}
$$

where $frac(x)$ denotes the fractional part of x.

For a three-dimensional texture, the texture value τ is found as

$$
\begin{aligned}
\tau = &(1-\alpha)(1-\beta)(1-\gamma)\tau_{i_0 j_0 k_0} + \alpha(1-\beta)(1-\gamma)\tau_{i_1 j_0 k_0} \\
&+ (1-\alpha)\beta(1-\gamma)\tau_{i_0 j_1 k_0} + \alpha\beta(1-\gamma)\tau_{i_1 j_1 k_0} \\
&+ (1-\alpha)(1-\beta)\gamma\tau_{i_0 j_0 k_1} + \alpha(1-\beta)\gamma\tau_{i_1 j_0 k_1} \\
&+ (1-\alpha)\beta\gamma\tau_{i_0 j_1 k_1} + \alpha\beta\gamma\tau_{i_1 j_1 k_1}
\end{aligned}
\tag{8.14}
$$

where τ_{ijk} is the texel at location (i, j, k) in the three-dimensional texture image.

For a two-dimensional, two-dimensional array, rectangle, or cube map texture,

$$
\begin{aligned}
\tau = &(1-\alpha)(1-\beta)\tau_{i_0 j_0} + \alpha(1-\beta)\tau_{i_1 j_0} \\
&+ (1-\alpha)\beta\tau_{i_0 j_1} + \alpha\beta\tau_{i_1 j_1}
\end{aligned}
$$

where τ_{ij} is the texel at location (i, j) in the two-dimensional texture image. For two-dimensional array textures, all texels are obtained from layer l, where

$$
l = clamp(\left\lfloor r + \frac{1}{2} \right\rfloor, 0, d_t - 1).
$$

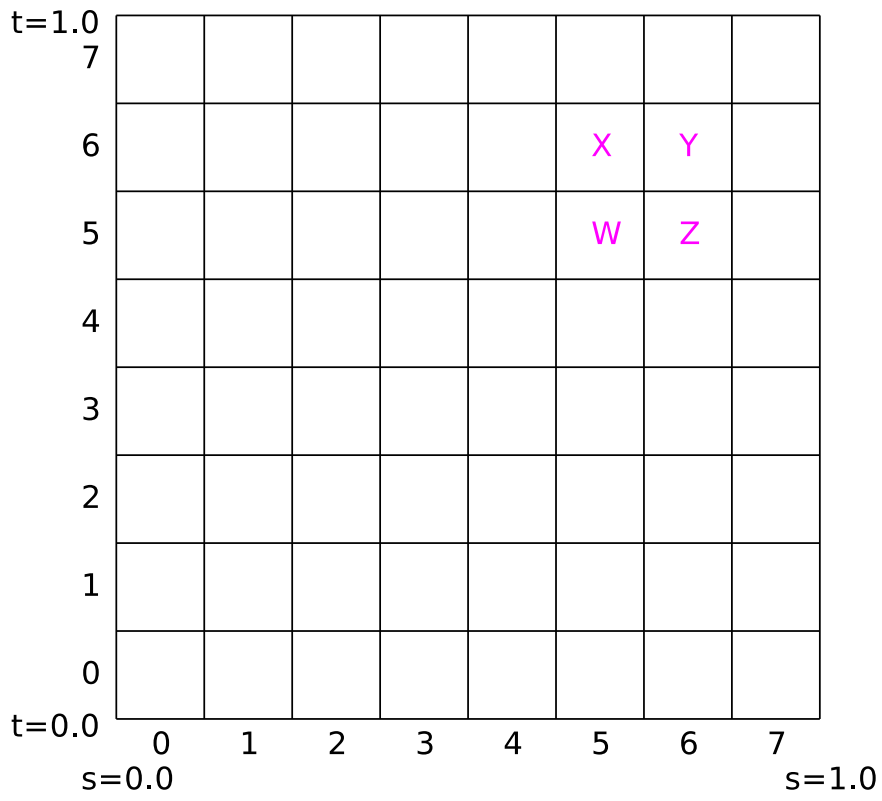


Figure 8.4. An example of an 8×8 texture image and the components returned for `textureGather`. The vector (X, Y, Z, W) is returned, where each component is taken from the post-swizzle R component of the corresponding texel.

The `textureGather` and `textureGatherOffset` built-in shader functions return a vector derived from sampling a 2×2 block of texels in the texture image of level $level_{base}$. The rules for the `LINEAR` minification filter are applied to identify the four selected texels. Each texel is then converted to a texture source color (R_s, G_s, B_s, A_s) according to table 16.1 and then swizzled as described in section 16.1. A four-component vector is then assembled by taking a single component from the swizzled texture source colors of the four texels, in the order $\tau_{i_0 j_1}$, $\tau_{i_1 j_1}$, $\tau_{i_1 j_0}$, and $\tau_{i_0 j_0}$ (see figure 8.4). The selected component is identified by the optional *comp* argument, where the values zero, one, two, and three identify the R_s, G_s, B_s, or A_s component, respectively. If *comp* is omitted, it is treated as identifying the R_s component. Incomplete textures (see section 8.17) are considered to return a texture source color of $(0.0, 0.0, 0.0, 1.0)$ in floating-point format for all four source texels.

The `textureGatherOffsets` functions operate identically to `textureGather`, except that the array of two-component integer vectors *offsets* is used to determine the location of the four texels to sample. Each of the four texels is obtained by applying the corresponding offset in the four-element array *offsets* as a (u, v) coordinate offset to the coordinates *coord*, identifying the four-texel `LINEAR` footprint, and then selecting the texel $\tau_{i_0 j_0}$ of that footprint. The specified values in *offsets* must be constant. A limited range of offset values are supported; the minimum and maximum offset values are implementation-dependent and given by the values of `MIN_PROGRAM_TEXTURE_GATHER_OFFSET` and `MAX_PROGRAM_TEXTURE_GATHER_OFFSET`, respectively. Note that *offset* does not apply to the layer coordinate for array textures.

And for a one-dimensional or one-dimensional array texture,

$$\tau = (1 - \alpha)\tau_{i_0} + \alpha\tau_{i_1}$$

where τ_i is the texel at location i in the one-dimensional texture. For one-dimensional array textures, both texels are obtained from layer l, where

$$l = clamp(\left\lfloor t + \frac{1}{2} \right\rfloor, 0, h_t - 1).$$

For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with `NEAREST` filtering.

8.14.2.1 Rendering Feedback Loops

If all of the following conditions are satisfied, then the value of the selected τ_{ijk}, τ_{ij}, or τ_i in the above equations is undefined instead of referring to the value of the

texel at location (i, j, k), (i, j), or (i) respectively. This situation is discussed in more detail in the description of feedback loops in section 9.3.1.

- The current DRAW_FRAMEBUFFER_BINDING names a framebuffer object *F*.
- The texture is attached to one of the attachment points, *A*, of framebuffer object *F*.
- The value of TEXTURE_MIN_FILTER is NEAREST or LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point *A* is equal to $level_{base}$

  -or-

  The value of TEXTURE_MIN_FILTER is NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point *A* is within the inclusive range from $level_{base}$ to q.

8.14.3 Mipmapping

TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, and LINEAR_MIPMAP_LINEAR each require the use of a *mipmap*. Rectangle textures do not support mipmapping (it is an error to specify a minification filter that requires mipmapping). A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the texture image of level $level_{base}$ (excluding its border) has dimensions $w_t \times h_t \times d_t$, then there are $\lfloor \log_2(maxsize) \rfloor + 1$ levels in the mipmap, where

$$maxsize = \begin{cases} w_t, & \text{for 1D and 1D array textures} \\ \max(w_t, h_t), & \text{for 2D, 2D array, cube map, and cube map array textures} \\ \max(w_t, h_t, d_t), & \text{for 3D textures} \end{cases}$$

Numbering the levels such that level $level_{base}$ is the 0th level, the ith array has dimensions

$$\max(1, \left\lfloor \frac{w_t}{w_d} \right\rfloor) \times \max(1, \left\lfloor \frac{h_t}{h_d} \right\rfloor) \times \max(1, \left\lfloor \frac{d_t}{d_d} \right\rfloor)$$

where

$$w_d = 2^i$$
$$h_d = \begin{cases} 1, & \text{for 1D and 1D array textures} \\ 2^i, & \text{otherwise} \end{cases}$$
$$d_d = \begin{cases} 2^i, & \text{for 3D textures} \\ 1, & \text{otherwise} \end{cases}$$

until the last array is reached with dimension $1 \times 1 \times 1$.

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, **TexImage1D**, or **CopyTexImage1D** or by functions that are defined in terms of these functions. The array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from $level_{base}$ for the original texture image through the maximum level p, with each unit increase indicating an array of half the dimensions of the previous one (rounded down to the next integer if fractional) as already described. For immutable-format textures (see section 8.19), $level_{base}$ is clamped to the range $[0, level_{immut} - 1]$, $level_{max}$ is then clamped to the range $[level_{base}, level_{immut} - 1]$, and p is one less than $level_{immut}$, where $level_{immut}$ is the *levels* parameter passed to **TexStorage*** for the texture object (the value of `TEXTURE_IMMUTABLE_LEVELS`; see section 8.19). Otherwise $p = \lfloor \log_2(maxsize) \rfloor + level_{base}$, and all arrays from $level_{base}$ through $q = \min(p, level_{max})$ must be defined, as discussed in section 8.17.

The mipmap is used in conjunction with the level-of-detail to approximate the application of an appropriately filtered texture to a fragment. Since this discussion pertains to minification, we are concerned only with values of λ where $\lambda > 0$.

For mipmap filters `NEAREST_MIPMAP_NEAREST` and `LINEAR_MIPMAP_NEAREST`, the dth mipmap array is selected, where

$$d = \begin{cases} level_{base}, & \lambda \leq 0 \\ nearest(\lambda), & \lambda > 0, level_{base} + \lambda \leq q + \frac{1}{2} \\ q, & \lambda > 0, level_{base} + \lambda > q + \frac{1}{2} \end{cases} \tag{8.15}$$

where

$$nearest(\lambda) = \begin{cases} \lceil level_{base} + \lambda + \frac{1}{2} \rceil - 1, & \text{preferred} \\ \lfloor level_{base} + \lambda + \frac{1}{2} \rfloor, & \text{alternative} \end{cases}$$

The rules for NEAREST or LINEAR filtering are then applied to the selected array. Specifically, the coordinate (u, v, w) is computed as in equation 8.11, with w_s, h_s, and d_s equal to the width, height, and depth of the texture image whose level is d.

For mipmap filters NEAREST_MIPMAP_LINEAR and LINEAR_MIPMAP_LINEAR, the level d_1 and d_2 mipmap arrays are selected, where

$$d_1 = \begin{cases} q, & level_{base} + \lambda \geq q \\ \lfloor level_{base} + \lambda \rfloor, & \text{otherwise} \end{cases} \tag{8.16}$$

$$d_2 = \begin{cases} q, & level_{base} + \lambda \geq q \\ d_1 + 1, & \text{otherwise} \end{cases} \tag{8.17}$$

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture values τ_1 and τ_2. Specifically, for level d_1, the coordinate (u, v, w) is computed as in equation 8.11, with w_s, h_s, and d_s equal to the width, height, and depth of the texture image whose level is d_1. For level d_2 the coordinate (u', v', w') is computed as in equation 8.11, with w_s, h_s, and d_s equal to the width, height, and depth of the texture image whose level is d_2.

The final texture value is then found as

$$\tau = [1 - \text{frac}(\lambda)]\tau_1 + \text{frac}(\lambda)\tau_2.$$

8.14.4 Manual Mipmap Generation

Mipmaps can be generated manually for a texture object with the commands

```
void GenerateMipmap( enum target );
void GenerateTextureMipmap( uint texture );
```

For **GenerateMipmap**, the texture object is that bound to *target*. For **GenerateTextureMipmap**, *texture* is the name of the texture object.

target or the effective target of *texture* must be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP, or TEXTURE_CUBE_MAP_ARRAY.

If *target* or the effective target of *texture* is TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY, then the texture object must be cube complete or cube array complete respectively, as defined in section 8.17. Otherwise, if

$level_{base}$ is not defined, or if any dimension is zero, all mipmap levels are left unchanged. This is not an error.

Mipmap generation replaces texture image levels $level_{base} + 1$ through q with images derived from the $level_{base}$ image, regardless of their previous contents. All other mipmap levels, including $level_{base}$, are left unchanged by this computation.

The internal formats and border widths of the derived mipmap texture images all match those of the $level_{base}$ image, and the dimensions of the derived images, follow the requirements described in section 8.17.

The contents of the derived images are computed by repeated, filtered reduction of the $level_{base}$ image. For one- and two-dimensional array and cube map array textures, each layer is filtered independently. No particular filter algorithm is required, though a box filter is recommended as the default filter. In some implementations, filter quality may be affected by hints (section 21.5).

Any synchronization required before performing this reduction will be done within the **Generate*Mipmap** commands themselves[5].

Errors

An `INVALID_ENUM` error is generated by **GenerateMipmap** if *target* is not one of the valid targets listed above.

An `INVALID_OPERATION` error is generated by **GenerateTextureMipmap** if *texture* is not the name of an existing texture object.

An `INVALID_OPERATION` error is generated by **GenerateTextureMipmap** if the effective *target* is not one of the valid targets listed above.

An `INVALID_OPERATION` error is generated by **GenerateTextureMipmap** if the effective *target* is `TEXTURE_CUBE_MAP` or `TEXTURE_CUBE_MAP_ARRAY`, and the texture object is not cube complete or cube array complete, respectively.

8.14.5 Automatic Mipmap Generation

If the value of texture parameter `GENERATE_MIPMAP` is `TRUE`, and a change is made to the interior or border texels of the $level_{base}$ texture image of a mipmap by one of the texture image specification operations defined in sections 8.5

[5] For example, if mipmaps were generated by texture fetches, a fetch barrier would be issued prior to reduction; or if mipmaps were generated on the CPU, a texture update barrier would be issued prior to reduction.

through 8.7, then a[6] mipmap complete set of mipmap texture images (as defined in section 8.17) will be computed (if the image being changed is one face of a cube map texture, only mipmaps for that face will be computed). Image levels $level_{base} + 1$ through p are replaced with images derived from the modified $level_{base}$ image, as described above for **Manual Mipmap Generation**. All other mipmap images, including the $level_{base}$ image, are left unchanged by this computation. For images in the range $level_{base} + 1$ through q, inclusive, automatic and manual mipmap generation generate the same derived images, given identical $level_{base}$ images.

Automatic mipmap generation is available only for non-proxy texture image targets.

8.15 Texture Magnification

When λ indicates magnification, the value assigned to `TEXTURE_MAG_FILTER` determines how the texture value is obtained. There are two possible values for `TEXTURE_MAG_FILTER`: `NEAREST` and `LINEAR`. `NEAREST` behaves exactly as `NEAREST` for `TEXTURE_MIN_FILTER` and `LINEAR` behaves exactly as `LINEAR` for `TEXTURE_MIN_FILTER` as described in section 8.14, including the texture coordinate wrap modes specified in table 8.28. The level-of-detail $level_{base}$ texture image is always used for magnification.

8.16 Combined Depth/Stencil Textures

If the texture image has a base internal format of `DEPTH_STENCIL`, then the stencil index texture component is ignored by default. The texture value τ does not include a stencil index component, but includes only the depth component.

In order to access the stencil index texture component the `DEPTH_STENCIL_TEXTURE_MODE` texture parameter should be set to `STENCIL_INDEX`. When this mode is set the depth component is ignored and the texture value includes only the stencil index component. The stencil index value is treated as an unsigned integer texture and returns an unsigned integer value when sampled. When sampling the stencil index only `NEAREST` filtering is supported. The `DEPTH_STENCIL_TEXTURE_MODE` is ignored for non depth/stencil textures.

[6] Automatic mipmap generation is not performed for changes resulting from rendering operations targeting a texture image bound as a color buffer of a framebuffer object.

8.17 Texture Completeness

A texture is said to be *complete* if the texture images and (where applicable) texture parameters required to utilize the texture are consistently defined. Different operations on the texture require different levels of completeness and the definitions vary depending on texture dimensionality and type.

A cube map texture is *cube complete* if:

- The $level_{base}$ images of each of the six texture images making up the cube map have identical, positive, and square dimensions.
- The $level_{base}$ images were each specified with the same effective internal format.
- The $level_{base}$ images each have the same border width.

A cube map array texture is *cube array complete* if every slice within the array is cube complete.

A texture is *base complete* if:

- All dimensions of the $level_{base}$ images are positive. For a multisample texture, $level_{base}$ is always zero.
- If the texture is a cube map texture, then it is cube complete.
- If the texture is a cube map array texture, then it is cube array complete.

A texture is *mipmap complete* if:

- It is base complete
- For every base level image, the set of mipmap images $level_{base}$ through q (where q is defined in section 8.14.3) were each specified with the same effective internal format.
- The dimensions of the images follow the sequence described in section 8.14.3.
- $level_{base} \leq level_{max}$
- The border widths of each image are the same.

Image levels k where $k < level_{max}$ or $k > q$ are insignificant to the definition of mipmap completeness.

A texture is complete unless any of the following conditions hold true:

- The texture is not base complete.
- The minification filter requires a mimap (is neither NEAREST nor LINEAR), the texture is not multisample, and the texture is not mipmap complete.
- The texture is not multisample; either the magnification filter is not NEAREST, or the minification filter is neither NEAREST nor NEAREST_MIPMAP_NEAREST; and any of
  - The internal format of the texture is integer (see table 8.19).
  - The internal format of the texture is DEPTH_STENCIL, and the value of DEPTH_STENCIL_TEXTURE_MODE for the texture is STENCIL_INDEX.
  - The internal format is STENCIL_INDEX.

8.17.1 Effects of Sampler Objects on Texture Completeness

If a sampler object and a texture object are simultaneously bound to the same texture unit, then the sampling state for that unit is taken from the sampler object (see section 8.2). This can have an effect on the effective completeness of the texture. In particular, if the texture is not mipmap complete and the sampler object specifies a TEXTURE_MIN_FILTER requiring mipmaps, the texture will be considered incomplete for the purposes of that texture unit. However, if the sampler object does not require mipmaps, the texture object will be considered complete. This means that a texture can be considered both complete and incomplete simultaneously if it is bound to two or more texture units along with sampler objects with different states.

8.17.2 Effects of Completeness on Texture Application

Texture lookup and texture fetch operations performed in shaders are affected by completeness of the texture being sampled as described in sections 11.1.3.5 and 15.2.1.

For fixed-function texture access, if texturing is enabled for a texture unit at the time a primitive is rasterized, and if the texture image bound to the enabled texture target is not complete, then it is as if texture mapping were disabled for that texture unit.

8.17.3 Effects of Completeness on Texture Image Specification

The implementation-dependent maximum sizes for texture images depend on the texture level. In particular, an implementation may allow a texture image of level

one or greater to be created only if a mipmap complete set of images consistent with the requested image can be supported with default values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL (see table 23.19). As a result, implementations may permit a texture image at level zero that will never be mipmap complete and can only be used with non-mipmapped minification filters.

8.18 Texture Views

A texture can be created which references the data store of another texture and interprets the data with a different format, and/or selects a subset of the levels and/or layers of the other texture. The data store for such a texture is shared with the data store of the original texture. Updating the shared data store using the original texture affects texture values read using the new texture, and vice versa. A texture data store remains in existence until all textures that reference it are deleted.

The command

```
void TextureView( uint texture, enum target,
    uint origtexture, enum internalformat, uint minlevel,
    uint numlevels, uint minlayer, uint numlayers );
```

initializes the texture named *texture* to the target specified by *target*. *texture*'s data store is inherited from the texture named *origtexture*, but elements of the data store are interpreted according to the internal format specified by *internalformat*. Additionally, if *origtexture* is an array or has multiple mipmap levels, the parameters *minlayer*, *numlayers*, *minlevel*, and *numlevels* control which of those slices and levels are considered part of the texture.

The *minlevel* and *minlayer* parameters are relative to the view of *origtexture*. If *numlayers* or *numlevels* extend beyond *origtexture*, they are clamped to the maximum extent of the original texture.

If the command is successful, the texture parameters in *texture* are updated as follows:

- TEXTURE_IMMUTABLE_FORMAT is set to TRUE.
- TEXTURE_IMMUTABLE_LEVELS is set to the value of TEXTURE_IMMUTABLE_LEVELS for *origtexture*.
- TEXTURE_VIEW_MIN_LEVEL is set to *minlevel* plus the value of TEXTURE_VIEW_MIN_LEVEL for *origtexture*.
- TEXTURE_VIEW_MIN_LAYER is set to *minlayer* plus the value of TEXTURE_VIEW_MIN_LAYER for *origtexture*.

Original target	Valid new targets
TEXTURE_1D	TEXTURE_1D, TEXTURE_1D_ARRAY
TEXTURE_2D	TEXTURE_2D, TEXTURE_2D_ARRAY
TEXTURE_3D	TEXTURE_3D
TEXTURE_CUBE_MAP	TEXTURE_CUBE_MAP, TEXTURE_2D, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY
TEXTURE_RECTANGLE	TEXTURE_RECTANGLE
TEXTURE_BUFFER	*none*
TEXTURE_1D_ARRAY	TEXTURE_1D_ARRAY, TEXTURE_1D
TEXTURE_2D_ARRAY	TEXTURE_2D_ARRAY, TEXTURE_2D, TEXTURE_CUBE_MAP, TEXTURE_CUBE_MAP_ARRAY
TEXTURE_CUBE_MAP_ARRAY	TEXTURE_CUBE_MAP_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_2D, TEXTURE_CUBE_MAP
TEXTURE_2D_MULTISAMPLE	TEXTURE_2D_MULTISAMPLE, TEXTURE_2D_MULTISAMPLE_ARRAY
TEXTURE_2D_MULTISAMPLE_ARRAY	TEXTURE_2D_MULTISAMPLE, TEXTURE_2D_MULTISAMPLE_ARRAY

Table 8.29: Legal texture targets for **TextureView**.

- TEXTURE_VIEW_NUM_LEVELS is set to the lesser of *numlevels* and the value of TEXTURE_VIEW_NUM_LEVELS for *origtexture* minus *minlevel*.
- TEXTURE_VIEW_NUM_LAYERS is set to the lesser of *numlayers* and the value of TEXTURE_VIEW_NUM_LAYERS for *origtexture* minus *minlayer*.

The new texture's target must be *compatible* with the target of *origtexture*, as defined by table 8.29.

Numerous constraints on *numlayers* and the texture dimensions depend on *target* and the target of *origtexture*. These constraints are summarized below in the errors section.

When *origtexture*'s target is TEXTURE_CUBE_MAP, the layer parameters are interpreted in the same order as if it were a TEXTURE_CUBE_MAP_ARRAY with 6 layer-faces.

The two textures' internal formats must be compatible according to table 8.30 if the internal format exists in that table. The internal formats must be identical if not in that table. If the internal formats are the same but are a base internal format,

Class	Internal formats
VIEW_CLASS_128_BITS	RGBA32F, RGBA32UI, RGBA32I
VIEW_CLASS_96_BITS	RGB32F, RGB32UI, RGB32I
VIEW_CLASS_64_BITS	RGBA16F, RG32F, RGBA16UI, RG32UI, RGBA16I, RG32I, RGBA16, RGBA16_SNORM
VIEW_CLASS_48_BITS	RGB16, RGB16_SNORM, RGB16F, RGB16UI, RGB16I
VIEW_CLASS_32_BITS	RG16F, R11F_G11F_B10F, R32F, RGB10_A2UI, RGBA8UI, RG16UI, R32UI, RGBA8I, RG16I, R32I, RGB10_A2, RGBA8, RG16, RGBA8_SNORM, RG16_SNORM, SRGB8_ALPHA8, RGB9_E5
VIEW_CLASS_24_BITS	RGB8, RGB8_SNORM, SRGB8, RGB8UI, RGB8I
VIEW_CLASS_16_BITS	R16F, RG8UI, R16UI, RG8I, R16I, RG8, R16, RG8_SNORM, R16_SNORM
VIEW_CLASS_8_BITS	R8UI, R8I, R8, R8_SNORM
VIEW_CLASS_RGTC1_RED	COMPRESSED_RED_RGTC1, COMPRESSED_SIGNED_RED_RGTC1
VIEW_CLASS_RGTC2_RG	COMPRESSED_RG_RGTC2, COMPRESSED_SIGNED_RG_RGTC2
VIEW_CLASS_BPTC_UNORM	COMPRESSED_RGBA_BPTC_UNORM, COMPRESSED_SRGB_ALPHA_BPTC_UNORM
VIEW_CLASS_BPTC_FLOAT	COMPRESSED_RGB_BPTC_SIGNED_FLOAT, COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT

Table 8.30: Compatible internal formats for **TextureView**. Formats in the same row may be cast to each other.

the implementation's effective internal format (see the end of section 8.5) for each texture must be the same.

If the internal format does not exactly match the internal format of the original texture, the contents of the memory are reinterpreted in the same manner as for image bindings described in section 8.26.

Texture commands that take a *level* or *layer* parameter, such as **TexSubImage2D**, interpret that parameter to be relative to the view of the texture. i.e. the mipmap level of the data store that would be updated via **TexSubImage2D** would be the sum of *level* and the value of TEXTURE_VIEW_MIN_LEVEL.

Errors

An INVALID_VALUE error is generated if *texture* is zero.
An INVALID_OPERATION error is generated if *texture* is not a valid name

returned by **GenTextures**, or if *texture* has already been bound and given a target.

An INVALID_VALUE error is generated if *origtexture* is not the name of a texture.

An INVALID_OPERATION error is generated if the value of TEXTURE_IMMUTABLE_FORMAT for *origtexture* is not TRUE.

An INVALID_OPERATION error is generated if *target* is not compatible with the target of *origtexture*, as defined by table 8.29.

An INVALID_OPERATION error is generated if the internal format of *origtexture* exists in table 8.30 and is not compatible with *internalformat*, as described in that table.

An INVALID_OPERATION error is generated if the internal format of *origtexture* does not exist in table 8.30, and is not identical to *internalformat*.

An INVALID_VALUE error is generated if *minlevel* or *minlayer* are larger than the greatest level or layer, respectively, of *origtexture*.

An INVALID_VALUE error is generated if *target* is TEXTURE_CUBE_MAP and the clamped *numlayers* is not 6.

An INVALID_VALUE error is generated if *target* is TEXTURE_CUBE_MAP_ARRAY and the clamped *numlayers* is not a multiple of 6.

An INVALID_VALUE error is generated if *target* is TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_RECTANGLE, or TEXTURE_2D_MULTISAMPLE and *numlayers* does not equal 1.

An INVALID_OPERATION error is generated if *target* is TEXTURE_CUBE_MAP or TEXTURE_CUBE_MAP_ARRAY, and the width and height of *origtexture*'s levels are not equal.

An INVALID_OPERATION error is generated if any dimension of *origtexture* is larger than the maximum supported corresponding dimension of the new target. For example, if *origtexture* has a TEXTURE_2D_ARRAY target and *target* is TEXTURE_CUBE_MAP, its width must be no greater than the value of MAX_CUBE_MAP_TEXTURE_SIZE.

An INVALID_OPERATION error is generated if the computed values of TEXTURE_VIEW_NUM_LEVELS or TEXTURE_VIEW_NUM_LAYERS for *texture*, as described above, are less than or equal to zero.

8.19 Immutable-Format Texture Images

An alternative set of commands is provided for specifying the properties of all levels of a texture at once. Once a texture is specified with such a command, the format and dimensions of all levels becomes immutable, unless it is a proxy texture

(since otherwise it would no longer be possible to use the proxy). The contents of the images and the parameters can still be modified. Such a texture is referred to as an *immutable-format* texture. The immutability status of a texture can be determined by calling **GetTexParameter** with *pname* TEXTURE_IMMUTABLE_FORMAT.

Each of the commands below is described by pseudocode which indicates the effect on the dimensions and format of the texture. For each command the following apply in addition to the pseudocode:

- If executing the pseudocode would result in any other error, the error is generated and the command will have no effect.
- Any existing levels that are not replaced are reset to their initial state.
- The pixel unpack buffer should be considered to be zero; i.e., the image contents are unspecified.
- Since no pixel data are provided, the *format* and *type* values used in the pseudocode are irrelevant; they can be considered to be any values that are legal to use with *internalformat*.
- If the command is successful, TEXTURE_IMMUTABLE_FORMAT becomes TRUE. TEXTURE_IMMUTABLE_LEVELS and TEXTURE_VIEW_NUM_LEVELS become *levels*. If the texture target is TEXTURE_1D_ARRAY then TEXTURE_VIEW_NUM_LAYERS becomes *height*. If the texture target is TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY, or TEXTURE_2D_MULTISAMPLE_ARRAY then TEXTURE_VIEW_NUM_LAYERS becomes *depth*. If the texture target is TEXTURE_CUBE_MAP, then TEXTURE_VIEW_NUM_LAYERS becomes 6. For any other texture target, TEXTURE_VIEW_NUM_LAYERS becomes 1.

The **TexStorage*** commands specify properties of the texture object bound to the *target* parameter of each command.

The **TextureStorage*** commands behave similarly to the equivalent **TexStorage*** commands, but specify properties of the texture object named by the *texture* parameter of each command. The effective target of *texture* must be compatible with the *target* parameter of the equivalent **TexStorage*** command.

For each command, the following errors are generated in addition to the errors described specific to that command:

Errors

An INVALID_OPERATION error is generated by **TexStorage*** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **TextureStorage*** if *texture* is not the name of an existing texture object.

An INVALID_VALUE error is generated if *width*, *height*, *depth* or *levels* are less than 1, for commands with the corresponding parameters.

An INVALID_ENUM error is generated if *internalformat* is one of the unsized base internal formats listed in table 8.18.

The commands

```
void TexStorage1D( enum target, sizei levels,
    enum internalformat, sizei width );
void TextureStorage1D( uint texture, sizei levels,
    enum internalformat, sizei width );
```

specify all the levels of a one-dimensional texture (or for **TexStorage1D**, proxy) at the same time. **TexStorage1D** is described by the pseudocode below:

```
for (i = 0; i < levels; i++) {
    TexImage1D(target, i, internalformat, width, 0,
        format, type, NULL);
    width = max(1, ⌊width/2⌋);
}
```

Errors

In addition to the generic errors described at the start of this section,

An INVALID_ENUM error is generated by **TexStorage1D** if *target* is not TEXTURE_1D or PROXY_TEXTURE_1D.

An INVALID_OPERATION error is generated by **TextureStorage1D** if the effective *target* is not TEXTURE_1D.

An INVALID_OPERATION error is generated if *levels* is greater than $\lfloor \log_2(width) \rfloor + 1$.

The commands

```
void TexStorage2D( enum target, sizei levels,
    enum internalformat, sizei width, sizei height );
```

```
void TextureStorage2D( uint texture, sizei levels,
    enum internalformat, sizei width, sizei height );
```

specify all the levels of a two-dimensional, cube map, one-dimensional array or rectangle texture (or for **TexStorage2D**, proxy) at the same time. **TexStorage2D** is described by the *target*-dependent pseudocode below:

*target*s TEXTURE_2D, PROXY_TEXTURE_2D, TEXTURE_RECTANGLE, PROXY_TEXTURE_RECTANGLE, or PROXY_TEXTURE_CUBE_MAP:

```
for (i = 0; i < levels; i++) {
    TexImage2D(target, i, internalformat, width, height, 0,
            format, type, NULL);
    width = max(1, ⌊width/2⌋);
    height = max(1, ⌊height/2⌋);
}
```

target TEXTURE_CUBE_MAP:

```
for (i = 0; i < levels; i++) {
    for face in (each target in table 8.27) {
        TexImage2D(face, i, internalformat, width, height, 0,
            format, type, NULL);
    }
    width = max(1, ⌊width/2⌋);
    height = max(1, ⌊height/2⌋);
}
```

*target*s TEXTURE_1D_ARRAY or PROXY_TEXTURE_1D_ARRAY:

```
for (i = 0; i < levels; i++) {
    TexImage2D(target, i, internalformat, width, height, 0,
        format, type, NULL);
    width = max(1, ⌊width/2⌋);
}
```

Errors

In addition to the generic errors described at the start of this section, An INVALID_ENUM error is generated by **TexStorage2D** if *target* is not

one of those listed above.

An INVALID_OPERATION error is generated by **TextureStorage2D** if the effective *target* is not one of those listed above.

An INVALID_OPERATION error is generated if any of the following conditions hold:

- The effective *target* is TEXTURE_1D_ARRAY or PROXY_TEXTURE_1D_ARRAY, and *levels* is greater than $\lfloor \log_2(width) \rfloor + 1$
- The effective *target* is not TEXTURE_1D_ARRAY or PROXY_TEXTURE_1D_ARRAY, and *levels* is greater than $\lfloor \log_2(\max(width, height)) \rfloor + 1$

An INVALID_OPERATION error is generated by **TexStorage2D** and **TextureStorage2D** if *internalformat* is one of the EAC, ETC2, or RGTC formats and the effective *target* is TEXTURE_RECTANGLE.

The commands

```
void TexStorage3D( enum target, sizei levels,
    enum internalformat, sizei width, sizei height,
    sizei depth );
void TextureStorage3D( uint texture, sizei levels,
    enum internalformat, sizei width, sizei height,
    sizei depth );
```

specify all the levels of a three-dimensional, two-dimensional array texture, or cube map array texture (or for **TexStorage3D**, proxy). **TexStorage3D** is described by the *target*-dependent pseudocode below:

*target*s TEXTURE_3D or PROXY_TEXTURE_3D:

```
for (i = 0; i < levels; i++) {
    TexImage3D(target, i, internalformat, width, height, depth, 0,
        format, type, NULL);
    width = max(1, ⌊width/2⌋);
    height = max(1, ⌊height/2⌋);
    depth = max(1, ⌊depth/2⌋);
}
```

*target*s TEXTURE_2D_ARRAY, PROXY_TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY or PROXY_TEXTURE_CUBE_MAP_ARRAY:

```
for (i = 0; i < levels; i++) {
   TexImage3D(target, i, internalformat, width, height, depth, 0,
      format, type, NULL);
   width = max(1, ⌊width/2⌋);
   height = max(1, ⌊height/2⌋);
}
```

Errors

In addition to the generic errors described at the start of this section,

An `INVALID_ENUM` error is generated by **TexStorage3D** if *target* is not one of those listed above.

An `INVALID_OPERATION` error is generated by **TextureStorage3D** if the effective *target* is not one of those listed above.

An `INVALID_OPERATION` error is generated if any of the following conditions hold:

- The effective *target* is `TEXTURE_3D` or `PROXY_TEXTURE_3D` and *levels* is greater than $\lfloor \log_2(\max(width, height, depth)) \rfloor + 1$
- The effective *target* is `TEXTURE_2D_ARRAY`, `PROXY_TEXTURE_2D_ARRAY`, `TEXTURE_CUBE_MAP_ARRAY` or `PROXY_TEXTURE_CUBE_MAP_ARRAY` and *levels* is greater than $\lfloor \log_2(\max(width, height)) \rfloor + 1$
-

The commands

```
void TexStorage2DMultisample( enum target, sizei samples,
   enum internalformat, sizei width, sizei height,
   boolean fixedsamplelocations );
void TextureStorage2DMultisample( uint texture,
   sizei samples, enum internalformat, sizei width,
   sizei height, boolean fixedsamplelocations );
```

specify a two-dimensional multisample texture (or for **TexStorage2DMultisample**, proxy). For **TexStorage2DMultisample**, *target* must be `TEXTURE_2D_MULTISAMPLE` or `PROXY_TEXTURE_2D_MULTISAMPLE`.

Calling **TexStorage2DMultisample** is equivalent to calling **TexImage2DMultisample** with the equivalently named parameters set to the same values, except that the resulting texture has immutable format.

Errors

In addition to the generic errors described at the start of this section,

An `INVALID_ENUM` error is generated by **TexStorage2DMultisample** if *target* is not `TEXTURE_2D_MULTISAMPLE` or `PROXY_TEXTURE_2D_MULTISAMPLE`.

An `INVALID_OPERATION` error is generated by **TextureStorage2DMultisample** if the effective *target* is not `TEXTURE_2D_MULTISAMPLE`.

The commands

```
void TexStorage3DMultisample( enum target, sizei samples,
    enum internalformat, sizei width, sizei height,
    sizei depth, boolean fixedsamplelocations );
void TextureStorage3DMultisample( uint texture,
    sizei samples, enum internalformat, sizei width,
    sizei height, sizei depth,
    boolean fixedsamplelocations );
```

specify a two-dimensional multisample array texture (or, for **TexStorage3DMultisample**, proxy). For **TexStorage3DMultisample**, *target* must be `TEXTURE_2D_MULTISAMPLE_ARRAY` or `PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY`.

Calling **TexStorage3DMultisample** is equivalent to calling **TexImage3DMultisample** with the equivalently named parameters set to the same values, except that the resulting texture has immutable format.

Errors

In addition to the generic errors described at the start of this section,

An `INVALID_ENUM` error is generated by **TexStorage3DMultisample** if *target* is not `TEXTURE_2D_MULTISAMPLE_ARRAY` or `PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY`.

An `INVALID_OPERATION` error is generated by **TextureStorage3DMultisample** if *target*

is not TEXTURE_2D_MULTISAMPLE_ARRAY.

8.19.1 Behavior of Immutable-Format Texture Images

After a successful call to any **Tex*Storage*** command with a non-proxy target, no further changes to the dimensions or format of the texture object may be made. Other commands may only alter the texel values and texture parameters.

Errors

An INVALID_OPERATION error is generated by any of the commands **CompressedTexImage***, **CopyTexImage***, **TexImage***, and **TexStorage*** with the same texture, even if it does not affect the dimensions or format:

8.20 Invalidating Texture Image Data

All or part of a texture image may be invalidated, effectively leaving those texels undefined, by calling

```
void InvalidateTexSubImage( uint texture, int level,
    int xoffset, int yoffset, int zoffset, sizei width,
    sizei height, sizei depth );
```

with *texture* and *level* indicating which texture image is being invalidated. After this command, data in that subregion have undefined values.

Arguments *xoffset*, *yoffset*, and *zoffset* specify the lower left back texel coordinates of a *width*-wide by *height*-high by *depth*-deep rectangular subregion of the texture image to invalidate, and are interpreted as described for **TexSubImage3D** in section 8.6. The subregion must lie within the bounds (including any texture border) of the texture image, as described in that section.

Cube map textures are treated as an array of six slices in the z-dimension, where a value of *zoffset* is interpreted as specifying the cube map face for the corresponding *layer* in table 9.3.

For texture types that do not have certain dimensions, **InvalidateTexSubImage** treats those dimensions as having a size of 1. For example, to invalidate a portion of a two-dimensional texture, use *zoffset* equal to zero and *depth* equal to one.

Errors

An INVALID_VALUE error is generated if *level* is negative or greater than

log_2 of the maximum texture width, height, or depth.

An `INVALID_OPERATION` error is generated if the specified subregion does not lie within the bounds of the texture image, as described for **TexSubImage3D** in section 8.6.

An `INVALID_OPERATION` error is generated if *texture* is zero or is not the name of a texture. It is not possible to invalidate a portion of a default texture.

An `INVALID_VALUE` error is generated if the effective target of *texture* is `TEXTURE_RECTANGLE`, `TEXTURE_BUFFER`, `TEXTURE_2D_MULTISAMPLE`, or `TEXTURE_2D_MULTISAMPLE_ARRAY`, and *level* is not zero.

An `INVALID_VALUE` error is generated if *width*, *height*, or *depth* is negative.

The command

```
void InvalidateTexImage( uint texture, int level );
```

is equivalent to calling **InvalidateTexSubImage** with *xoffset*, *yoffset*, and *zoffset* equal to $-b$ and *width*, *height*, and *depth* equal to the dimensions of the texture image plus $2 \times b$ (or zero and one for dimensions the texture doesn't have).

8.21 Clearing Texture Image Data

All or part of a texture image may be filled with a constant value with the command

```
void ClearTexSubImage( uint texture, int level,
    int xoffset, int yoffset, int zoffset, sizei width,
    sizei height, sizei depth, enum format, enum type,
    const void *data );
```

with *texture* and *level* indicating which texture image is being cleared. It is an error if *texture* is zero or not the name of a texture object, if *texture* is a buffer texture, or if the texture image has a compressed internal format.

Arguments *xoffset*, *yoffset*, and *zoffset* specify the lower left back texel coordinates of a *width*-wide by *height*-high by *depth*-deep rectangular subregion of the texture image, and are interpreted as described for **TexSubImage3D** in section 8.6. The subregion must lie within the bounds (including any texture border) of the texture image, as described in that section.

For one-dimensional array textures, *yoffset* is interpreted as the first layer to be cleared and *height* is the number of layers to clear. For two-dimensional array textures, *zoffset* is interpreted as the first layer to be cleared and *depth* is the number

of layers to clear. Cube map textures are treated as an array of six slices in the z-dimension, where the value of *zoffset* is interpreted as specifying the cube map face for the corresponding *layer* in table 9.3 and *depth* is the number of faces to clear. For cube map array textures, *zoffset* is the first layer-face to clear, and *depth* is the number of layer-faces to clear. Each layer-face is translated into an array layer and a cube map face as described for layer-face numbers in section 8.5.3.

For texture types that do not have certain dimensions, **ClearTexSubImage** treats those dimensions as having a size of 1. For example, to clear a portion of a two-dimensional texture, use *zoffset* equal to zero and *depth* equal to one.

format and *type* specify the format and type of the source data and are interpreted as they are for **TexImage3D**, as described in section 8.4.4. Textures with a base internal format of `DEPTH_COMPONENT`, `STENCIL_INDEX`, `DEPTH_STENCIL` require depth component, stencil, or depth/stencil component data respectively. Textures with other base internal formats require `RGBA` formats. Textures with integer internal formats (see table 8.19) require integer data.

data is a pointer to an array of between one and four components of texel data that will be used as the source for the constant fill value. The elements of *data* are converted by the GL into the *internalformat* of the texture image (that was specified when the level was defined by any of the **TexImage**, **TexStorage** or **CopyTexImage** commands) in the manner described in section 8.4.4, and then used to fill the specified range of the destination texture level. If *data* is `NULL`, then the pointer is ignored and the sub-range of the texture image is filled with zeros. If *texture* is a multisample texture, all the samples in a texel are cleared to the value specified by *data*.

Errors

An `INVALID_OPERATION` error is generated if *texture* is zero or not the name of a texture object.

An `INVALID_OPERATION` error is generated if *texture* is a buffer texture.

An `INVALID_OPERATION` error is generated if *texture* has a compressed internal format.

An `INVALID_VALUE` error is generated if *width*, *height*, or *depth* is negative.

An `INVALID_OPERATION` error is generated if the base internal format is `DEPTH_COMPONENT` and *format* is not `DEPTH_COMPONENT`.

An `INVALID_OPERATION` error is generated if the base internal format is `DEPTH_STENCIL` and *format* is not `DEPTH_STENCIL`.

An `INVALID_OPERATION` error is generated if the base internal format is

STENCIL_INDEX and *format* is not STENCIL_INDEX.

An INVALID_OPERATION error is generated if the base internal format is RGBA and the *format* is DEPTH_COMPONENT, STENCIL_INDEX, or DEPTH_STENCIL.

An INVALID_OPERATION error is generated if the internal format is integer and *format* does not specify integer data.

An INVALID_OPERATION error is generated if the internal format is not integer and *format* does specify integer data.

An INVALID_OPERATION error is generated if the specified subregion does not lie within the bounds of the texture image, as described for **TexSubImage3D** in section 8.6.

The command

```
void ClearTexImage( uint texture, int level, enum format,
    enum type, const void * data );
```

is equivalent to calling **ClearTexSubImage** with *xoffset*, *yoffset*, and *zoffset* equal to $-b$, and *width*, *height*, and *depth* equal to the dimensions of the texture image plus $2 \times b$ (or zero and one for dimensions the texture doesn't have). b is the border width of the texture image.

Errors

In addition to the errors generated by **ClearTexSubImage**:

An INVALID_OPERATION error is generated if the texture image identified by *level* has not previously been defined by a **TexImage*** or **TexStorage*** command.

8.22 Texture State and Proxy State

The state necessary for texture can be divided into two categories. First, there are the multiple sets of texture images (a single image for the rectangle texture target; one set of mipmap images each for the one-, two-, and three-dimensional and one- and two-dimensional array texture targets; and six sets of mipmap images each for the cube map and cube map array texture targets) and their number. Each array has associated with it a width, height (two- and three-dimensional, rectangle, one-dimensional array, cube map, and cube map array only), and depth (three-dimensional, two-dimensional array, and cube map array only), a border width, an integer describing the internal format of the image, integer values describing

the resolutions of each of the red, green, blue, alpha, luminance, intensity, depth, and stencil components of the image, integer values describing the type (unsigned normalized, integer, floating-point, etc.) of each of the components, a boolean describing whether the image is compressed or not, an integer size of a compressed image, and an integer containing the name of a buffer object bound as the data store of the image.

Each initial texture image is null. It has zero width, height, and depth, zero border width, internal format 1, or LUMINANCE8 for buffer textures, component sizes set to zero and component types set to NONE, the compressed flag set to FALSE, a zero compressed size, and the bound buffer object name is zero.

Multisample textures also contain an integer identifying the number of samples in each texel, and a boolean indicating whether identical sample locations and the same number of samples will be used for all texels in the image.

Buffer textures also contain two pointer sized integers containing the offset and range of the buffer object's data store.

Next, there are the five sets of texture properties, corresponding to the one-, two-, three-dimensional, cube map and cube map array texture targets. Each set consists of the selected minification and magnification filters, the wrap modes for s, t (two- and three-dimensional and cube map only), and r (three-dimensional only), the TEXTURE_BORDER_COLOR, two floating-point numbers describing the minimum and maximum level-of-detail, two integers describing the base and maximum mipmap image levels, a boolean flag indicating whether the texture is resident, a boolean indicating whether automatic mipmap generation should be performed, the priority associated with each set of properties, a boolean flag indicating whether the format and dimensions of the texture are immutable, three integers describing the depth texture mode, compare mode, and compare function, an integer describing the depth stencil texture mode, and four integers describing the red, green, blue, and alpha swizzle modes (see section 15.2.1). The value of the resident flag is determined by the GL and may change as a result of other GL operations. The flag may only be queried, not set, by applications (see section 8.1).

In the initial state, the value assigned to TEXTURE_MIN_FILTER is NEAREST_MIPMAP_LINEAR (except for rectangle textures, where the initial value is LINEAR), and the value for TEXTURE_MAG_FILTER is LINEAR. s, t, and r wrap modes are all set to REPEAT (except for rectangle textures, where the initial value is CLAMP_TO_EDGE). The values of TEXTURE_MIN_LOD and TEXTURE_MAX_LOD are -1000 and 1000 respectively. The values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL are 0 and 1000 respectively. The value of TEXTURE_PRIORITY is 1.0. The value of TEXTURE_BORDER_COLOR is (0,0,0,0). The value of GENERATE_MIPMAP is FALSE. The value of TEXTURE_IMMUTABLE_FORMAT is FALSE. The values of

DEPTH_TEXTURE_MODE, TEXTURE_COMPARE_MODE, and TEXTURE_COMPARE_FUNC are LUMINANCE, NONE, and LEQUAL respectively. The initial value of TEXTURE_RESIDENT is determined by the GL. The value of DEPTH_STENCIL_TEXTURE_MODE is DEPTH_COMPONENT. The values of TEXTURE_SWIZZLE_R, TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A are RED, GREEN, BLUE, and ALPHA, respectively. The values of TEXTURE_IMMUTABLE_LEVELS, TEXTURE_VIEW_MIN_LEVEL, TEXTURE_VIEW_NUM_LEVELS, TEXTURE_VIEW_MIN_LAYER, TEXTURE_VIEW_NUM_LAYERS are each zero.

In addition to texture images for the non-proxy texture targets described above, partially instantiated texture images are maintained for one-, two-, and three-dimensional, rectangle, one- and two-dimensional array, and cube map array textures. Additionally, a single proxy image is maintained for the cube map texture. Each proxy image includes width, height, depth, border width, number of samples, and internal format state values, as well as state for the red, green, blue, alpha, luminance, intensity, depth, and stencil component resolutions and types. Proxy images do not include image data or texture parameters. When **TexImage3D** is executed with *target* specified as PROXY_TEXTURE_3D, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image would not be supported by **TexImage3D** called with *target* set to TEXTURE_3D, no error is generated, but the proxy width, height, depth, border width, number of samples, and component resolutions are set to zero, and the component types are set to NONE. If the image would be supported by such a call to **TexImage3D**, the proxy state values are set exactly as though the actual image were being specified. No pixel data are transferred or processed in either case.

Proxy images for one- and two-dimensional textures, one- and two-dimensional array textures, and cube map array textures are operated on in the same way when **TexImage1D** is executed with *target* specified as PROXY_TEXTURE_1D, **TexImage2D** is executed with *target* specified as PROXY_TEXTURE_2D, PROXY_TEXTURE_1D_ARRAY, or PROXY_TEXTURE_RECTANGLE, or **TexImage3D** is executed with *target* specified as PROXY_TEXTURE_2D_ARRAY or PROXY_TEXTURE_CUBE_MAP_ARRAY.

Proxy images for two-dimensional multisample and two-dimensional multisample array textures are operated on in the same way when **TexImage2DMultisample** is called with *target* specified as PROXY_TEXTURE_2D_MULTISAMPLE, or **TexImage3DMultisample** is called with *target* specified as PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY. However, if *samples* is not supported, then no error is generated.

The cube map proxy images are operated on in the same manner when **TexImage2D** is executed with the *target* field specified as PROXY_TEXTURE_CUBE_-

MAP, with the addition that determining that a given cube map texture is supported with PROXY_TEXTURE_CUBE_MAP indicates that all six of the cube map two-dimensional images are supported. Likewise, if the specified PROXY_TEXTURE_CUBE_MAP is not supported, none of the six cube map two-dimensional images are supported.

Errors

An INVALID_ENUM error is generated by **BindTexture**, **GetTexImage**, **GetTexParameteriv**, and **GetTexParameterfv** when called with a proxy texture *target*. There is no image or non-level-related state associated with proxy textures, therefore they may not be used as textures.

8.23 Texture Comparison Modes

Texture values can also be computed according to a specified comparison function. Texture parameter TEXTURE_COMPARE_MODE specifies the comparison operands, and parameter TEXTURE_COMPARE_FUNC specifies the comparison function. The format of the resulting texture sample is determined by the value of DEPTH_TEXTURE_MODE.

8.23.1 Depth Texture Comparison Mode

If the currently bound texture's base internal format is DEPTH_COMPONENT or DEPTH_STENCIL, then DEPTH_TEXTURE_MODE, TEXTURE_COMPARE_MODE and TEXTURE_COMPARE_FUNC control the output of the texture unit as described below. Otherwise, the texture unit operates in the normal manner and texture comparison is bypassed.

Let D_t be the depth texture value and S_t be the stencil index component. If there is no stencil component, the value of S_t is undefined. Let D_{ref} be the reference value, defined as follows:

- For fixed-function, non-cubemap texture lookups, D_{ref} is the interpolated r texture coordinate.
- For fixed-function, cubemap texture lookups, D_{ref} is the interpolated q texture coordinate.
- For texture lookups generated by an OpenGL Shading Language lookup function, D_{ref} is the reference value for depth comparisons provided by the lookup function.

If the texture's internal format indicates a fixed-point depth texture, then D_t and D_{ref} are clamped to the range $[0, 1]$; otherwise no clamping is performed.

Then the effective texture value is computed as follows:

- If the base internal format is `STENCIL_INDEX`, then $r = S_t$.
- If the base internal format is `DEPTH_STENCIL` and the value of `DEPTH_STENCIL_TEXTURE_MODE` is `STENCIL_INDEX`, then $r = S_t$.
- Otherwise, if the value of `TEXTURE_COMPARE_MODE` is `NONE`, then $r = D_t$.
- Otherwise, if the value of `TEXTURE_COMPARE_MODE` is `COMPARE_REF_TO_TEXTURE`, then r depends on the texture comparison function as shown in table 8.31.

Texture Comparison Function	Computed result r
`LEQUAL`	$r = \begin{cases} 1.0, & D_{ref} \leq D_t \\ 0.0, & D_{ref} > D_t \end{cases}$
`GEQUAL`	$r = \begin{cases} 1.0, & D_{ref} \geq D_t \\ 0.0, & D_{ref} < D_t \end{cases}$
`LESS`	$r = \begin{cases} 1.0, & D_{ref} < D_t \\ 0.0, & D_{ref} \geq D_t \end{cases}$
`GREATER`	$r = \begin{cases} 1.0, & D_{ref} > D_t \\ 0.0, & D_{ref} \leq D_t \end{cases}$
`EQUAL`	$r = \begin{cases} 1.0, & D_{ref} = D_t \\ 0.0, & D_{ref} \neq D_t \end{cases}$
`NOTEQUAL`	$r = \begin{cases} 1.0, & D_{ref} \neq D_t \\ 0.0, & D_{ref} = D_t \end{cases}$
`ALWAYS`	$r = 1.0$
`NEVER`	$r = 0.0$

Table 8.31: Depth texture comparison functions.

The resulting r is assigned to R_t, L_t, I_t, or A_t if the value of `DEPTH_TEXTURE_MODE` is respectively `RED`, `LUMINANCE`, `INTENSITY`, or `ALPHA`.

If the value of `TEXTURE_MAG_FILTER` is not `NEAREST`, or the value of `TEXTURE_MIN_FILTER` is not `NEAREST` or `NEAREST_MIPMAP_NEAREST`, then r

Internal Format
SRGB
SRGB8
SRGB_ALPHA
SRGB8_ALPHA8
SLUMINANCE_ALPHA
SLUMINANCE8_ALPHA8
SLUMINANCE
SLUMINANCE8
COMPRESSED_SLUMINANCE
COMPRESSED_SLUMINANCE_ALPHA
COMPRESSED_SRGB
COMPRESSED_SRGB8_ETC2
COMPRESSED_SRGB_ALPHA
COMPRESSED_SRGB8_ALPHA8_ETC2_EAC
COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2
COMPRESSED_SRGB_ALPHA_BPTC_UNORM

Table 8.32: sRGB texture internal formats.

may be computed by comparing more than one depth texture value to the texture reference value. The details of this are implementation-dependent, but r should be a value in the range $[0, 1]$ which is proportional to the number of comparison passes or failures.

8.24 sRGB Texture Color Conversion

If the currently bound texture's internal format is one of the sRGB formats in table 8.32, the red, green, and blue components are converted from an sRGB color space to a linear color space as part of filtering described in sections 8.14 and 8.15. Any alpha component is left unchanged. Ideally, implementations should perform this color conversion on each sample prior to filtering but implementations are allowed to perform this conversion after filtering (though this post-filtering approach is inferior to converting from sRGB prior to filtering).

The conversion from an sRGB encoded component c_s to a linear component c_l

is as follows.

$$c_l = \begin{cases} \frac{c_s}{12.92}, & c_s \leq 0.04045 \\ \left(\frac{c_s+0.055}{1.055}\right)^{2.4}, & c_s > 0.04045 \end{cases} \tag{8.18}$$

Assume c_s is the sRGB component in the range $[0, 1]$.

8.25 Shared Exponent Texture Color Conversion

If the currently bound texture's internal format is `RGB9_E5`, the red, green, blue, and shared bits are converted to color components (prior to filtering) using shared exponent decoding. The component red_s, $green_s$, $blue_s$, and exp_s values (see section 8.5.2) are treated as unsigned integers and are converted to floating-point red, $green$, and $blue$ as follows:

$$red = red_s 2^{exp_s - B - N}$$
$$green = green_s 2^{exp_s - B - N}$$
$$blue = blue_s 2^{exp_s - B - N}$$

8.26 Texture Image Loads and Stores

The contents of a texture may be made available for shaders to read and write by binding the texture to one of a collection of image units. The GL implementation provides an array of image units numbered beginning with zero, with the total number of image units provided determined by the implementation-dependent value of `MAX_IMAGE_UNITS`. Unlike texture image units, image units do not have a separate binding point for each texture target; each image unit may have only one texture bound at a time.

A texture may be bound to an image unit for use by image loads and stores with the command

`void` **BindImageTexture**(`uint` *unit*, `uint` *texture*, `int` *level*, `boolean` *layered*, `int` *layer*, `enum` *access*, `enum` *format*);

where *unit* identifies the image unit, *texture* is the name of the texture, and *level* selects a single level of the texture. If *texture* is zero, any texture currently bound to image unit *unit* is unbound.

If the texture identified by *texture* is a one-dimensional array, two-dimensional array, three-dimensional, cube map, cube map array, or two-dimensional multisample array texture, it is possible to bind either the entire texture level or a single layer or face of the texture level. If *layered* is `TRUE`, the entire level is bound. If *layered* is `FALSE`, only the single layer identified by *layer* will be bound. When *layered* is `FALSE`, the single bound layer is treated as a different texture target for image accesses:

- one-dimensional array texture layers are treated as one-dimensional textures;
- two-dimensional array, three-dimensional, cube map, and cube map array texture layers are treated as two-dimensional textures; and
- two-dimensional multisample array textures are treated as two-dimensional multisample textures.

For cube map textures where *layered* is `FALSE`, the face is taken by mapping the layer number to a face according to table 9.3. For cube map array textures where *layered* is `FALSE`, the selected layer number is mapped to a texture layer and cube face using the following equations and mapping *face* to a face according to table 9.3.

$$layer = \left\lfloor \frac{layer_{orig}}{6} \right\rfloor$$

$$face = layer_{orig} - (layer \times 6)$$

If the texture identified by *texture* does not have multiple layers or faces, the entire texture level is bound, regardless of the values specified for *layered* and *layer*.

format specifies the format that the elements of the image will be treated as when doing formatted stores, as described later in this section. This is referred to as the *image unit format*.

access specifies whether the texture bound to the image will be treated as `READ_ONLY`, `WRITE_ONLY`, or `READ_WRITE`. If a shader reads from an image unit with a texture bound as `WRITE_ONLY`, or writes to an image unit with a texture bound as `READ_ONLY`, the results of that shader operation are undefined and may lead to application termination.

If a texture object bound to one or more image units is deleted by **DeleteTextures**, it is detached from each such image unit, as though **BindImageTexture** were called with *unit* identifying the image unit and *texture* set to zero.

Errors

An INVALID_VALUE error is generated if *unit* is greater than or equal to the value of MAX_IMAGE_UNITS, if *level* or *layer* is negative, or if *texture* is not the name of an existing texture object.

An INVALID_VALUE error is generated if *format* is not one of the formats listed in table 8.34.

The command

```
void BindImageTextures( uint first, sizei count, const
    uint *textures );
```

binds *count* existing texture objects to image units numbered *first* through $first + count - 1$. If *textures* is not NULL, it specifies an array of *count* values, each of which must be zero or the name of an existing texture object. If *textures* is NULL, each affected image unit from *first* through $first + count - 1$ will be reset to have no bound texture object.

When binding a non-zero texture object to an image unit, the image unit *level*, *layered*, *layer*, and *access* parameters are set to zero, TRUE, zero, and READ_WRITE, respectively. The image unit *format* parameter is taken from the internal format of the texture image at level zero of the texture object identified by *textures*. For cube map textures, the internal format of the TEXTURE_CUBE_MAP_POSITIVE_X image of level zero is used. For multisample, multisample array, buffer, and rectangle textures, the internal format of the single texture level is used.

When unbinding a texture object from an image unit, the image unit parameters *level*, *layered*, *layer*, and *format* will be reset to their default values of zero, FALSE, 0, and R8, respectively.

BindImageTextures is equivalent (assuming no errors are generated) to:

```
for (i = 0; i < count; i++) {
    if (textures == NULL || textures[i] = 0) {
        BindImageTexture(first + i, 0, 0, FALSE, 0,
            READ_ONLY, R8);
    } else {
        BindImageTexture(first + i, textures[i], 0, TRUE, 0,
            READ_WRITE, lookupInternalFormat(textures[i]));
    }
}
```

where lookupInternalFormat returns the internal format of the specified texture object.

The values specified in *textures* will be checked separately for each image unit. When a value for a specific image unit is invalid, the state for that image unit will be unchanged and an error will be generated. However, state for other image units will still be changed if their corresponding values are valid.

Errors

An `INVALID_OPERATION` error is generated if $first + count$ is greater than the number of image units supported by the implementation.

An `INVALID_VALUE` error is generated if *count* is negative.

An `INVALID_OPERATION` error is generated if any value in *textures* is not zero or the name of an existing texture object (per binding).

An `INVALID_OPERATION` error is generated if the internal format of the level zero texture image of any texture in *textures* is not found in table 8.34 (per binding).

An `INVALID_OPERATION` error is generated if the width, height, or depth of the level zero texture image of any texture in *textures* is zero (per binding).

When a shader accesses the texture bound to an image unit using a built-in image load, store, or atomic function, it identifies a single texel by providing a one-, two-, or three-dimensional coordinate. Multisample texture accesses also specify a sample number. A coordinate vector is mapped to an individual texel τ_i, τ_{ij}, or τ_{ijk} according to the target of the texture bound to the image unit using table 8.33. As noted above, single-layer bindings of array or cube map textures are considered to use a texture target corresponding to the bound layer, rather than that of the full texture.

If the texture target has layers or cube map faces, the layer or face number is taken from the *layer* argument of **BindImageTexture** if the texture is bound with *layered* set to `FALSE`, or from the coordinate identified by table 8.33 otherwise. For cube map and cube map array textures with *layered* set to `TRUE`, the coordinate is mapped to a layer and face in the same manner as described for the *layer* argument of **BindImageTexture**.

If the individual texel identified for an image load, store, or atomic operation doesn't exist, the access is treated as invalid. Invalid image loads will return zero. Invalid image stores will have no effect. Invalid image atomics will not update any texture bound to the image unit and will return zero. An access is considered invalid if:

- no texture is bound to the selected image unit;
- the texture bound to the selected image unit is incomplete;

Texture target	i	j	k	Face / layer
TEXTURE_1D	x	-	-	-
TEXTURE_2D	x	y	-	-
TEXTURE_3D	x	y	z	-
TEXTURE_RECTANGLE	x	y	-	-
TEXTURE_CUBE_MAP	x	y	-	z
TEXTURE_BUFFER	x	-	-	-
TEXTURE_1D_ARRAY	x	-	-	y
TEXTURE_2D_ARRAY	x	y	-	z
TEXTURE_CUBE_MAP_ARRAY	x	y	-	z
TEXTURE_2D_MULTISAMPLE	x	y	-	-
TEXTURE_2D_MULTISAMPLE_ARRAY	x	y	-	z

Table 8.33: Mapping of image load, store, and atomic texel coordinate components to texel numbers.

- the texture level bound to the image unit is less than the base level or greater than the maximum level of the texture;
- the texture bound to the image unit is bordered;
- the internal format of the texture bound to the image unit is not found in table 8.34;
- the internal format of the texture bound to the image unit is incompatible with the specified *format*, as described below;
- the texture bound to the image unit has layers, and the selected layer or cube map face doesn't exist;
- the selected texel τ_i, τ_{ij}, or τ_{ijk} doesn't exist;
- the image has more samples than the implementation-dependent value of MAX_IMAGE_SAMPLES.

Additionally, there are a number of cases where image load, store, or atomic operations are considered to involve a format mismatch. In such cases, undefined values will be returned by image loads and atomic operations and undefined values will be written by stores and atomic operations. A format mismatch will occur if:

- the type of image variable used to access the image unit does not match the target of a texture bound to the image unit with *layered* set to TRUE;

- the type of image variable used to access the image unit does not match the target corresponding to a single layer of a multi-layer texture target bound to the image unit with *layered* set to `FALSE`;
- the type of image variable used to access the image unit has a component data type (floating-point, signed integer, unsigned integer) incompatible with the format of the image unit;
- the format `layout` qualifier for an image variable used for an image load or atomic operation does not match the format of the image unit, according to table 8.34; or
- the image variable used for an image store has a format `layout` qualifier, and that qualifier does not match the format of the image unit, according to table 8.34.

For textures with multiple samples per texel, the sample selected for an image load, store, or atomic is undefined if the *sample* coordinate is negative or greater than or equal to the number of samples in the texture.

If a shader performs an image load, store, or atomic operation using an image variable declared as an array, and if the index used to select an individual element is negative or greater than or equal to the size of the array, the results of the operation are undefined but may not lead to termination.

Accesses to textures bound to image units do format conversions based on the *format* argument specified when the image is bound. Loads always return a value as a `vec4`, `ivec4`, or `uvec4`, and stores always take the source data as a `vec4`, `ivec4`, or `uvec4`. Data are converted to/from the specified format according to the process described for a **TexImage2D** or **GetTexImage** command with *format* and *type* as `RGBA` and `FLOAT` for `vec4` data, as `RGBA_INTEGER` and `INT` for `ivec4` data, or as `RGBA_INTEGER` and `UNSIGNED_INT` for `uvec4` data, respectively. Unused components are filled in with $(0, 0, 0, 1)$ (where 0 and 1 are either floating-point or integer values, depending on the format).

Any image variable used for shader loads or atomic memory operations must be declared with a format `layout` qualifier matching the format of its associated image unit, as enumerated in table 8.34. Otherwise, the access is considered to involve a format mismatch, as described above. Image variables used exclusively for image stores need not include a format `layout` qualifier, but any declared qualifier must match the image unit format to avoid a format mismatch.

Image Unit Format	Format Qualifer
RGBA32F	rgba32f
RGBA16F	rgba16f
RG32F	rg32f
RG16F	rg16f
R11F_G11F_B10F	r11f_g11f_b10f
R32F	r32f
R16F	r16f
RGBA32UI	rgba32ui
RGBA16UI	rgba16ui
RGB10_A2UI	rgb10_a2ui
RGBA8UI	rgba8ui
RG32UI	rg32ui
RG16UI	rg16ui
RG8UI	rg8ui
R32UI	r32ui
R16UI	r16ui
R8UI	r8ui
RGBA32I	rgba32i
RGBA16I	rgba16i
RGBA8I	rgba8i
RG32I	rg32i
RG16I	rg16i
RG8I	rg8i
R32I	r32i
R16I	r16i
R8I	r8i
RGBA16	rgba16
RGB10_A2	rgb10_a2
RGBA8	rgba8
RG16	rg16
RG8	rg8
R16	r16
R8	r8
RGBA16_SNORM	rgba16_snorm
RGBA8_SNORM	rgba8_snorm
RG16_SNORM	rg16_snorm

(Continued on next page)

Supported image unit formats (continued)	
Image Unit Format	Format Qualifer
RG8_SNORM	rg8_snorm
R16_SNORM	r16_snorm
R8_SNORM	r8_snorm

Table 8.34: Supported image unit formats, with equivalent format `layout` qualifiers.

When a texture is bound to an image unit, the *format* parameter for the image unit need not exactly match the texture internal format as long as the formats are considered compatible. A pair of formats is considered to match in size if the corresponding entries in the *Size* column of table 8.35 are identical. A pair of formats is considered to match by class if the corresponding entries in the *Class* column of table 8.35 are identical. For textures allocated by the GL, an image unit format is compatible with a texture internal format if they match by size. For textures allocated outside the GL, format compatibility is determined by matching by size or by class, in an implementation-dependent manner. The matching criterion used for a given texture may be determined by calling **GetTexParameter** with *pname* set to `IMAGE_FORMAT_COMPATIBILITY_TYPE`, with return values of `IMAGE_FORMAT_COMPATIBILITY_BY_SIZE` and `IMAGE_FORMAT_COMPATIBILITY_BY_CLASS`, specifying matches by size and class, respectively.

When the format associated with an image unit does not exactly match the internal format of a compatible texture bound to the image unit, image loads, stores, and atomic operations re-interpret the memory holding the components of an accessed texel according to the format of the image unit. The re-interpretation for image loads and the read portion of image atomics is performed as though data were copied from the texel of the bound texture to a similar texel represented in the format of the image unit. Similarly, the re-interpretation for image stores and the write portion of image atomics is performed as though data were copied from a texel represented in the format of the image unit to the texel in the bound texture. In both cases, this copy operation would be performed by:

- reading the texel from the source format to scratch memory according to the process described for **GetTexImage** (see section 8.11), using default pixel storage modes and *format* and *type* parameters corresponding to the source format in table 8.35; and

- writing the texel from scratch memory to the destination format according to the process described for **TexSubImage3D** (see section 8.6), using default pixel storage modes and *format* and *type* parameters corresponding to the destination format in table 8.35.

No pixel transfer operations are performed during this conversion.

Image Format	Size	Class	Pixel *format*	Pixel *type*
RGBA32F	128	4x32	RGBA	FLOAT
RGBA16F	64	4x16	RGBA	HALF_FLOAT
RG32F	64	2x32	RG	FLOAT
RG16F	32	2x16	RG	HALF_FLOAT
R11F_G11F_B10F	32	(a)	RGB	UNSIGNED_INT_10F_11F_11F_REV
R32F	32	1x32	RED	FLOAT
R16F	16	1x16	RED	HALF_FLOAT
RGBA32UI	128	4x32	RGBA_INTEGER	UNSIGNED_INT
RGBA16UI	64	4x16	RGBA_INTEGER	UNSIGNED_SHORT
RGB10_A2UI	32	(b)	RGBA_INTEGER	UNSIGNED_INT_2_10_10_10_REV
RGBA8UI	32	4x8	RGBA_INTEGER	UNSIGNED_BYTE
RG32UI	64	2x32	RG_INTEGER	UNSIGNED_INT
RG16UI	32	2x16	RG_INTEGER	UNSIGNED_SHORT
RG8UI	16	2x8	RG_INTEGER	UNSIGNED_BYTE
R32UI	32	1x32	RED_INTEGER	UNSIGNED_INT
R16UI	16	1x16	RED_INTEGER	UNSIGNED_SHORT
R8UI	8	1x8	RED_INTEGER	UNSIGNED_BYTE
RGBA32I	128	4x32	RGBA_INTEGER	INT
RGBA16I	64	4x16	RGBA_INTEGER	SHORT
RGBA8I	32	4x8	RGBA_INTEGER	BYTE
RG32I	64	2x32	RG_INTEGER	INT
RG16I	32	2x16	RG_INTEGER	SHORT
RG8I	16	2x8	RG_INTEGER	BYTE
R32I	32	1x32	RED_INTEGER	INT
R16I	16	1x16	RED_INTEGER	SHORT
R8I	8	1x8	RED_INTEGER	BYTE
RGBA16	64	4x16	RGBA	UNSIGNED_SHORT
RGB10_A2	32	(b)	RGBA	UNSIGNED_INT_2_10_10_10_REV
RGBA8	32	4x8	RGBA	UNSIGNED_BYTE

(Continued on next page)

Texel sizes, compatibility classes ... (continued)				
Image Format	Size	Class	Pixel *format*	Pixel *type*
RG16	32	2x16	RG	UNSIGNED_SHORT
RG8	16	2x8	RG	UNSIGNED_BYTE
R16	16	1x16	RED	UNSIGNED_SHORT
R8	8	1x8	RED	UNSIGNED_BYTE
RGBA16_SNORM	64	4x16	RGBA	SHORT
RGBA8_SNORM	32	4x8	RGBA	BYTE
RG16_SNORM	32	2x16	RG	SHORT
RG8_SNORM	16	2x8	RG	BYTE
R16_SNORM	16	1x16	RED	SHORT
R8_SNORM	8	1x8	RED	BYTE

Table 8.35: Texel sizes, compatibility classes, and pixel format/type combinations for each image format. Class (a) is for 11/11/10 packed floating-point formats; class (b) is for 10/10/10/2 packed formats.

Implementations may support a limited combined number of image units, shader storage blocks (see section 7.8), and active fragment shader outputs (see section 17.4.1). A link error is generated if the sum of the number of active image uniforms used in all shaders, the number of active shader storage blocks, and the number of active fragment shader outputs exceeds the implementation-dependent value of MAX_COMBINED_SHADER_OUTPUT_RESOURCES.

8.26.1 Image Unit Queries

The state required for each image unit is summarized in table 23.56 and may be queried using the indexed query commands in that table. The initial values of image unit state are described above for **BindImageTexture**.

Chapter 9

Framebuffers and Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values from) a framebuffer.

Initially, the GL uses the window system-provided *default framebuffer*. The storage, dimensions, allocation, and format of the images attached to this framebuffer are managed entirely by the window system. Consequently, the state of the default framebuffer, including its images, can not be changed by the GL, nor can the default framebuffer be deleted by the GL.

This chapter begins with an overview of the structure and contents of the framebuffer in section 9.1, followed by describing the commands used to create, destroy, and modify the state and attachments of application-created *framebuffer objects* which may be used instead of the default framebuffer.

9.1 Framebuffer Overview

The framebuffer consists of a set of pixels arranged as a two-dimensional array. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may vary depending on the GL implementation, the type of framebuffer selected, and parameters specified when the framebuffer was created. Creation and management of the default framebuffer is outside the scope of this specification, while creation and management of framebuffer objects is described in detail in section 9.2.

Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several *logical buffers*. These are the *color*, *accumulation*, *depth*, and

stencil buffers. The color buffer actually consists of a number of buffers, and these color buffers serve related but slightly different purposes depending on whether the GL is bound to the default framebuffer or a framebuffer object.

For the default framebuffer, the color buffers are the *front left* buffer, the *front right* buffer, the *back left* buffer, the *back right* buffer, and some number of *auxiliary* buffers. Typically the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) The contents of the auxiliary buffers are never visible. All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, back buffers, or auxiliary buffers at all. Further, an implementation or context may choose not to provide accumulation, depth or stencil buffers. If no default framebuffer is associated with the GL context, the framebuffer is incomplete except when a framebuffer object is bound (see sections 9.2 and 9.4).

Framebuffer objects are not visible, and do not have any of the color buffers present in the default framebuffer. Instead, the buffers of an framebuffer object are specified by attaching individual textures or renderbuffers (see section 9) to a set of attachment points. A framebuffer object has an array of color buffer attachment points, numbered zero through n, a depth buffer attachment point, and a stencil buffer attachment point. In order to be used for rendering, a framebuffer object must be *complete*, as described in section 9.4. Not all attachments of a framebuffer object need to be populated.

Each pixel in a color buffer consists of either a single unsigned integer color index or up to four color components. The four color components are named R, G, B, and A, in that order; color buffers are not required to have all four color components. R, G, B, and A components may be represented as signed or unsigned normalized fixed-point, floating-point, or signed or unsigned integer values; all components must have the same representation. Each pixel in a depth buffer consists of a single unsigned integer value in the format described in section 13.8.1 or a floating-point value. Each pixel in a stencil buffer consists of a single unsigned integer value. Each pixel in an accumulation buffer consists of up to four color components. If an accumulation buffer is present, it must have at least as many bitplanes per component as in the color buffers.

The number of bitplanes in the accumulation, color, depth, and stencil buffers is dependent on the currently bound framebuffer. For the default framebuffer, the number of bitplanes is fixed. For framebuffer objects, the number of bitplanes in a given logical buffer may change if the image attached to the corresponding attachment point changes.

The GL has two active framebuffers; the *draw framebuffer* is the destination

for rendering operations, and the *read framebuffer* is the source for readback operations. The same framebuffer may be used for both drawing and reading. Section 9.2 describes the mechanism for controlling framebuffer usage.

The default framebuffer is initially used as the draw and read framebuffer [1], and the initial state of all provided bitplanes is undefined. The format and encoding of buffers in the draw and read framebuffers may be queried as described in section 9.2.3.

9.2 Binding and Managing Framebuffer Objects

Framebuffer objects encapsulate the state of a framebuffer in a similar manner to the way texture objects encapsulate the state of a texture. In particular, a framebuffer object encapsulates state necessary to describe a collection of color, depth, and stencil logical buffers (other types of buffers are not allowed). For each logical buffer, a framebuffer-attachable image can be attached to the framebuffer to store the rendered output for that logical buffer. Examples of framebuffer-attachable images include texture images and renderbuffer images. Renderbuffers are described further in section 9.2.4

By allowing the images of a renderbuffer to be attached to a framebuffer, the GL provides a mechanism to support *off-screen* rendering. Further, by allowing the images of a texture to be attached to a framebuffer, the GL provides a mechanism to support *render to texture*.

The default framebuffer for rendering and readback operations is provided by the window system. In addition, named framebuffer objects can be created and operated upon. The name space for framebuffer objects is the unsigned integers, with zero reserved by the GL for the default framebuffer.

A framebuffer object is created by binding a name returned by **GenFramebuffers** (see below) to `DRAW_FRAMEBUFFER` or `READ_FRAMEBUFFER`. The binding is effected by calling

`void` **BindFramebuffer**(`enum` *target*, `uint` *framebuffer*);

with *target* set to the desired framebuffer target and *framebuffer* set to the framebuffer object name. The resulting framebuffer object is a new state vector, comprising all the state and with the same initial values listed in table 23.30, as well as one set of the state values listed in table 23.31 for each attachment point of the

[1] The window system binding API may allow associating a GL context with two separate "default framebuffers" provided by the window system as the draw and read framebuffers, but if so, both default framebuffers are referred to by the name zero at their respective binding points.

framebuffer, with the same initial values. There are the value of MAX_COLOR_ATTACHMENTS color attachment points, plus one each for the depth and stencil attachment points.

BindFramebuffer may also be used to bind an existing framebuffer object to DRAW_FRAMEBUFFER and/or READ_FRAMEBUFFER. If the bind is successful no change is made to the state of the newly bound framebuffer object, and any previous binding to *target* is broken.

If a framebuffer object is bound to DRAW_FRAMEBUFFER or READ_FRAMEBUFFER, it becomes the target for rendering or readback operations, respectively, until it is deleted or another framebuffer object is bound to the corresponding bind point. Calling **BindFramebuffer** with *target* set to FRAMEBUFFER binds *framebuffer* to both the draw and read targets.

Errors

An INVALID_ENUM error is generated if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_OPERATION error is generated if *framebuffer* is not zero or a name returned from a previous call to **GenFramebuffers**, or if such a name has since been deleted with **DeleteFramebuffers**.

While a framebuffer object is bound, GL operations on the target to which it is bound affect the images attached to the bound framebuffer object, and queries of the target to which it is bound return state from the bound object. Queries of the values specified in tables 23.85 and 23.30 are derived from the framebuffer object bound to DRAW_FRAMEBUFFER, with the exception of those marked as properties of the read framebuffer, which are derived from the framebuffer object bound to READ_FRAMEBUFFER.

Framebuffer objects may also be created with the command

`void` **CreateFramebuffers**(`sizei` *n*, `uint` **framebuffers*);

CreateFramebuffers returns *n* previously unused framebuffer names in *framebuffers*, each representing a new framebuffer object which is a state vector, comprising all the state and with the same initial values listed in table 23.30, as well as one set of the state values listed in table 23.31 for each attachment point of the framebuffer, with the same initial values.

Errors

An INVALID_VALUE error is generated if n is negative.

The initial state of DRAW_FRAMEBUFFER and READ_FRAMEBUFFER refers to the default framebuffer. In order that access to the default framebuffer is not lost, it is treated as a framebuffer object with the name of zero. The default framebuffer is therefore rendered to and read from while zero is bound to the corresponding targets. On some implementations, the properties of the default framebuffer can change over time (e.g., in response to window system events such as attaching the context to a new window system drawable.)

Framebuffer objects (those with a non-zero name) differ from the default framebuffer in a few important ways. First and foremost, unlike the default framebuffer, framebuffer objects have modifiable attachment points for each logical buffer in the framebuffer. Framebuffer-attachable images can be attached to and detached from these attachment points, which are described further in section 9.2.2. Also, the size and format of the images attached to framebuffer objects are controlled entirely within the GL interface, and are not affected by window system events, such as pixel format selection, window resizes, and display mode changes.

Additionally, when rendering to or reading from an application created-framebuffer object,

- The pixel ownership test always succeeds. In other words, framebuffer objects own all of their pixels.
- There are no visible color buffer bitplanes. This means there is no color buffer corresponding to the back, front, left, or right color bitplanes.
- The only color buffer bitplanes are the ones defined by the framebuffer attachment points named COLOR_ATTACHMENT0 through COLOR_ATTACHMENTn. Each COLOR_ATTACHMENTi adheres to COLOR_ATTACHMENTi = COLOR_ATTACHMENT0 + i[2].
- The only depth buffer bitplanes are the ones defined by the framebuffer attachment point DEPTH_ATTACHMENT.
- The only stencil buffer bitplanes are the ones defined by the framebuffer attachment point STENCIL_ATTACHMENT.
- There are no accumulation buffer bitplanes, so the value of the implementation-dependent state variables ACCUM_RED_BITS, ACCUM_GREEN_BITS, ACCUM_BLUE_BITS, and ACCUM_ALPHA_BITS are all zero.

[2]The header files define tokens COLOR_ATTACHMENTi for i in the range $[0, 31]$. Most implementations support fewer than 32 color attachments, and it is an INVALID_OPERATION error to pass an unsupported attachment name to a command accepting color attachment names.

- There are no auxiliary buffer bitplanes, so the value of the implementation-dependent state variable AUX_BUFFERS is zero.

- If the attachment sizes are not all identical, the results of rendering are defined only within the largest area that can fit in all of the attachments. This area is defined as the intersection of rectangles having a lower left of $(0, 0)$ and an upper right of $(width, height)$ for each attachment. Contents of attachments outside this area are undefined after execution of a rendering command (as defined in section 2.4).

  If there are no attachments, rendering will be limited to a rectangle having a lower left of $(0, 0)$ and an upper right of $(width, height)$, where $width$ and $height$ are the framebuffer object's default width and height.

- If the number of layers of each attachment are not all identical, rendering will be limited to the smallest number of layers of any attachment. If there are no attachments, the number of layers will be taken from the framebuffer object's default layer count.

The command

`void` **GenFramebuffers**(`sizei` *n*, `uint` **framebuffers*);

returns *n* previously unused framebuffer object names in *framebuffers*. These names are marked as used, for the purposes of **GenFramebuffers** only, but they acquire state and type only when they are first bound.

Errors

An INVALID_VALUE error is generated if *n* is negative.

Framebuffer objects are deleted by calling

`void` **DeleteFramebuffers**(`sizei` *n*, `const uint` **framebuffers*);

framebuffers contains *n* names of framebuffer objects to be deleted. After a framebuffer object is deleted, it has no attachments, and its name is again unused. If a framebuffer that is currently bound to one or more of the targets DRAW_FRAMEBUFFER or READ_FRAMEBUFFER is deleted, it is as though **BindFramebuffer** had been executed with the corresponding *target* and *framebuffer* zero. Unused names in *framebuffers* that have been marked as used for the purposes of **GenFramebuffers** are marked as unused again. Unused names in *framebuffers* are silently ignored, as is the value zero.

Errors

An INVALID_VALUE error is generated if *n* is negative.

The command

```
boolean IsFramebuffer( uint framebuffer );
```

returns TRUE if *framebuffer* is the name of an framebuffer object. If *framebuffer* is zero, or if *framebuffer* is a non-zero value that is not the name of a framebuffer object, **IsFramebuffer** returns FALSE.

The names bound to the draw and read framebuffer bindings may be queried by calling **GetIntegerv** with *pname*s DRAW_FRAMEBUFFER_BINDING and READ_FRAMEBUFFER_BINDING, respectively. FRAMEBUFFER_BINDING is equivalent to DRAW_FRAMEBUFFER_BINDING.

9.2.1 Framebuffer Object Parameters

Parameters of a framebuffer object are set using the commands

```
void FramebufferParameteri( enum target, enum pname,
    int param );
void NamedFramebufferParameteri( uint framebuffer,
    enum pname, int param );
```

For **FramebufferParameteri**, the framebuffer object is that bound to *target*, which must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. For **NamedFramebufferParameteri**, *framebuffer* is the name of the framebuffer object.

pname specifies the parameter of the framebuffer object to set.

When a framebuffer has one or more attachments, the width, height, layer count (see section 9.8), sample count, and sample location pattern of the framebuffer are derived from the properties of the framebuffer attachments. When the framebuffer has no attachments, these properties are taken from framebuffer parameters. When *pname* is FRAMEBUFFER_DEFAULT_WIDTH, FRAMEBUFFER_DEFAULT_HEIGHT, FRAMEBUFFER_DEFAULT_LAYERS, FRAMEBUFFER_DEFAULT_SAMPLES, or FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS, *param* specifies the width, height, layer count, sample count, or sample location pattern, respectively, used when the framebuffer has no attachments.

When a framebuffer has no attachments, it is considered layered (see section 9.8) if and only if the value of FRAMEBUFFER_DEFAULT_LAYERS is non-zero.

It is considered to have sample buffers if and only if the value of FRAMEBUFFER_DEFAULT_SAMPLES is non-zero. The number of samples in the framebuffer is derived from the value of FRAMEBUFFER_DEFAULT_SAMPLES in an implementation-dependent manner similar to that described for the command **RenderbufferStorageMultisample** (see section 9.2.4). If the framebuffer has sample buffers and the value of FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS is non-zero, it is considered to have a fixed sample location pattern as described for **TexImage2DMultisample** (see section 8.8).

Errors

An INVALID_ENUM error is generated by **FramebufferParameteri** if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_OPERATION error is generated if the default framebuffer is bound to *target*.

An INVALID_OPERATION error is generated by **NamedFramebufferParameteri** if *framebuffer* is not the name of an existing framebuffer object.

An INVALID_ENUM error is generated if *pname* is not FRAMEBUFFER_DEFAULT_WIDTH, FRAMEBUFFER_DEFAULT_HEIGHT, FRAMEBUFFER_DEFAULT_LAYERS, FRAMEBUFFER_DEFAULT_SAMPLES, or FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS.

An INVALID_VALUE error is generated if *pname* is FRAMEBUFFER_DEFAULT_WIDTH, FRAMEBUFFER_DEFAULT_HEIGHT, FRAMEBUFFER_DEFAULT_LAYERS, or FRAMEBUFFER_DEFAULT_SAMPLES, and *param* is either negative or greater than the value of the corresponding implementation-dependent limit MAX_FRAMEBUFFER_WIDTH, MAX_FRAMEBUFFER_HEIGHT, MAX_FRAMEBUFFER_LAYERS, or MAX_FRAMEBUFFER_SAMPLES, respectively.

9.2.2 Attaching Images to Framebuffer Objects

Framebuffer-attachable images may be attached to, and detached from, framebuffer objects. In contrast, the image attachments of the default framebuffer may not be changed by the GL.

A single framebuffer-attachable image may be attached to multiple framebuffer objects, potentially avoiding some data copies, and possibly decreasing memory consumption.

For each logical buffer, a framebuffer object stores a set of state which defines the logical buffer's *attachment point*. The attachment point state contains enough information to identify the single image attached to the attachment point, or to

indicate that no image is attached. The per-logical buffer attachment point state is listed in table 23.31.

There are several types of framebuffer-attachable images:

- The image of a renderbuffer object, which is always two-dimensional.
- A single level of a one-dimensional texture, which is treated as a two-dimensional image with a height of one.
- A single level of a two-dimensional, two-dimensional multisample, or rectangle texture.
- A single face of a cube map texture level, which is treated as a two-dimensional image.
- A single layer of a one- or two-dimensional array texture, two-dimensional multisample array texture, or three-dimensional texture, which is treated as a two-dimensional image.
- A single layer-face of a cube map array texture, which is treated as a two-dimensional image.

Additionally, an entire level of a three-dimensional, cube map, cube map array, or one- or two-dimensional array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be *layered* (also see section 9.8).

9.2.3 Framebuffer Object Queries

Parameters of a framebuffer object may be queried with the commands

```
void GetFramebufferParameteriv( enum target, enum pname,
    int *params );
void GetNamedFramebufferParameteriv( uint framebuffer,
    enum pname, int *params );
```

For **GetFramebufferParameteriv**, the framebuffer object is that bound to *target*, which must be must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. For **GetNamedFramebufferParameteriv**, *framebuffer* may be zero, indicating the default draw framebuffer, or the name of the framebuffer object.

pname may be one of FRAMEBUFFER_DEFAULT_WIDTH, FRAMEBUFFER_DEFAULT_HEIGHT, FRAMEBUFFER_DEFAULT_LAYERS, FRAMEBUFFER_DEFAULT_SAMPLES, or FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS, indicating one of the corresponding parameters set with **FramebufferParameteri** (see section 9.2.1). These values may only be queried from a framebuffer object, not from a default framebuffer.

pname may also be one of DOUBLEBUFFER, IMPLEMENTATION_COLOR_READ_FORMAT, IMPLEMENTATION_COLOR_READ_TYPE, SAMPLES, SAMPLE_BUFFERS, or STEREO, indicating the corresponding framebuffer-dependent state from table 23.85. Values of framebuffer-dependent state are identical to those that would be obtained were the framebuffer object bound and queried using the simple state queries in that table. These values may be queried from either a framebuffer object or a default framebuffer. The values of SAMPLES and SAMPLE_BUFFERS are determined as described in section 9.2.3.1.

The value of parameter *pname* for the framebuffer object is returned in *params*.

Errors

An INVALID_ENUM error is generated by **GetNamedFramebufferParameteriv** if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_OPERATION error is generated by **GetNamedFramebufferParameteriv** if *framebuffer* is not zero or the name of an existing framebuffer object.

An INVALID_ENUM error is generated if *pname* is not one of the valid values listed above.

An INVALID_OPERATION error is generated by **GetFramebufferParameteriv** if the default framebuffer is bound to *target* and *pname* is not one of the accepted values from table 23.85, other than SAMPLE_POSITION.

An INVALID_OPERATION error is generated by **GetNamedFramebufferParameteriv** if *framebuffer* is zero, and *pname* is not one of the valid values from table 23.85, other than SAMPLE_POSITION.

Attachments of a framebuffer object or buffers of a default framebuffer may be queried with the commands

```
void GetFramebufferAttachmentParameteriv( enum target,
    enum attachment, enum pname, int *params );
void GetNamedFramebufferAttachmentParameteriv(
    uint framebuffer, enum attachment, enum pname,
    int *params );
```

attachment	Buffer Queried
FRONT	Front Left Color
FRONT_LEFT	Front Left Color
FRONT_RIGHT	Front Right Color
BACK	Back Left Color
BACK_LEFT	Back Left Color
BACK_RIGHT	Back Right Color
AUXi	Auxiliary buffer i
DEPTH	Depth buffer
STENCIL	Stencil buffer

Table 9.1: Valid *attachment* parameters when a default framebuffer is queried with **Get*FramebufferAttachmentParameteriv**, and the buffers they select.

For **GetFramebufferAttachmentParameteriv**, the framebuffer object is that bound to *target*, which must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. For **GetNamedFramebufferAttachmentParameteriv**, *framebuffer* is zero or the name of a framebuffer object. If *framebuffer* is zero, then the default draw framebuffer is queried.

If a default framebuffer is queried, then *attachment* must be one of the values listed in table 9.1, selecting a single color, depth or stencil buffer as shown in that table.

Otherwise, *attachment* must be one of the framebuffer object attachment points listed in table 9.2. If *attachment* is DEPTH_STENCIL_ATTACHMENT, the same object must be bound to both the depth and stencil attachment points of the framebuffer object, and information about that object is returned.

Upon successful return from **Get*FramebufferAttachmentParameteriv**, if *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, then *params* will contain one of NONE, FRAMEBUFFER_DEFAULT, TEXTURE, or RENDERBUFFER, identifying the type of object which contains the attached image. Other values accepted for *pname* depend on the type of object, as described below.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is NONE, then one of the following conditions is true:

- no framebuffer is bound to *target*, or
- the default framebuffer is bound, and

  - *attachment* is DEPTH or STENCIL, and the number of depth or stencil bits, respectively, is zero; or,
  - *attachment* does not indicate one of the color buffers allocated to the default framebuffer.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is NONE, querying *pname* FRAMEBUFFER_ATTACHMENT_OBJECT_NAME will return zero, and the only other valid query is FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is not NONE, these queries apply to all other framebuffer types:

- If *pname* is FRAMEBUFFER_ATTACHMENT_RED_SIZE, FRAMEBUFFER_ATTACHMENT_GREEN_SIZE, FRAMEBUFFER_ATTACHMENT_BLUE_SIZE, FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE, FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE, or FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE, then *params* will contain the number of bits in the corresponding red, green, blue, alpha, depth, or stencil component of the specified *attachment*. If the requested component is not present in *attachment*, or if no data storage or texture image has been specified for the attachment, then *params* will contain zero.

- If *pname* is FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, then *params* will contain the format of components of the specified attachment, one of FLOAT, INDEX, INT, UNSIGNED_INT, SIGNED_NORMALIZED, or UNSIGNED_NORMALIZED for floating-point, index, signed integer, unsigned integer, signed normalized fixed-point, or unsigned normalized fixed-point components respectively. Only color buffers may have index components. If no data storage or texture image has been specified for the attachment, then *params* will contain NONE. This query cannot be performed for a combined depth+stencil attachment, since it does not have a single format.

- If *pname* is FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, then *params* will contain the encoding of components of the specified attachment, one of LINEAR or SRGB for linear or sRGB-encoded components, respectively. Only color buffer components may be sRGB-encoded; such components are treated as described in sections 17.3.6 and 17.3.7. For the default framebuffer, color encoding is determined by the implementation. For framebuffer objects, components are sRGB-encoded if the internal format of a color attachment is one of the color-renderable SRGB formats described in section 8.24. If *attachment* is not a color attachment, or no data storage or

texture image has been specified for the attachment, then *params* will contain the value LINEAR.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is RENDERBUFFER, then

- If *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, *params* will contain the name of the renderbuffer object which contains the attached image.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE, then

- If *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, then *params* will contain the name of the texture object which contains the attached image.
- If *pname* is FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, then *params* will contain the mipmap level of the texture object which contains the attached image.
- If *pname* is FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is the name of a cube map texture object, then *params* will contain the cube map face of the cubemap texture object which contains the attached image. Otherwise *params* will contain zero.
- If *pname* is FRAMEBUFFER_ATTACHMENT_LAYERED, then *params* will contain TRUE if an entire level of a three-dimensional texture, cube map texture, or one- or two-dimensional array texture is attached. Otherwise, *params* will contain FALSE.
- If *pname* is FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER; the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is the name of a three-dimensional texture, or a one- or two-dimensional array texture object; and the value of FRAMEBUFFER_ATTACHMENT_LAYERED is FALSE, then *params* will contain the texture layer which contains the attached image. Otherwise *params* will contain zero.

Errors

An INVALID_ENUM error is generated by **GetFramebufferAttachment-Parameteriv** if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_OPERATION error is generated by **GetNamedFrame-**

bufferAttachmentParameteriv if *framebuffer* is not zero or the name of an existing framebuffer object.

An INVALID_ENUM error is generated by any combinations of framebuffer type and *pname* not described above.

An INVALID_OPERATION error is generated if the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is NONE and *pname* is not FRAMEBUFFER_ATTACHMENT_OBJECT_NAME or FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE.

An INVALID_ENUM error is generated if the default framebuffer is queried and *attachment* is not one the values specified in table 9.1.

An INVALID_OPERATION error is generated if a framebuffer object is bound to *target* and *attachment* is COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

An INVALID_ENUM error is generated if a framebuffer object is queried, *attachment* is not one of the attachments in table 9.2, and *attachment* is not COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

An INVALID_OPERATION error is generated if *attachment* is DEPTH_STENCIL_ATTACHMENT and different objects are bound to the depth and stencil attachment points of the framebuffer object.

9.2.3.1 Multisample Queries

The values of SAMPLE_BUFFERS and SAMPLES control whether and how multisampling is performed (see section 14.3.1). They are framebuffer-dependent constants derived from the attachments of a framebuffer object or the buffers of a default framebuffer, and may be determined either by calling **GetFramebufferParameteriv** and **GetNamedFramebufferParameteriv** for a specific framebuffer (see section 9.2.3), or by calling **GetIntegerv** with *pname* set to SAMPLE_BUFFERS or SAMPLES.

If a framebuffer object is not framebuffer complete, as defined in section 9.4.2, then the values of SAMPLE_BUFFERS and SAMPLES are undefined.

Otherwise, the value of SAMPLES is equal to the value of RENDERBUFFER_SAMPLES or TEXTURE_SAMPLES (depending on the type of the attached images), which must all have the same value. The value of SAMPLE_BUFFERS is one if SAMPLES is non-zero, and zero otherwise.

9.2.4 Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. The commands described below allocate and delete a renderbuffer's image, and attach a renderbuffer's image to a framebuffer object.

The name space for renderbuffer objects is the unsigned integers, with zero reserved by the GL. A renderbuffer object is created by binding a name returned by **GenRenderbuffers** (see below) to RENDERBUFFER. The binding is effected by calling

```
void BindRenderbuffer( enum target, uint renderbuffer );
```

with *target* set to RENDERBUFFER and *renderbuffer* set to the renderbuffer object name. If *renderbuffer* is not zero, then the resulting renderbuffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising all the state and with the same initial values listed in table 23.33. Any previous binding to *target* is broken.

BindRenderbuffer may also be used to bind an existing renderbuffer object. If the bind is successful, no change is made to the state of the newly bound renderbuffer object, and any previous binding to *target* is broken.

While a renderbuffer object is bound, GL operations on the target to which it is bound affect the bound renderbuffer object, and queries of the target to which a renderbuffer object is bound return state from the bound object.

The name zero is reserved. A renderbuffer object cannot be created with the name zero. If *renderbuffer* is zero, then any previous binding to *target* is broken and the *target* binding is restored to the initial state.

In the initial state, the reserved name zero is bound to RENDERBUFFER. There is no renderbuffer object corresponding to the name zero, so client attempts to modify or query renderbuffer state for the target RENDERBUFFER while zero is bound will generate GL errors, as described in section 9.2.3.

The current RENDERBUFFER binding can be determined by calling **GetIntegerv** with *pname* RENDERBUFFER_BINDING.

Errors

An INVALID_ENUM error is generated if *target* is not RENDERBUFFER.

An INVALID_OPERATION error is generated if *renderbuffer* is not zero or a name returned from a previous call to **GenRenderbuffers**, or if such a name has since been deleted with **DeleteRenderbuffers**.

New renderbuffers may also be created with the command

`void` **CreateRenderbuffers**(`sizei` *n*, `uint` **renderbuffers*);

CreateRenderbuffers returns *n* previously unused renderbuffer names in *renderbuffers*, each representing a new renderbuffer object which is a state vector comprising all the state and with the initial values listed in table 23.33. The state of each renderbuffer object is as if a name returned from **GenRenderbuffers** had been bound to the `RENDERBUFFER` target, except that any existing binding to `RENDERBUFFER` is not affected.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

The command

`void` **GenRenderbuffers**(`sizei` *n*, `uint` **renderbuffers*);

returns *n* previously unused renderbuffer object names in *renderbuffers*. These names are marked as used, for the purposes of **GenRenderbuffers** only, but they acquire renderbuffer state only when they are first bound.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

Renderbuffer objects are deleted by calling

`void` **DeleteRenderbuffers**(`sizei` *n*, `const uint` **renderbuffers*);

where *renderbuffers* contains *n* names of renderbuffer objects to be deleted. After a renderbuffer object is deleted, it has no contents, and its name is again unused. If a renderbuffer that is currently bound to `RENDERBUFFER` is deleted, it is as though **BindRenderbuffer** had been executed with the *target* `RENDERBUFFER` and *name* of zero. Additionally, special care must be taken when deleting a renderbuffer if the image of the renderbuffer is attached to a framebuffer object (see section 9.2.7). Unused names in *renderbuffers* that have been marked as used for the purposes of **GenRenderbuffers** are marked as unused again. Unused names in *renderbuffers* are silently ignored, as is the value zero.

Errors

An INVALID_VALUE error is generated if *n* is negative.

The command

```
boolean IsRenderbuffer( uint renderbuffer );
```

returns TRUE if *renderbuffer* is the name of a renderbuffer object. If *renderbuffer* is zero, or if *renderbuffer* is a non-zero value that is not the name of a renderbuffer object, **IsRenderbuffer** returns FALSE.

The data storage, format, dimensions, and number of samples of a renderbuffer object's image are established with the commands

```
void RenderbufferStorageMultisample( enum target,
    sizei samples, enum internalformat, sizei width,
    sizei height );
void NamedRenderbufferStorageMultisample(
    uint renderbuffer, sizei samples, enum internalformat,
    sizei width, sizei height );
```

For **RenderbufferStorageMultisample**, the renderbuffer object is that bound to *target*, which must be RENDERBUFFER. For **NamedRenderbufferStorageMultisample**, *renderbuffer* is the name of the renderbuffer object.

internalformat must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 9.4). *width* and *height* are the dimensions in pixels of the renderbuffer.

Upon success, ***RenderbufferStorageMultisample** deletes any existing data store for the renderbuffer image, and the contents of the data store are undefined. RENDERBUFFER_WIDTH is set to *width*, RENDERBUFFER_HEIGHT is set to *height*, and RENDERBUFFER_INTERNAL_FORMAT is set to *internalformat*.

If *samples* is zero, then RENDERBUFFER_SAMPLES is set to zero. Otherwise *samples* represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled rendering, the actual number of samples allocated for the renderbuffer image is implementation-dependent. However, the resulting value for RENDERBUFFER_SAMPLES is guaranteed to be greater than or equal to *samples* and no more than the next larger sample count supported by the implementation.

A GL implementation may vary its allocation of internal component resolution based on any ***RenderbufferStorageMultisample** parameter (except *target* and *renderbuffer*), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established.

Errors

An `INVALID_ENUM` error is generated by **RenderbufferStorageMultisample** if *target* is not `RENDERBUFFER`.

An `INVALID_OPERATION` error is generated by **NamedRenderbufferStorageMultisample** if *renderbuffer* is not the name of an existing renderbuffer object.

An `INVALID_VALUE` error is generated if *samples*, *width*, or *height* is negative.

An `INVALID_OPERATION` error is generated if *samples* is greater than the maximum number of samples supported for *internalformat* (see **GetInternalformativ** in section 22.3).

An `INVALID_ENUM` error is generated if *internalformat* is not one of the color-renderable, depth-renderable, or stencil-renderable formats defined in section 9.4.

An `INVALID_VALUE` error is generated if either *width* or *height* is greater than the value of `MAX_RENDERBUFFER_SIZE`.

The commands

```
void RenderbufferStorage( enum target, enum internalformat,
    sizei width, sizei height );
void NamedRenderbufferStorage( uint renderbuffer,
    enum internalformat, sizei width, sizei height );
```

are equivalent to

RenderbufferStorageMultisample(*target*, 0, *internalformat*, *width*, *height*);

and

NamedRenderbufferStorageMultisample(*renderbuffer*, 0, *internalformat*, *width*, *height*);

respectively.

9.2.5 Required Renderbuffer Formats

Implementations are required to support at least one allocation of internal component resolution for each type (unsigned int, float, etc.) for each base internal format.

In addition, implementations are required to support the following sized and compressed internal formats. Requesting one of these sized internal formats for

a renderbuffer will allocate at least the internal component sizes, and exactly the component types shown for that format in the corresponding table:

- Color formats which are checked in the "Req. rend." column of table 8.19.
- Depth, depth+stencil, and stencil formats which are checked in the "Req. format" column of table 8.21.

The required color formats for renderbuffers are a subset of the required formats for textures (see section 8.5.1).

Implementations must support creation of renderbuffers in these required formats with up to the value of `MAX_SAMPLES` multisamples, with the exception that the signed and unsigned integer formats are required only to support creation of renderbuffers with up to the value of `MAX_INTEGER_SAMPLES` multisamples, which must be at least one.

9.2.6 Renderbuffer Object Queries

Parameters of a renderbuffer object may be queried with the commands

```
void GetRenderbufferParameteriv( enum target, enum pname,
    int *params );
void GetNamedRenderbufferParameteriv( uint renderbuffer,
    enum pname, int *params );
```

For **GetRenderbufferParameteriv**, the renderbuffer object is that bound to *target*, which must be `RENDERBUFFER`. For **GetNamedRenderbufferParameteriv**, *renderbuffer* is the name of the renderbuffer object.

The value of renderbuffer parameter *pname* for the renderbuffer object is returned in *params*. *pname* must be one of the symbolic values in table 23.33.

If *pname* is `RENDERBUFFER_WIDTH`, `RENDERBUFFER_HEIGHT`, `RENDERBUFFER_INTERNAL_FORMAT`, or `RENDERBUFFER_SAMPLES`, then *params* will contain the width in pixels, height in pixels, internal format, or number of samples, respectively, of the image of the renderbuffer object.

If *pname* is `RENDERBUFFER_RED_SIZE`, `RENDERBUFFER_GREEN_SIZE`, `RENDERBUFFER_BLUE_SIZE`, `RENDERBUFFER_ALPHA_SIZE`, `RENDERBUFFER_DEPTH_SIZE`, or `RENDERBUFFER_STENCIL_SIZE`, then *params* will contain the actual resolutions (not the resolutions specified when the image was defined) for the red, green, blue, alpha, depth, or stencil components, respectively, of the image of the renderbuffer object.

Errors

An INVALID_ENUM error is generated by **GetRenderbufferParameteriv** if *target* is not RENDERBUFFER.

An INVALID_OPERATION error is generated by **GetRenderbufferParameteriv** if the renderbuffer currently bound to *target* is zero.

An INVALID_OPERATION error is generated by **GetNamedRenderbufferParameteriv** if *renderbuffer* is not the name of an existing renderbuffer object.

An INVALID_ENUM error is generated if *pname* is not one of the renderbuffer state names in table 23.33.

9.2.7 Attaching Renderbuffer Images to a Framebuffer

A renderbuffer object can be attached as one of the logical buffers of a framebuffer object with the commmands

```
void FramebufferRenderbuffer( enum target,
    enum attachment, enum renderbuffertarget,
    uint renderbuffer );
void NamedFramebufferRenderbuffer( uint framebuffer,
    enum attachment, enum renderbuffertarget,
    uint renderbuffer );
```

For **FramebufferRenderbuffer** the framebuffer object is that bound to *target*, which must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER.

For **NamedFramebufferRenderbuffer**, *framebuffer* is the name of the framebuffer object.

attachment must be set to one of the attachment points of the framebuffer listed in table 9.2.

renderbuffertarget must be RENDERBUFFER and *renderbuffer* is zero or the name of a renderbuffer object of type *renderbuffertarget* to be attached to the framebuffer. If *renderbuffer* is zero, then the value of *renderbuffertarget* is ignored.

If *renderbuffer* is not zero and if ***FramebufferRenderbuffer** is successful, then the renderbuffer named *renderbuffer* will be used as the logical buffer identified by *attachment* of the framebuffer object. The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the specified attachment point is set to RENDERBUFFER and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to *renderbuffer*. All other state values of the attachment point specified

by *attachment* are set to their default values listed in table 23.31. No change is made to the state of the renderbuffer object and any previous attachment to the *attachment* logical buffer of the framebuffer object is broken. If the attachment is not successful, then no change is made to the state of either the renderbuffer object or the framebuffer object.

Calling ***FramebufferRenderbuffer** with the *renderbuffer* name zero will detach the image, if any, identified by *attachment*, in the framebuffer object currently bound to *target*. All state values of the attachment point specified by *attachment* in the framebuffer object are set to their default values listed in table 23.31.

Setting *attachment* to the value `DEPTH_STENCIL_ATTACHMENT` is a special case causing both the depth and stencil attachments of the framebuffer object to be set to *renderbuffer*, which should have base internal format `DEPTH_STENCIL`.

If a renderbuffer object is deleted while its image is attached to one or more attachment points in a currently bound framebuffer object, then it is as if **FramebufferRenderbuffer** had been called, with a *renderbuffer* of zero, for each attachment point to which this image was attached in that framebuffer object. In other words, the renderbuffer image is first detached from all attachment points in that framebuffer object. Note that the renderbuffer image is specifically **not** detached from any non-bound framebuffer objects. Detaching the image from any non-bound framebuffer objects is the responsibility of the application.

Name of attachment
`COLOR_ATTACHMENT`i (see caption)
`DEPTH_ATTACHMENT`
`STENCIL_ATTACHMENT`
`DEPTH_STENCIL_ATTACHMENT`

Table 9.2: Framebuffer attachment points. i in `COLOR_ATTACHMENT`i may range from zero to the value of `MAX_COLOR_ATTACHMENTS` minus one.

Errors

An `INVALID_ENUM` error is generated by **FramebufferRenderbuffer** if *target* is not `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`.

An `INVALID_OPERATION` error is generated by **FramebufferRenderbuffer** if zero is bound to *target*.

An `INVALID_OPERATION` error is generated by **NamedFramebufferRenderbuffer** if *framebuffer* is not the name of an existing framebuffer object.

An INVALID_OPERATION error is generated if *attachment* is COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

An INVALID_ENUM error is generated if *attachment* is not one of the attachments in table 9.2, and *attachment* is not COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

An INVALID_ENUM error is generated if *renderbuffertarget* is not RENDERBUFFER.

An INVALID_OPERATION error is generated if *renderbuffer* is not zero or the name of an existing renderbuffer object of type *renderbuffertarget*.

9.2.8 Attaching Texture Images to a Framebuffer

The GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines **CopyTexImage*** and **CopyTexSubImage***. Additionally, the GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified level of a texture object can be attached as one of the logical buffers of a framebuffer object with the commands

```
void FramebufferTexture( enum target, enum attachment,
    uint texture, int level );
void NamedFramebufferTexture( uint framebuffer,
    enum attachment, uint texture, int level );
```

For **FramebufferTexture**, the framebuffer object is that bound to *target*, which must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. For **NamedFramebufferTexture**, *framebuffer* is the name of the framebuffer object.

attachment must be one of the attachment points of the framebuffer listed in table 9.2.

If *texture* is non-zero, the specified mipmap *level* of the texture object named *texture* is attached to the framebuffer attachment point named by *attachment*.

If *texture* is the name of a three-dimensional texture, cube map array texture, cube map texture, one- or two-dimensional array texture, or two-dimensional multisample array texture, the texture level attached to the framebuffer attachment point is an array of images, and the framebuffer attachment is considered layered.

Errors

An INVALID_ENUM error is generated by **FramebufferTexture** if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_OPERATION error is generated by **FramebufferTexture** if zero is bound to *target*.

An INVALID_OPERATION error is generated by **NamedFramebuffer-Texture** if *framebuffer* is not the name of an existing framebuffer object.

An INVALID_OPERATION error is generated if *attachment* is COLOR_-ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_-ATTACHMENTS.

An INVALID_ENUM error is generated if *attachment* is not one of the attachments in table 9.2, and *attachment* is not COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

An INVALID_VALUE error is generated if *texture* is not zero and is not the name of a texture object, or if *level* is not a supported texture level for *texture*.

An INVALID_OPERATION error is generated if *texture* is the name of a buffer texture.

Additionally, a specified image from a texture object can be attached as one of the logical buffers of a framebuffer object with the commands

```
void FramebufferTexture1D( enum target, enum attachment,
    enum textarget, uint texture, int level );
void FramebufferTexture2D( enum target, enum attachment,
    enum textarget, uint texture, int level );
void FramebufferTexture3D( enum target, enum attachment,
    enum textarget, uint texture, int level, int layer );
```

target specifies the target to which the framebuffer object is bound, and must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER.

attachment must be one of the attachment points of the framebuffer listed in table 9.2.

If *texture* is not zero, then *texture* must either name an existing texture object with a target of *textarget*, or *texture* must name an existing cube map texture and *textarget* must be one of the cube map face targets from table 8.27.

level specifies the mipmap level of the texture image to be attached to the framebuffer, and must satisfy the following conditions:

- If *texture* refers to an immutable-format texture, *level* must be greater than or equal to zero and smaller than the value of TEXTURE_VIEW_NUM_LEVELS for *texture*.

- If *textarget* is TEXTURE_RECTANGLE or TEXTURE_2D_MULTISAMPLE, then *level* must be zero.
- If *textarget* is TEXTURE_3D, then *level* must be greater than or equal to zero and less than or equal to log_2 of the value of MAX_3D_TEXTURE_SIZE.
- If *textarget* is one of the cube map face targets from table 8.27, then *level* must be greater than or equal to zero and less than or equal to log_2 of the value of MAX_CUBE_MAP_TEXTURE_SIZE.
- For all other values of *textarget*, *level* must be greater than or equal to zero and no larger than log_2 of the value of MAX_TEXTURE_SIZE.

layer specifies the layer of a two-dimensional image within a three-dimensional texture.

Errors

An INVALID_ENUM error is generated if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_OPERATION error is generated if *attachment* is COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

An INVALID_ENUM error is generated if *attachment* is not one of the attachments in table 9.2, and *attachment* is not COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

An INVALID_OPERATION error is generated if zero is bound to *target*.

An INVALID_VALUE error is generated if *texture* is not zero and *level* is not a supported texture level for *textarget*, as described above.

An INVALID_VALUE error is generated if *texture* is not zero and *layer* is larger than the value of MAX_3D_TEXTURE_SIZE minus one.

An INVALID_OPERATION error is generated for **FramebufferTexture1D** if *texture* is not zero and *textarget* is not TEXTURE_1D.

An INVALID_OPERATION error is generated for **FramebufferTexture2D** if *texture* is not zero and *textarget* is not one of TEXTURE_2D, TEXTURE_2D_MULTISAMPLE, TEXTURE_RECTANGLE, or one of the cube map face targets from table 8.27.

An INVALID_OPERATION error is generated for **FramebufferTexture3D** if *texture* is not zero and *textarget* is not TEXTURE_3D.

An INVALID_OPERATION error is generated if *texture* is not zero, and does not name an existing texture object of type matching *textarget*, as de-

scribed above.

An INVALID_OPERATION error is generated if *texture* is the name of a buffer texture.

A single layer of a three-dimensional or array texture object can be attached as one of the logical buffers of a framebuffer object with the commands

```
void FramebufferTextureLayer( enum target,
    enum attachment, uint texture, int level, int layer );
void NamedFramebufferTextureLayer( uint framebuffer,
    enum attachment, uint texture, int level, int layer );
```

These commands operate identically to the equivalent **FramebufferTexture** and **NamedFramebufferTexture** commands, respectively, except for the additional *layer* argument which selects a layer of the texture object to attach.

layer specifies the layer of a one- or two-dimensional image within *texture*, except for cube map and cube map array textures. For cube map textures, *layer* is translated into a cube map face as described in table 9.3. For cube map array textures, *layer* is translated into an array layer and a cube map face as described for layer-face numbers in section 8.5.3.

level specifies the mipmap level of the texture image to be attached to the framebuffer, and must satisfy the following conditions:

- If *texture* refers to an immutable-format texture, *level* must be greater than or equal to zero and smaller than the value of TEXTURE_VIEW_NUM_LEVELS for *texture*.
- If *texture* is a three-dimensional texture, then *level* must be greater than or equal to zero and less than or equal to log_2 of the value of MAX_3D_TEXTURE_SIZE.
- If *texture* is a two-dimensional array texture, then *level* must be greater than or equal to zero and less than or equal to log_2 of the value of MAX_TEXTURE_SIZE.
- If *texture* is a two-dimensional multisample array texture, then *level* must be zero.

Errors

In addition to the corresponding errors for **FramebufferTexture** and **NamedFramebufferTexture** when called with the same parameters (other

than *layer*):

An INVALID_VALUE error is generated if *texture* is a three-dimensional texture, and *layer* is larger than the value of MAX_3D_TEXTURE_SIZE minus one.

An INVALID_VALUE error is generated if *texture* is an array texture, and *layer* is larger than the value of MAX_ARRAY_TEXTURE_LAYERS minus one.

An INVALID_VALUE error is generated if *texture* is a cube map array texture, and

$$\frac{layer}{6}$$

is larger than the value of MAX_CUBE_MAP_TEXTURE_SIZE minus one (see section 9.8).

An INVALID_VALUE error is generated if *texture* is non-zero and *layer* is negative.

An INVALID_OPERATION error is generated if *texture* is non-zero and is not the name of a three-dimensional, two-dimensional multisample array, one- or two-dimensional array, cube map, or cube map array texture.

An INVALID_VALUE error is generated if *texture* is not zero and *level* is not a supported texture level for *texture*, as described above.

If *texture* is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to *attachment* is updated as in the other **FramebufferTexture*** commands, except that the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is set to *layer*.

9.2.8.1 Effects of Attaching a Texture Image

The remaining comments in this section apply to all forms of ***FramebufferTexture***.

If *texture* is zero, any image or array of images attached to the attachment point named by *attachment* is detached. Any additional parameters (*level*, *textarget*, and/or *layer*) are ignored when *texture* is zero. All state values of the attachment point specified by *attachment* are set to their default values listed in table 23.31.

If *texture* is not zero, and if ***FramebufferTexture*** is successful, then the specified texture image will be used as the logical buffer identified by *attachment* of the framebuffer object currently bound to *target*. State values of the specified attachment point are set as follows:

- The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is set to TEXTURE.

- The value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to *texture*.
- The value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL is set to *level*.
- If ***FramebufferTexture2D** is called and *texture* is a cube map texture, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE is set to *textarget*; otherwise it is set to the default value (NONE).
- If ***FramebufferTextureLayer** or ***FramebufferTexture3D** is called, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is set to *layer*; otherwise it is set to zero.
- If ***FramebufferTexture*** is called and *texture* is the name of a three-dimensional, cube map, two-dimensional multisample array, or one- or two-dimensional array texture, the value of FRAMEBUFFER_ATTACHMENT_LAYERED is set to TRUE; otherwise it is set to FALSE.

All other state values of the attachment point specified by *attachment* are set to their default values listed in table 23.31. No change is made to the state of the texture object, and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.

Setting *attachment* to the value DEPTH_STENCIL_ATTACHMENT is a special case causing both the depth and stencil attachments of the framebuffer object to be set to *texture*. *texture* must have base internal format DEPTH_STENCIL, or the depth and stencil framebuffer attachments will be incomplete (see section 9.4.1).

If a texture object is deleted while its image is attached to one or more attachment points in a currently bound framebuffer object, then it is as if ***FramebufferTexture*** had been called, with a *texture* of zero, for each attachment point to which this image was attached in that framebuffer object. In other words, the texture image is first detached from all attachment points in that framebuffer object. Note that the texture image is specifically **not** detached from any non-bound framebuffer objects. Detaching the texture image from any non-bound framebuffer objects is the responsibility of the application.

9.3 Feedback Loops Between Textures and the Framebuffer

A *feedback loop* may exist when the data store of a texture object is used as both the source and destination of a GL operation. When a feedback loop exists, un-

defined behavior results. This section describes *rendering feedback loops* (see section 8.14.2.1) and *texture copying feedback loops* (see section 8.6.1) in more detail.

9.3.1 Rendering Feedback Loops

The mechanisms for attaching textures to a framebuffer object do not prevent a one- or two-dimensional texture level, a face of a cube map texture level, or a layer of a two-dimensional array, three-dimensional texture, or texture view from being attached to the draw framebuffer while the same texture is bound to a texture unit. While this condition holds, texturing operations accessing that image will produce undefined results, as described at the end of section 8.14. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturing operations are likely to leave the final results of the shader or fixed-function fragment processing operations undefined, and should be avoided.

Special precautions need to be taken to avoid attaching a texture image to the currently bound draw framebuffer object while the texture object is currently bound and enabled for texturing. Doing so could lead to the creation of a rendering feedback loop between the writing of pixels by GL rendering operations and the simultaneous reading of those same pixels when used as texels in the currently bound texture. In this scenario, the framebuffer will be considered framebuffer complete (see section 9.4), but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well, as described under "Rendering Feedback Loops" in section 8.14.2.1

Specifically, the values of rendered fragments are undefined if any fixed-function or shader stage fetches texels and the same texels are written via fixed-function fragment processing or fragment shader outputs, even if the reads and writes are not in the same draw call, unless any of the following exceptions apply:

- The reads and writes are from/to disjoint sets of texels (after accounting for texture filtering rules).
- There is only a single read and write of each texel, and the read is in the fragment shader invocation that writes the same texel (e.g. using `texelFetch2D(sampler, ivec2(gl_FragCoord.xy), 0);`).
- If a texel has been written, then in order to safely read the result a texel fetch must be in a subsequent draw call separated by the command

  `void` **TextureBarrier**`( void );`

TextureBarrier will guarantee that writes have completed and caches have been invalidated before subsequent draw calls are executed.

9.3.2 Texture Copying Feedback Loops

Similarly to rendering feedback loops, it is possible for a texture image to be attached to the currently bound read framebuffer object while the same texture image is the destination of a **CopyTexImage*** operation, as described under "Texture Copying Feedback Loops" in section 8.6.1. While this condition holds, a texture copying feedback loop between the writing of texels by the copying operation and the reading of those same texels when used as pixels in the read framebuffer may exist. In this scenario, the values of texels written by the copying operation will be undefined (in the same fashion that overlapping copies via **BlitFramebuffer** are undefined).

Specifically, the values of copied texels are undefined if all of the following conditions are true:

- an image from texture object *T* is attached to the currently bound read framebuffer object at attachment point *A*
- the selected read buffer (see section 18.2.1) is attachment point *A*
- *T* is bound to the texture target of a **CopyTexImage*** operation
- the *level* argument of the copying operation selects the same image that is attached to *A*

9.4 Framebuffer Completeness

A framebuffer must be *framebuffer complete* to effectively be used as the draw or read framebuffer of the GL.

The default framebuffer is always complete if it exists; however, if no default framebuffer exists (no window system-provided drawable is associated with the GL context), it is deemed to be incomplete.

A framebuffer object is said to be framebuffer complete if all of its attached images, and all framebuffer parameters required to utilize the framebuffer for rendering and reading, are consistently defined and meet the requirements defined below. The rules of framebuffer completeness are dependent on the properties of the attached images, and on certain implementation-dependent restrictions.

The internal formats of the attached images can affect the completeness of the framebuffer, so it is useful to first define the relationship between the internal format of an image and the attachment points to which it can be attached.

- An internal format is *color-renderable* if it is ALPHA, RED, RG, RGB, RGBA, or one of the sized internal formats from table 8.19 whose "CR" (color-renderable) column is checked in that table No other formats, including compressed internal formats, are color-renderable.
- An internal format is *depth-renderable* if it is DEPTH_COMPONENT or one of the formats from table 8.21 whose base internal format is DEPTH_COMPONENT or DEPTH_STENCIL. No other formats are depth-renderable.
- An internal format is *stencil-renderable* if it is STENCIL_INDEX, DEPTH_STENCIL, or one of the formats from table 8.21 whose base internal format is STENCIL_INDEX or DEPTH_STENCIL. No other formats are stencil-renderable.

9.4.1 Framebuffer Attachment Completeness

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the framebuffer attachment point *attachment* is not NONE, then it is said that a framebuffer-attachable image, named *image*, is attached to the framebuffer at the attachment point. *image* is identified by the state in *attachment* as described in section 9.2.2.

The framebuffer attachment point *attachment* is said to be *framebuffer attachment complete* if the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for *attachment* is NONE (i.e., no image is attached), or if all of the following conditions are true:

- *image* is a component of an existing object with the name specified by the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, and of the type specified by the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE.
- The width and height of *image* are greater than zero and less than or equal to the values of the implementation-dependent limits MAX_FRAMEBUFFER_WIDTH and MAX_FRAMEBUFFER_HEIGHT, respectively.
- If *image* is a three-dimensional, one- or two-dimensional array, or cube map array texture and the attachment is not layered, the selected layer is less than the depth (for three-dimensional) or layer count (otherwise) of the texture.

- If *image* is a three-dimensional, one- or two-dimensional array, or cube map array texture and the attachment is layered, the depth or layer count of the texture is less than or equal to the value of the implementation-dependent limit MAX_FRAMEBUFFER_LAYERS.
- If *image* has multiple samples, its sample count is less than or equal to the value of the implementation-dependent limit MAX_FRAMEBUFFER_SAMPLES.
- If *image* is not an immutable-format texture, the selected level number is in the range $[level_{base}, q]$, where $level_{base}$ and q are as defined in section 8.14.3.
- If *image* is not an immutable-format texture and the selected level is not $level_{base}$, the texture must be mipmap complete.
- If the attachment is layered and the selected level is $level_{base}$, then the texture must be base complete.
- If the attachment is layered and the selected level is not $level_{base}$, then the texture must be mipmap complete.
- If *attachment* is COLOR_ATTACHMENTi, then *image* must have a color-renderable internal format.
- If *attachment* is DEPTH_ATTACHMENT, then *image* must have a depth-renderable internal format.
- If *attachment* is STENCIL_ATTACHMENT, then *image* must have a stencil-renderable internal format.

9.4.2 Whole Framebuffer Completeness

Each rule below is followed by an error token enclosed in { brackets }. The meaning of these errors is explained below and under "Effects of Framebuffer Completeness on Framebuffer Operations" in section 9.4.4.

The framebuffer object bound to *target* is said to be *framebuffer complete* if all the following conditions are true:

- if the default framebuffer is bound to *target*, the default framebuffer exists.

  { FRAMEBUFFER_UNDEFINED }

- All framebuffer attachment points are *framebuffer attachment complete*.

  { FRAMEBUFFER_INCOMPLETE_ATTACHMENT }

- There is at least one image attached to the framebuffer, or the value of the framebuffer's FRAMEBUFFER_DEFAULT_WIDTH and FRAMEBUFFER_DEFAULT_HEIGHT parameters are both non-zero.

  { FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT }

- The combination of internal formats of the attached images does not violate an implementation-dependent set of restrictions.

  { FRAMEBUFFER_UNSUPPORTED }

- The value of RENDERBUFFER_SAMPLES is the same for all attached renderbuffers; the value of TEXTURE_SAMPLES is the same for all attached textures; and, if the attached images are a mix of renderbuffers and textures, the value of RENDERBUFFER_SAMPLES matches the value of TEXTURE_SAMPLES.

  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }

- The value of TEXTURE_FIXED_SAMPLE_LOCATIONS is the same for all attached textures; and, if the attached images are a mix of renderbuffers and textures, the value of TEXTURE_FIXED_SAMPLE_LOCATIONS must be TRUE for all attached textures.

  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }

- If any framebuffer attachment is layered, all populated attachments must be layered. Additionally, all populated color attachments must be from textures of the same target (three-dimensional, one- or two-dimensional array, cube map, or cube map array textures).

  { FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS }

The token in brackets after each clause of the framebuffer completeness rules specifies the return value of **CheckFramebufferStatus** (see below) that is generated when that clause is violated. If more than one clause is violated, it is

implementation-dependent which value will be returned by **CheckFramebuffer-Status**.

Performing any of the following actions may change whether the framebuffer is considered complete or incomplete:

- Binding to a different framebuffer with **BindFramebuffer**.
- Attaching an image to the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Detaching an image from the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Changing the internal format of a texture image that is attached to the framebuffer by calling **TexImage***, **TexStorage***, **CopyTexImage***, or **CompressedTexImage***.
- Changing the internal format of a renderbuffer that is attached to the framebuffer by calling **RenderbufferStorage***.
- Deleting, with **DeleteTextures** or **DeleteRenderbuffers**, an object containing an image that is attached to a currently bound framebuffer object.
- Associating a different window system-provided drawable, or no drawable, with the default framebuffer using a window system binding API such as those described in section 1.3.6.

Although the GL defines a wide variety of internal formats for framebuffer-attachable images, such as texture images and renderbuffer images, some implementations may not support rendering to particular combinations of internal formats. If the combination of formats of the images attached to a framebuffer object are not supported by the implementation, then the framebuffer is not complete under the clause labeled `FRAMEBUFFER_UNSUPPORTED`.

Implementations are required to support certain combinations of framebuffer internal formats as described under "Required Framebuffer Formats" in section 9.4.3.

Because of the *implementation-dependent* clause of the framebuffer completeness test in particular, and because framebuffer completeness can change when the set of attached images is modified, it is strongly advised, though not required, that an application check to see if the framebuffer is complete prior to rendering.

The status of a framebuffer object or default framebuffer may be queried with the commands

```
enum CheckFramebufferStatus( enum target );
enum CheckNamedFramebufferStatus( uint framebuffer,
    enum target );
```

For **CheckFramebufferStatus**, the framebuffer object is that bound to *target*. For **CheckNamedFramebufferStatus**, *framebuffer* is the name of the framebuffer object.

target must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER.

If *framebuffer* is zero, then the status of the default read or draw framebuffer (as determined by *target*) is returned.

A value is returned that identifies whether or not the framebuffer object or default framebuffer is complete when treated as a read or draw framebuffer (as determined by *target*). If the framebuffer object is complete, then FRAMEBUFFER_COMPLETE is returned. Otherwise, the value returned is one of the error codes defined at the start of section 9.4.2 identifying one of the rules of framebuffer completeness that is violated.

If **CheckFramebufferStatus** generates an error, zero is returned.

Errors

An INVALID_ENUM error is generated if *target* is not DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER.

An INVALID_OPERATION error is generated by **CheckNamedFramebufferStatus** if *framebuffer* is not the name of an existing framebuffer object.

9.4.3 Required Framebuffer Formats

Implementations must support framebuffer objects with up to the value of MAX_COLOR_ATTACHMENTS color attachments, a depth attachment, and a stencil attachment. Each color attachment may be in any of the color-renderable formats described in section 9.4 (although implementations are not required to support creation of attachments in all color-renderable formats). The depth attachment may be in any of the required depth or combined depth+stencil formats described in sections 8.5.1 and 9.2.5, and the stencil attachment may be in any of the required stencil or combined depth+stencil formats. However, when both depth and stencil attachments are present, implementations are only required to support framebuffer objects where both attachments refer to the same image.

Any internal format that returns FULL_SUPPORT from the **GetInternalformativ** query for *pname* FRAMEBUFFER_RENDERABLE (see section 22.3.1) can be used

in non-layered attachments in any combination with other required formats or formats that offer FULL_SUPPORT.

Any internal format that returns FULL_SUPPORT from the query for *pname* FRAMEBUFFER_RENDERABLE_LAYERED can be used in any combination with other required formats or formats that offer FULL_SUPPORT.

There must be at least one default framebuffer format allowing creation of a default framebuffer supporting front-buffered rendering.

9.4.4 Effects of Framebuffer Completeness on Framebuffer Operations

Errors

An INVALID_FRAMEBUFFER_OPERATION error is generated by attempts to render to or read from a framebuffer which is not framebuffer complete. This error is generated regardless of whether fragments are actually read from or written to the framebuffer. For example, it is generated when a rendering command is called and the framebuffer is incomplete, even if RASTERIZER_DISCARD is enabled.

An INVALID_FRAMEBUFFER_OPERATION error is generated by rendering commands (see section 2.4), **RasterPos**, and commands that read from the framebuffer such as **ReadPixels**, **CopyTexImage***, and **CopyTexSubImage*** if called while the framebuffer is not framebuffer complete.

9.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 23.85 may change when a change is made to the current read or draw framebuffer binding, to the state of a currently bound framebuffer object, or to an image attached to that framebuffer object. Most such state is dependent on the draw framebuffer (the value DRAW_FRAMEBUFFER_BINDING), but IMPLEMENTATION_COLOR_READ_TYPE and IMPLEMENTATION_COLOR_READ_FORMAT are dependent on the read framebuffer (the value of READ_FRAMEBUFFER_BINDING).

The values of the state variables listed in table 23.85 may change when a change is made to DRAW_FRAMEBUFFER_BINDING, to the state of the currently bound draw framebuffer object, or to an image attached to that framebuffer object.

When DRAW_FRAMEBUFFER_BINDING is zero, the values of the state variables listed in table 23.85 are implementation-defined.

When DRAW_FRAMEBUFFER_BINDING is non-zero, if the currently bound draw framebuffer object is not framebuffer complete, then the values of the state

variables listed in table 23.85 are undefined.

When DRAW_FRAMEBUFFER_BINDING is non-zero and the currently bound draw framebuffer object is framebuffer complete, then the values of the state variables listed in table 23.85 are completely determined by DRAW_FRAMEBUFFER_BINDING, the state of the currently bound draw framebuffer object, and the state of the images attached to that framebuffer object. The values of RED_BITS, GREEN_BITS, BLUE_BITS, and ALPHA_BITS are defined only if all color attachments of the draw framebuffer have identical formats, in which case the color component depths of color attachment zero are returned. The values returned for DEPTH_BITS and STENCIL_BITS are the depth or stencil component depth of the corresponding attachment of the draw framebuffer, respectively.

The actual sizes of the color, depth, or stencil bit planes can be obtained by querying an attachment point using **Get*FramebufferAttachmentParameteriv**, or querying the object attached to that point. If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE at a particular attachment point is RENDERBUFFER, the sizes may be determined by calling **GetRenderbufferParameteriv** as described in section 9.2.6. If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE at a particular attachment point is TEXTURE, the sizes may be determined by calling **GetTexParameter**, as described in section 8.11.

9.5 Mapping between Pixel and Element in Attached Image

When DRAW_FRAMEBUFFER_BINDING is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an operation that reads from the framebuffer reads from the image attached to the selected logical buffer.

If the attached image is a renderbuffer image, then the window coordinates (x_w, y_w) correspond to the value in the renderbuffer image at the same coordinates.

If the attached image is a texture image, then the window coordinates (x_w, y_w) correspond to the texel (i, j, k) from figure 8.3 as follows:

$$i = (x_w - b)$$
$$j = (y_w - b)$$
$$k = (layer - b)$$

where b is the texture image's border width and $layer$ is the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER for the selected logical buffer.

For a two-dimensional texture, k and $layer$ are irrelevant; for a one-dimensional texture, j, k, and $layer$ are irrelevant.

(x_w, y_w) corresponds to a border texel if x_w, y_w, or $layer$ is less than the border width, or if x_w, y_w, or $layer$ is greater than or equal to the border width plushe width, height, or depth, respectively, of the texture image.

9.6 Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while the draw framebuffer binding is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components as described in table 8.18, according to the table row corresponding to the internal format of the framebuffer-attachable image attached to the selected logical buffer, and the resulting internal components are written to the image attached to logical buffer. The masking operations described in section 17.4.2 are also effective.

9.7 Conversion to RGBA Values

When a color value is read while the read framebuffer binding is non-zero, or is used as the source of a logical operation or for blending while the draw framebuffer binding is non-zero, components of that color taken from the framebuffer-attachable image attached to the selected logical buffer are first converted to R, G, B, and A values according to table 16.1 and the internal format of the attached image.

9.8 Layered Framebuffers

A framebuffer is considered to be *layered* if it is complete and all of its populated attachments are layered. When rendering to a layered framebuffer, each fragment generated by the GL is assigned a layer number. The layer number for a fragment is zero if

- the fragment is generated by **DrawPixels**, **CopyPixels**, or **Bitmap**,
- geometry shaders are disabled, or
- the current geometry shader does not statically assign a value to the built-in output variable `gl_Layer`.

Layer Number	Cube Map Face
0	TEXTURE_CUBE_MAP_POSITIVE_X
1	TEXTURE_CUBE_MAP_NEGATIVE_X
2	TEXTURE_CUBE_MAP_POSITIVE_Y
3	TEXTURE_CUBE_MAP_NEGATIVE_Y
4	TEXTURE_CUBE_MAP_POSITIVE_Z
5	TEXTURE_CUBE_MAP_NEGATIVE_Z

Table 9.3: Layer numbers for cube map texture faces. The layers are numbered in the same sequence as the cube map face token values.

Otherwise, the layer for each point, line, or triangle emitted by the geometry shader is taken from the `gl_Layer` output of one of the vertices of the primitive. The vertex used is implementation-dependent. To get defined results, all vertices of each primitive emitted should set the same value for `gl_Layer`. Since the `EndPrimitive` built-in function starts a new output primitive, defined results can be achieved if `EndPrimitive` is called between two vertices emitted with different layer numbers. A layer number written by a geometry shader has no effect if the framebuffer is not layered.

When fragments are written to a layered framebuffer, the fragment's layer number selects an image from the array of images at each attachment point to use for the stencil test (see section 17.3.3), depth buffer test (see section 17.3.4), and for blending and color buffer writes (see section 17.3.6). If the fragment's layer number is negative, or greater than or equal to the minimum number of layers of any attachment, the effects of the fragment on the framebuffer contents are undefined.

When the **Clear** or **ClearBuffer*** commands described in section 17.4.3 are used to clear a layered framebuffer attachment, all layers of the attachment are cleared.

When commands such as **ReadPixels** or **CopyPixels** read from a layered framebuffer, the image at layer zero of the selected attachment is always used to obtain pixel values.

When cube map texture levels are attached to a layered framebuffer, there are six layers, numbered zero through five. Each layer number corresponds to a cube map face, as shown in table 9.3.

When cube map array texture levels are attached to a layered framebuffer, the layer number corresponds to a layer-face. The layer-face can be translated into an array layer and a cube map face by

$$array_layer = \left\lfloor \frac{layer}{6} \right\rfloor$$
$$face = layer \bmod 6$$

.

The face number corresponds to the cube map faces as shown in table 9.3.

Chapter 10

Vertex Specification and Drawing Commands

Most geometric primitives are drawn by specifying a series of generic attribute sets corresponding to vertices of a primitive using **DrawArrays** or one of the other drawing commands defined in section 10.4. Points, lines, polygons, and a variety of related geometric primitives (see section 10.1) can be drawn in this way.

Alternatively, non-generic attributes of vertices such as vertex coordinates, normals, texture coordinates, and vertex colors may be specified for use by shaders or by fixed-function vertex processing functionality.

Geometric objects may also be drawn by specifying attributes between **Begin** / **End** pairs as described in section 10.7.

The process of specifying attributes of a vertex and passing them to fixed-function vertex processing or a shader is referred to as *transferring* a vertex to the GL.

Vertex Shader Processing and Vertex State

Each vertex is specified with one or more generic vertex attributes. Each attribute is specified with one, two, three, or four scalar values.

Generic vertex attributes can be accessed from within vertex shaders (see section 11.1) and used to compute values for consumption by later processing stages.

Before vertex shader execution, the state required by a vertex is its generic vertex attributes. Vertex shader execution processes vertices producing a homogeneous vertex position and any outputs explicitly written by the vertex shader.

Figure 10.1 shows the sequence of operations that builds a *primitive* (point, line segment, or polygon) from a sequence of vertices. After a primitive is formed, it is clipped to a clip volume. This may modify the primitive by altering vertex coordi-

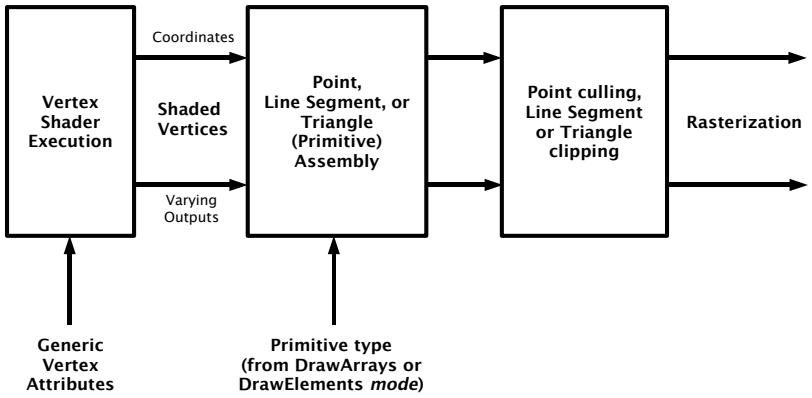


Figure 10.1. Vertex processing and primitive assembly.

nates and vertex shader outputs. In the case of line and polygon primitives, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have output variables associated with them.

Fixed-Function Vertex Processing and Vertex State

When fixed-function vertex processing is performed, each vertex is specified with two, three, or four scalar vertex coordinates. In addition, a *current normal*, multiple *current texture coordinate sets*, multiple *current generic vertex attributes*, *current color*, *current secondary color*, and *current fog coordinate* may be used in processing each vertex.

Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it.

Texture coordinates determine how a texture image is mapped onto a primitive in fixed-function fragment processing. Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture units supported is implementation-dependent but must be at least two. The number of texture units supported may be queried with the state `MAX_TEXTURE_UNITS`.

Primary and secondary colors are associated with each vertex (see section 16.3). These *associated* colors are either based on the current color and current secondary color or produced by lighting, depending on whether or not lighting is enabled. Texture and fog coordinates are similarly associated with each vertex. Multiple sets of texture coordinates may be associated with a vertex. Figure 10.2 summarizes the association of auxiliary data with a transformed vertex to produce a *processed vertex*.

Current values are part of GL state. Vertices and normals are transformed, colors may be affected or replaced by lighting, and texture coordinates are transformed and possibly affected by a texture coordinate generation function. The processing indicated for each current value is applied for each vertex that is sent to the GL. The details of fixed-function vertex processing are discussed later.

Before colors have been assigned to a vertex, the state required by a vertex is the vertex's coordinates, the current normal, the current edge flag (see section 10.1.17), the current material properties (see section 12.2.2), the current fog coordinate, the multiple generic vertex attribute sets, and the multiple current texture coordinate sets. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex's coordinates, its edge flag, its fog coordinate, its assigned colors, and its multiple texture coordinate sets.

Figure 10.3 shows the sequence of operations that builds a *primitive* (point, line segment, or polygon) from a sequence of vertices using using fixed-functin

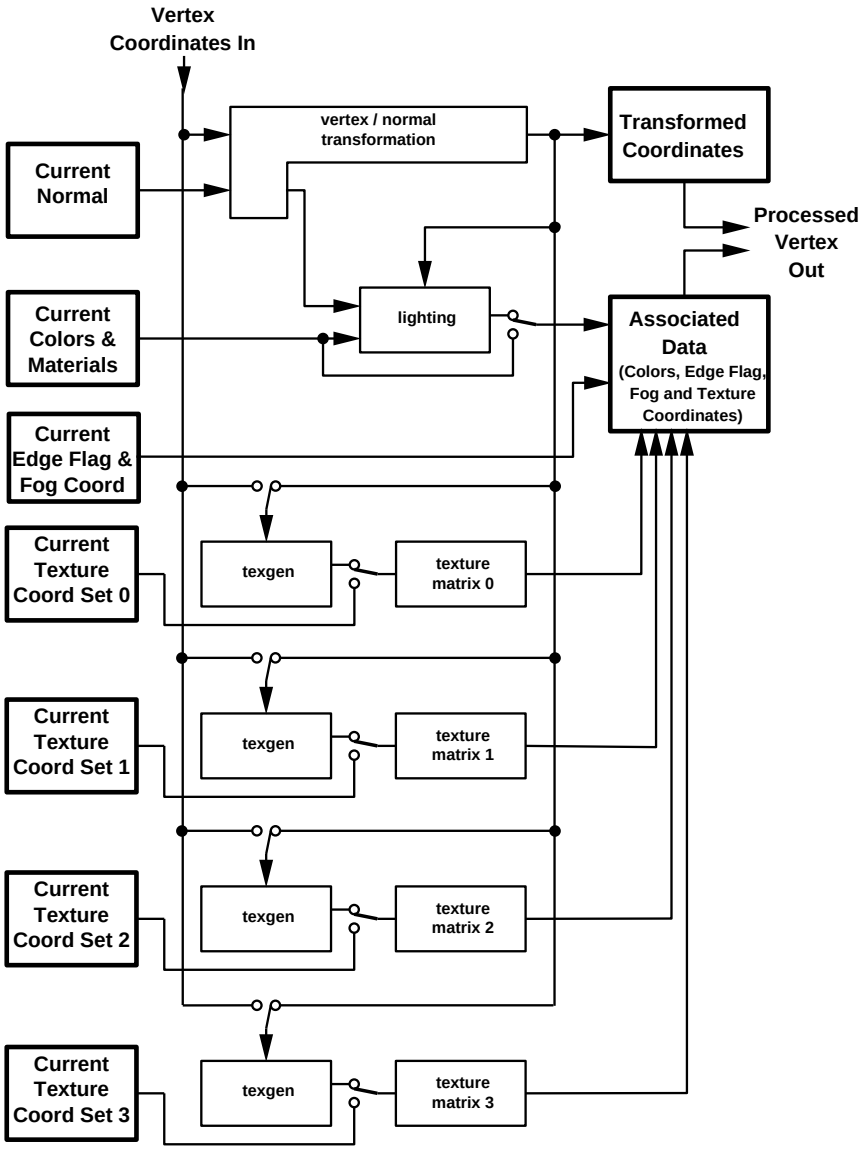


Figure 10.2. Association of current values with a vertex. The heavy lined boxes represent GL state. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

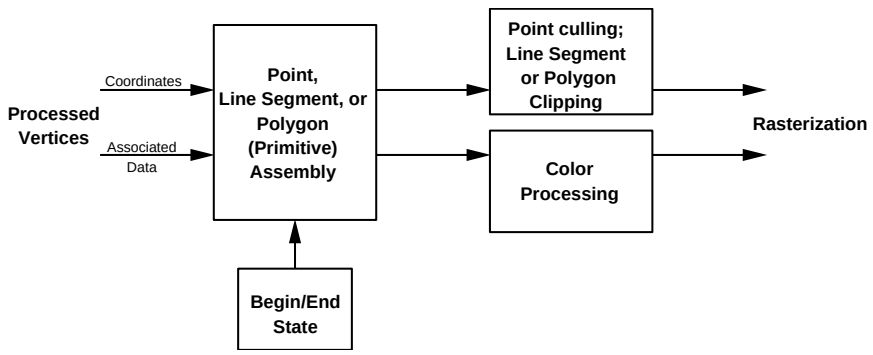


Figure 10.3. Primitive assembly and processing.

vertex processing. The vertices defining a primitive to be rasterized have vertex coordinates, texture coordinates and colors associated with them. After a primitive is formed, clipping may alter all these attributes.

10.1 Primitive Types

A sequence of vertices is passed to the GL using **DrawArrays** or one of the other drawing commands defined in section 10.4. There is no limit to the number of vertices that may be specified, other than the size of the vertex arrays. The *mode* parameter of these commands determines the type of primitives to be drawn using the vertices. Primitive types and the corresponding *mode* parameters are summarized below, together with any additional state required when assembling primitives from multiple vertices.

10.1.1 Points

A series of individual points are specified with *mode* `POINTS`. Each vertex defines a separate point. No state is required for points, since each point is independent of any previous and following points.

10.1.2 Line Strips

A series of one or more connected line segments are specified with *mode* LINE_STRIP. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the ith vertex (for $i > 1$) specifies the beginning of the ith segment and the end of the $i - 1$st. The last vertex specifies the end of the last segment. If only one vertex is specified, then no primitive is generated.

The required state consists of the processed vertex produced from the last vertex that was sent (so that a line segment can be generated from it to the current vertex), and a boolean flag indicating if the current vertex is the first vertex.

10.1.3 Line Loops

A line loop is specified with *mode* LINE_LOOP. Loops are the same as line strips except that a final segment is added from the final specified vertex to the first vertex. The required state consists of the processed first vertex, in addition to the state required for line strips.

10.1.4 Separate Lines

Individual line segments, each defined by a pair of vertices, are specified with *mode* LINES. The first two vertices passed define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of vertices passed is odd, then the last vertex is ignored. The state required is the same as for line strips but it is used differently: a processed vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

10.1.5 Polygons

A polygon is described by specifying its boundary as a series of line segments with *mode* POLYGON. The bounding line segments are specified in the same way as line loops. A polygon described with fewer than three vertices does not generate a primitive.

The state required to support polygons consists of at least two processed vertices (more than two are never required, although an implementation may use more); this is because a convex polygon can be rasterized as its vertices arrive, before all of them have been specified.

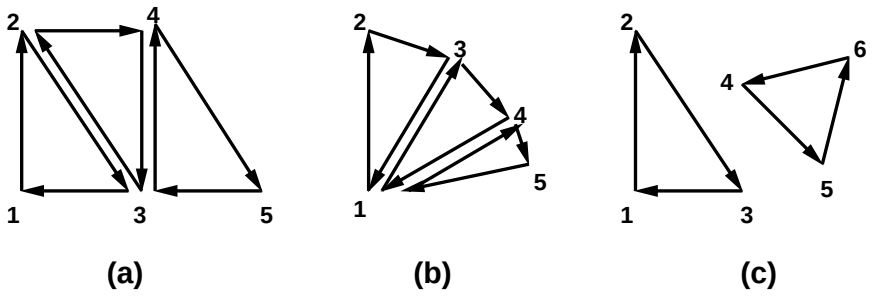


Figure 10.4. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

10.1.6 Triangle Strips

A triangle strip is a series of triangles connected along shared edges, and is specified with *mode* `TRIANGLE_STRIP`. In this case, the first three vertices define the first triangle (and their order is significant). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. If fewer than three vertices are specified, no primitive is produced. See figure 10.4.

The required state consists of a flag indicating if the first triangle has been completed, two stored processed vertices (called vertex A and vertex B), and a one bit pointer indicating which stored vertex will be replaced with the next vertex. When a series of vertices are transferred to the GL, the pointer is initialized to point to vertex A. Each successive vertex toggles the pointer. Therefore, the first vertex is stored as vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after the second one sent forms a triangle from vertex A, vertex B, and the current vertex (in that order).

10.1.7 Triangle Fans

A triangle fan is specified with *mode* `TRIANGLE_FAN`, and is the same as a triangle strip with one exception: each vertex after the first always replaces vertex B of the two stored vertices.

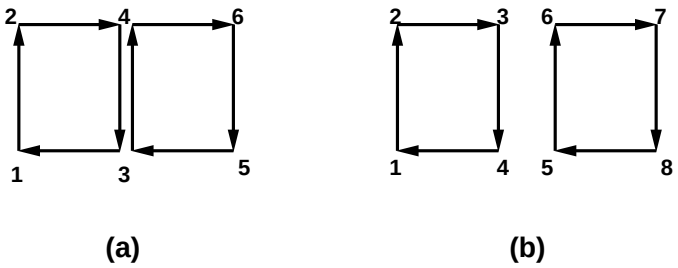


Figure 10.5. (a) A quad strip. (b) Independent quads. The numbers give the sequencing of the vertices passed to the GL.

10.1.8 Separate Triangles

Separate triangles are specified with *mode* `TRIANGLES`. In this case, the $3i + 1$st, $3i + 2$nd, and $3i + 3$rd vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n - 1$, where there are $3n + k$ vertices drawn. k is either 0, 1, or 2; if k is not zero, the final k vertices are ignored. For each triangle, vertex A is vertex $3i$ and vertex B is vertex $3i + 1$. Otherwise, separate triangles are the same as a triangle strip.

10.1.9 Quadrilateral (quad) strips

Quad strips generate a series of edge-sharing quadrilaterals, and are specified with *mode* `QUAD_STRIP`. If the m vertices passed are labelled are $v_1, \ldots, v_m$, where v_j is the jth specified vertex, then quad i has vertices (in order) v_{2i}, v_{2i+1}, v_{2i+3}, and v_{2i+2} with $i = 0, \ldots, \lfloor m/2 \rfloor$. The state required is thus three processed vertices, to store the last two vertices of the previous quad along with the third vertex (the first new vertex) of the current quad, a flag to indicate when the first quad has been completed, and a one-bit counter to count members of a vertex pair. See figure 10.5.

A quad strip with fewer than four vertices generates no primitive. If the number of vertices passed is odd, the final vertex is ignored.

10.1.10 Separate Quadrilaterals

Separate quads are specified with *mode* `QUADS`, and are just like quad strips except that each group of four vertices, the $4j + 1$st, the $4j + 2$nd, the $4j + 3$rd, and the

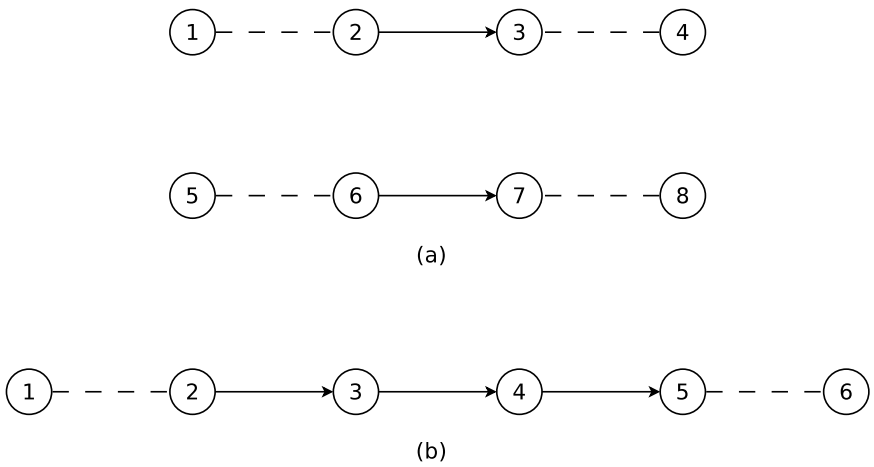


Figure 10.6. Lines with adjacency (a) and line strips with adjacency (b). The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

$4j + 4$th, generate a single quad, for $j = 0, 1, \ldots, n - 1$. The total number of vertices passed is $4n + k$, where $0 \leq k \leq 3$; if k is not zero, the final k vertices are ignored. Separate quads are generated by calling **Begin** with the argument value `QUADS`.

10.1.11 Lines with Adjacency

Lines with adjacency are specified with *mode* `LINES_ADJACENCY`, and are independent line segments where each endpoint has a corresponding *adjacent* vertex that can be accessed by a geometry shader (section 11.3). If a geometry shader is not active, the adjacent vertices are ignored.

A line segment is drawn from the $4i+2$nd vertex to the $4i+3$rd vertex for each $i = 0, 1, \ldots, n - 1$, where there are $4n + k$ vertices passed. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line segment i, the $4i + 1$st and $4i + 4$th vertices are considered adjacent to the $4i + 2$nd and $4i + 3$rd vertices, respectively (see figure 10.6).

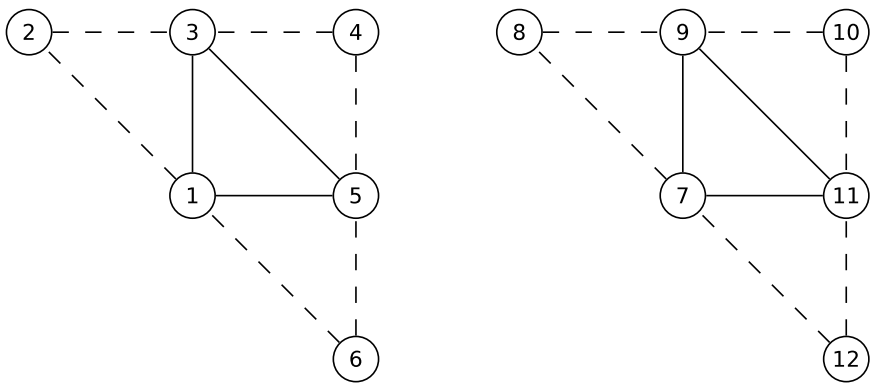


Figure 10.7. Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

10.1.12 Line Strips with Adjacency

Line strips with adjacency are specified with *mode* `LINE_STRIP_ADJACENCY` and are similar to line strips, except that each line segment has a pair of adjacent vertices that can be accessed by a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

A line segment is drawn from the $i + 2$nd vertex to the $i + 3$rd vertex for each $i = 0, 1, \ldots, n - 1$, where there are $n + 3$ vertices passed. If there are fewer than four vertices, all vertices are ignored. For line segment i, the $i + 1$st and $i + 4$th vertex are considered adjacent to the $i + 2$nd and $i + 3$rd vertices, respectively (see figure 10.6).

10.1.13 Triangles with Adjacency

Triangles with adjacency are specified with *mode* `TRIANGLES_ADJACENCY`, and are similar to separate triangles except that each triangle edge has an adjacent vertex that can be accessed by a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

The $6i + 1$st, $6i + 3$rd, and $6i + 5$th vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n - 1$, where there are $6n + k$ vertices passed. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i,

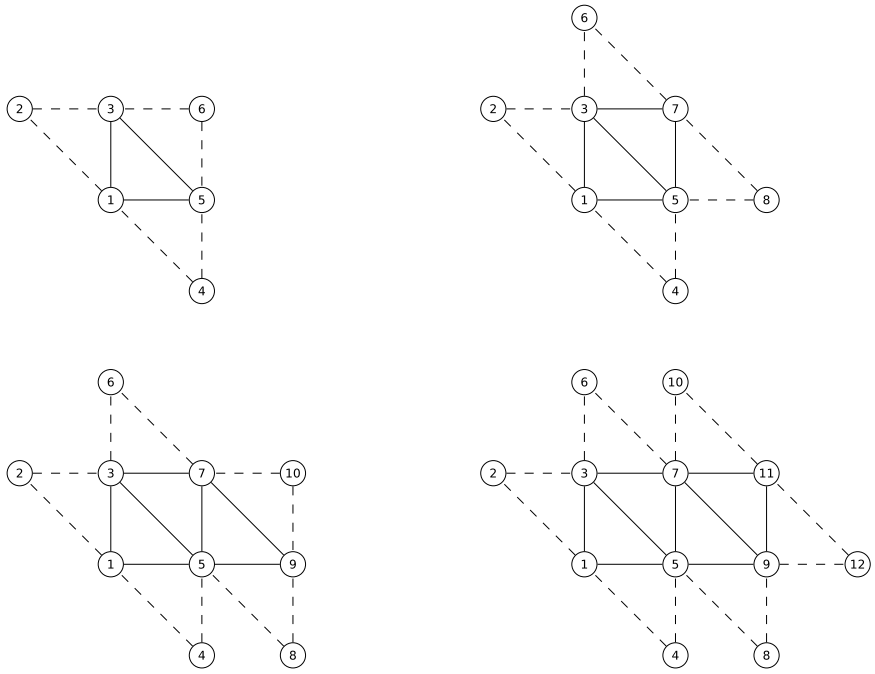


Figure 10.8. Triangle strips with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

the $i + 2$nd, $i + 4$th, and $i + 6$th vertices are considered adjacent to edges from the $i + 1$st to the $i + 3$rd, from the $i + 3$rd to the $i + 5$th, and from the $i + 5$th to the $i + 1$st vertices, respectively (see figure 10.7).

10.1.14 Triangle Strips with Adjacency

Triangle strips with adjacency are specified with *mode* `TRIANGLE_STRIP_ADJACENCY`, and are similar to triangle strips except that each triangle edge has an adjacent vertex that can be accessed by a geometry shader (see section 11.3). If a geometry shader is not active, the adjacent vertices are ignored.

In triangle strips with adjacency, n triangles are drawn where there are $2(n + 2) + k$ vertices passed. k is either 0 or 1; if k is 1, the final vertex is ignored.

Primitive	Primitive Vertices			Adjacent Vertices		
	1st	2nd	3rd	1/2	2/3	3/1
only ($i = 0, n = 1$)	1	3	5	2	6	4
first ($i = 0$)	1	3	5	2	7	4
middle (i odd)	$2i + 3$	$2i + 1$	$2i + 5$	$2i - 1$	$2i + 4$	$2i + 7$
middle (i even)	$2i + 1$	$2i + 3$	$2i + 5$	$2i - 1$	$2i + 7$	$2i + 4$
last ($i = n - 1$, i odd)	$2i + 3$	$2i + 1$	$2i + 5$	$2i - 1$	$2i + 4$	$2i + 6$
last ($i = n - 1$, i even)	$2i + 1$	$2i + 3$	$2i + 5$	$2i - 1$	$2i + 6$	$2i + 4$

Table 10.1: Triangles generated by triangle strips with adjacency. Each triangle is drawn using the vertices whose numbers are in the *1st*, *2nd*, and *3rd* columns under *primitive vertices*, in that order. The vertices in the 1/2, 2/3, and 3/1 columns under *adjacent vertices* are considered adjacent to the edges from the first to the second, from the second to the third, and from the third to the first vertex of the triangle, respectively. The six rows correspond to six cases: the first and only triangle ($i = 0, n = 1$), the first triangle of several ($i = 0, n > 0$), "odd" middle triangles ($i = 1, 3, 5 \ldots$), "even" middle triangles ($i = 2, 4, 6, \ldots$), and special cases for the last triangle, when i is either even or odd. For the purposes of this table, the first vertex passed is numbered 1 and the first triangle is numbered 0.

If there are fewer than 6 vertices, the entire primitive is ignored. The order of vertices used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle (see figure 10.8), is implementation defined but must be according to either table 10.1 or 10.2.

10.1.15 Separate Patches

Separate patches are specified with mode `PATCHES`. A patch is an ordered collection of vertices used for primitive tessellation (section 11.2). The vertices comprising a patch have no implied geometric ordering. The vertices of a patch are used by tessellation shaders and the fixed-function tessellator to generate new point, line, or triangle primitives.

Each patch in the series has a fixed number of vertices, which is specified by calling

`void` **PatchParameteri**(`enum` *pname*, `int` *value*);

with *pname* set to `PATCH_VERTICES`.

Primitive	Primitive Vertices			Adjacent Vertices		
	1st	2nd	3rd	1/2	2/3	3/1
only ($i = 0, n = 1$)	1	3	5	2	6	4
first ($i = 0$)	1	3	5	2	7	4
middle (i odd)	$2i + 1$	$2i + 5$	$2i + 3$	$2i + 4$	$2i + 7$	$2i - 1$
middle (i even)	$2i + 1$	$2i + 3$	$2i + 5$	$2i - 1$	$2i + 7$	$2i + 4$
last ($i = n - 1$, i odd)	$2i + 1$	$2i + 5$	$2i + 3$	$2i + 4$	$2i + 6$	$2i - 1$
last ($i = n - 1$, i even)	$2i + 1$	$2i + 3$	$2i + 5$	$2i - 1$	$2i + 6$	$2i + 4$

Table 10.2: Alternative ordering to triangles generated by triangle strips with adjacency (see table 10.1).

Errors

An INVALID_ENUM error is generated if *pname* is not PATCH_VERTICES.

An INVALID_VALUE error is generated if *value* is less than or equal to zero, or greater than the implementation-dependent maximum patch size (the value of MAX_PATCH_VERTICES). The patch size is initially three vertices.

If the number of vertices in a patch is given by v, the $vi + 1$st through $vi + v$th vertices (in that order) determine a patch for each $i = 0, 1, \ldots n - 1$, where there are $vn + k$ vertices. k is in the range $[0, v - 1]$; if k is not zero, the final k vertices are ignored.

10.1.16 General Considerations For Polygon Primitives

Depending on the current state of the GL, a *polygon primitive* generated from a drawing command with *mode* POLYGON, QUADS, QUAD_STRIP, TRIANGLE_FAN, TRIANGLE_STRIP, TRIANGLES, TRIANGLES_ADJACENCY, or TRIANGLE_STRIP_ADJACENCY may be rendered in one of several ways, such as outlining its border or filling its interior. The order of vertices in such a primitive is significant in lighting (see section 12.2.1.1), polygon rasterization (see section 14.6.1) and fragment shading (see section 15.2.2).

Only convex polygons are guaranteed to be drawn correctly by the GL. If a specified polygon is nonconvex when projected onto the window, then the rendered polygon need only lie within the convex hull of the projected vertices defining its boundary.

10.1.17 Polygon Edges

Each edge of each polygon primitive generated is flagged as either *boundary* or *non-boundary*. These classifications are used during polygon rasterization; some modes affect the interpretation of polygon boundary edges (see section 14.6.4). By default, all edges are boundary edges, but the flagging of polygons, separate triangles, or separate quadrilaterals may be altered by calling

```
void EdgeFlag( boolean flag );
void EdgeFlagv( const boolean *flag );
```

to change the value of a flag bit. If *flag* is zero, then the flag bit is set to `FALSE`; if *flag* is non-zero, then the flag bit is set to `TRUE`.

When a primitive of type `POLYGON`, `TRIANGLES`, or `QUADS` is drawn, each vertex transferred begins an edge. If the edge flag bit is `TRUE`, then each specified vertex begins an edge that is flagged as boundary. If the bit is `FALSE`, then induced edges are flagged as non-boundary.

The state required for edge flagging consists of one current flag bit. Initially, the bit is `TRUE`. In addition, each processed vertex of an assembled polygonal primitive must be augmented with a bit indicating whether or not the edge beginning on that vertex is boundary or non-boundary.

10.2 Current Vertex Attribute Values

The commands in this section are used to specify *current attribute values*. These values are used by drawing commands to define the attributes transferred for a vertex when a vertex array defining a required attribute is not enabled, as described in section 10.3.

Current attribute values also define the values transferred when specifying vertices between **Begin** and **End**, as described in section 10.7.

10.2.1 Current Generic Attributes

Vertex shaders (see section 11.1) access an array of 4-component *generic vertex attributes*. The first slot of this array is numbered zero, and the size of the array is specified by the value of the implementation-dependent constant `MAX_VERTEX_ATTRIBS`.

The current values of a generic shader attribute declared as a floating-point scalar, vector, or matrix may be changed at any time by issuing one of the commands

```
void VertexAttrib{1234}{sfd}( uint index, T values );
void VertexAttrib{123}{sfd}v( uint index, const
    T *values );
void VertexAttrib4{bsifd ub us ui}v( uint index, const
    T *values );
void VertexAttrib4Nub( uint index, ubyte x, ubyte y,
    ubyte z, ubyte w );
void VertexAttrib4N{bsi ub us ui}v( uint index, const
    T *values );
void VertexAttribI{1234}{i ui}( uint index, T values );
void VertexAttribI{1234}{i ui}v( uint index, const
    T *values );
void VertexAttribI4{b s ub us}v( uint index, const
    T *values );
void VertexAttribL{1234}d( uint index, const T values );
void VertexAttribL{1234}dv( uint index, const
    T *values );
void VertexAttribP{1234}ui(uint index,enum
    type,boolean normalized,uint value);
void VertexAttribP{1234}uiv(uint index,enum
    type,boolean normalized,const uint *value);
```

The **VertexAttrib4N*** commands specify fixed-point values that are converted to a normalized $[0, 1]$ or $[-1, 1]$ range as described in equations 2.1 and 2.2, respectively.

The **VertexAttribI*** commands specify signed or unsigned fixed-point values that are stored as signed or unsigned integers, respectively. Such values are referred to as *pure integers*.

The **VertexAttribL*** commands specify double-precision values that will be stored as double-precision values.

The **VertexAttribP*** commands specify up to four attribute component values packed into a single natural type *type* as described in section 10.3.8. *type* must be `INT_2_10_10_10_REV`, `UNSIGNED_INT_2_10_10_10_REV`, or `UNSIGNED_INT_10F_11F_11F_REV`, specifying signed, unsigned, or unsigned floating-point data, respectively. The first one (x), two (x, y), three (x, y, z), or four (x, y, z, w) components of the packed data are consumed by **VertexAttribP1ui**, **VertexAttribP2ui**, **VertexAttribP3ui**, and **VertexAttribP4ui**, respectively. If *normalized* is `TRUE`, signed or unsigned components are converted to floating-point by normalizing to $[-1, 1]$ or $[0, 1]$ respectively. If *normalized* is `FALSE`, signed and unsigned components are directly cast to floating-point. For floating-point formats, *normal-*

ized is ignored. The number of components specified must be no greater than the number of components in the packed *type*. For **VertexAttribP*uiv**, *value* contains the address of a single uint containing the packed attribute components.

All other **VertexAttrib*** commands specify values that are converted directly to the internal floating-point representation.

The resulting value(s) are loaded into the generic attribute at slot *index*, whose components are named *x*, *y*, *z*, and *w*. The **VertexAttrib1*** family of commands sets the *x* coordinate to the provided single argument while setting *y* and *z* to 0 and *w* to 1. Similarly, **VertexAttrib2*** commands set *x* and *y* to the specified values, *z* to 0 and *w* to 1; **VertexAttrib3*** commands set *x*, *y*, and *z*, with *w* set to 1, and **VertexAttrib4*** commands set all four coordinates.

The **VertexAttrib*** entry points may also be used to load shader attributes declared as a floating-point matrix. Each column of a matrix takes up one generic 4-component attribute slot out of the `MAX_VERTEX_ATTRIBS` available slots. Matrices are loaded into these slots in column major order. Matrix columns are loaded in increasing slot numbers.

When values for a vertex shader attribute variable are sourced from a current generic attribute value, the attribute must be specified by a command compatible with the data type of the variable. The values loaded into a shader attribute variable bound to generic attribute *index* are undefined if the current value for attribute *index* was not specified by

- **VertexAttrib[1234]*** or **VertexAttribP***, for single-precision floating-point scalar, vector, and matrix types
- **VertexAttribI[1234]i** or **VertexAttribI[1234]iv**, for signed integer scalar and vector types
- **VertexAttribI[1234]ui** or **VertexAttribI[1234]uiv**, for unsigned integer scalar and vector types
- **VertexAttribL***, for double-precision floating-point scalar and vector types.

Setting any generic vertex attribute updates the current values of the attribute. In addition, setting generic vertex attribute zero specifies a vertex, as described in section 10.7.2[1].

There is no aliasing among generic attributes and conventional attributes. In other words, an application can set all `MAX_VERTEX_ATTRIBS` generic attributes

[1]This is a change in behavior of the compatibilty profile for increased interoperability with core profile and OpenGL ES. Previously, generic vertex attribute zero did not have current values, and could not be queried.

and all conventional attributes without fear any attribute overwriting the value of another attribute.

Errors

An `INVALID_VALUE` error is generated for all **VertexAttrib*** commands if *index* is greater than or equal to the value of `MAX_VERTEX_ATTRIBS`.

An `INVALID_ENUM` error is generated by **VertexAttribP4ui** and **VertexAttribP4uiv** if *type* is `UNSIGNED_INT_10F_11F_11F_REV`.

10.2.2 Current Conventional Attributes

When using fixed-function vertex processing, or a vertex shader using conventional vertex attributes, current values for these attributes are defined by the commands

```
void TexCoord{1234}{sifd}( T coords );
void TexCoord{1234}{sifd}v( const T *coords );
```

specify the current homogeneous texture coordinates, named s, t, r, and q.

Texture coordinates may be stored as packed components within a larger natural type. Such data may be specified using

```
void TexCoordP{1234}ui(enum type,uint coords);
void TexCoordP{1234}uiv(enum type,const uint
   *coords);
```

This command specifies up to four components as described above, packed into a single natural type as described in section 10.3.8. The *type* parameter must be `INT_2_10_10_10_REV` or `UNSIGNED_INT_2_10_10_10_REV`, specifying signed or unsigned data, respectively. The first one (x), two (x, y), three (x, y, z), or four (x, y, z, w) components of the packed data are consumed by **TexCoordP1ui***, **TexCoordP2ui***, **TexCoordP3ui***, and **TexCoordP4ui***, respectively. For **TexCoordP*uiv**, *coords* contains the address of a single `uint` containing the packed texture coordinate components.

The **TexCoord*1*** family of commands set the s coordinate to the provided single argument while setting t and r to 0 and q to 1. Similarly, **TexCoord*2*** sets s and t to the specified values, r to 0 and q to 1; **TexCoord*3*** sets s, t, and r, with q set to 1, and **TexCoord*4*** sets all four texture coordinates.

Implementations must support at least two sets of texture coordinates. The commands

```
void MultiTexCoord{1234}{sifd}(enum texture, T coords);
void MultiTexCoord{1234}{sifd}v(enum texture, const T
    *coords);
void MultiTexCoordP{1234}ui(enum texture, enum
    type, uint coords);
void MultiTexCoordP{1234}uiv(enum texture, enum
    type, const uint *coords);
```

take the coordinate set to be modified as the *texture* parameter. *texture* is one of the tokens TEXTUREi, indicating that texture coordinate set i is to be modified. i must be in the range zero to the value of MAX_TEXTURE_COORDS minus one.

The **TexCoord*** commands are exactly equivalent to the corresponding **MultiTexCoord*** commands with *texture* set to TEXTURE0.

Specifying an invalid texture coordinate set for the *texture* argument of **MultiTexCoord** results in undefined behavior.

The current normal is set using

```
void Normal3{bsifd}( T coords );
void Normal3{bsifd}v( const T *coords );
```

Byte, short, or integer values passed to **Normal** are converted to floating-point values as described in equation 2.2 for the corresponding (signed) type.

Normals may be stored as packed components within a larger natural type. Such data may be specified using

```
void NormalP3ui(enum type, uint normal);
void NormalP3uiv(enum type, uint *normal);
```

This specifies a three component normal, packed into the first three (x, y, z) components of the natural type as described in section 10.3.8. *type* must be INT_2_10_10_10_REV or UNSIGNED_INT_2_10_10_10_REV, specifying signed or unsigned data, respectively. Individual signed or unsigned components are converted to floating-point values according to equations 2.1 or 2.2, respectively. For **NormalP3uiv**, *normal* contains the address of a single uint containing the packed normal components.

The current fog coordinate is set using

```
void FogCoord{fd}( T coord );
void FogCoord{fd}v( const T *coord );
```

There are several ways to set the current color and secondary color. The GL stores a current single-valued *color index*, as well as a current four-valued RGBA color and secondary color. Either the index or the color and secondary color are significant depending as the GL is in *color index mode* or *RGBA mode*. The mode selection is made when the GL is initialized.

The commands to set RGBA colors are

```
void Color{34}{bsifd ubusui}( T components );
void Color{34}{bsifd ubusui}v( const T *components );
void SecondaryColor3{bsifd ubusui}( T components );
void SecondaryColor3{bsifd ubusui}v( const
    T *components );
```

The **Color** command has two major variants: **Color3** and **Color4**. The four value versions set all four values. The three value versions set R, G, and B to the provided values; A is set to 1.0. (The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 12.2.)

The secondary color has only the three value versions. Secondary A is always set to 1.0.

Versions of the **Color** and **SecondaryColor** commands that take floating-point values accept values nominally between 0.0 and 1.0. 0.0 corresponds to the minimum while 1.0 corresponds to the maximum (machine dependent) value that a component may take on in the framebuffer (see section 12.2 on colors and coloring). Values outside $[0, 1]$ are not clamped.

RGBA colors may be stored as packed components within a larger natural type. Such data may be specified using

```
void ColorP{34}ui(enum type,uint coords);
void ColorP{34}uiv(enum type,const uint *coords);
void SecondaryColorP3ui(enum type,uint coords);
void SecondaryColorP3uiv(enum type,const uint
    *coords);
```

The **ColorP*** commands set the primary color similarly to **Color***, above. The **SecondaryColorP*** commands set the secondary color similarly to **SecondaryColor***. *type* must be `INT_2_10_10_10_REV` or `UNSIGNED_INT_2_10_10_10_REV`, specifying signed or unsigned data, respectively. Colors are packed into a single natural type as described in section 10.3.8. The first three (x, y, z) or four (x, y, z, w) components of the packed data are consumed by ***ColorP3ui*** and ***ColorP4ui***, respectively. Individual signed or unsigned components are converted to floating-point values according to equations 2.1 or 2.2, respectively. For

ColorP*uiv and **SecondaryColorP*uiv**, *coords* contains the address of a single `uint` containing the packed color components.

The command

```
void Index{sifd ub}( T index );
void Index{sifd ub}v( const T *index );
```

updates the current (single-valued) color index. It takes one argument, the value to which the current color index should be set. Values outside the (machine-dependent) representable range of color indices are not clamped.

10.2.3 Vertex Attribute Queries

Current generic vertex attribute values may be queried using the **GetVertexAttrib*** commands as described in section 10.5.

10.2.4 Required State

The state required to support vertex specification consists of four floating-point numbers per texture coordinate set to store the current texture coordinates s, t, r, and q, three floating-point numbers to store the three coordinates of the current normal, one floating-point number to store the current fog coordinate, four floating-point values to store the current RGBA color, four floating-point values to store the current RGBA secondary color, one floating-point value to store the current color index, and the value of `MAX_VERTEX_ATTRIBS` minus one four-component vectors to store generic vertex attributes.

The initial texture coordinates are $(s, t, r, q) = (0, 0, 0, 1)$ for each texture coordinate set. The initial current normal has coordinates $(0, 0, 1)$. The initial fog coordinate is zero. The initial RGBA color is $(R, G, B, A) = (1, 1, 1, 1)$ and the initial RGBA secondary color is $(0, 0, 0, 1)$. The initial color index is 1.

The initial values for all generic vertex attributes are $(0.0, 0.0, 0.0, 1.0)$.

10.3 Vertex Arrays

Vertex data may also be placed into arrays that are stored in the client's address space (described here) or in the server's address space (described in section 10.3.9). Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command.

All of the state required to represent the vertex arrays is stored in a vertex array object.

10.3.1 Vertex Array Objects

The buffer objects that are to be used by the vertex stage of the GL are collected together to form a vertex array object. All state related to the definition of data used by the vertex processor is encapsulated in a vertex array object.

The name space for vertex array objects is the unsigned integers, with zero reserved by the GL to represent the default vertex array object.

The command

```
void GenVertexArrays( sizei n, uint *arrays );
```

returns *n* previous unused vertex array object names in *arrays*. These names are marked as used, for the purposes of **GenVertexArrays** only, but they acquire array state only when they are first bound.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

Vertex array objects are deleted by calling

```
void DeleteVertexArrays( sizei n, const uint *arrays );
```

arrays contains *n* names of vertex array objects to be deleted. Once a vertex array object is deleted it has no contents and its name is again unused. If a vertex array object that is currently bound is deleted, the binding for that object reverts to zero and the default vertex array becomes current. Unused names in *arrays* that have been marked as used for the purposes of **GenVertexArrays** are marked as unused again. Unused names in *arrays* are silently ignored, as is the value zero.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

A vertex array object is created by binding a name returned by **GenVertexArrays** with the command

```
void BindVertexArray( uint array );
```

array is the vertex array object name. The resulting vertex array object is a new state vector, comprising all the state and with the same initial values listed in tables 23.4- 23.7.

BindVertexArray may also be used to bind an existing vertex array object. If the bind is successful no change is made to the state of the bound vertex array object, and any previous binding is broken.

The currently bound vertex array object is used for all commands which modify vertex array state, such as **VertexAttribPointer** and **EnableVertexAttribArray**; all commands which draw from vertex arrays, such as **DrawArrays** and **DrawElements**; and all queries of vertex array state (see chapter 22).

Errors

An `INVALID_OPERATION` error is generated if *array* is not zero or a name returned from a previous call to **CreateVertexArrays** or **GenVertexArrays**, or if such a name has since been deleted with **DeleteVertexArrays**.

Vertex array objects may also be created with the command

`void` **CreateVertexArrays**(`sizei` *n*, `uint` **arrays*);

CreateVertexArrays returns *n* previously unused vertex array object names in *arrays*, each representing a state vector comprising all the state and with the same initial values listed in tables 23.4- 23.7.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

The command

`boolean` **IsVertexArray**(`uint` *array*);

returns `TRUE` if *array* is the name of a vertex array object. If *array* is zero, or a non-zero value that is not the name of a vertex array object, **IsVertexArray** returns `FALSE`. No error is generated if *array* is not a valid vertex array object name.

To bind a buffer object to the element array buffer bind point of a vertex array object, use the command

`void` **VertexArrayElementBuffer**(`uint` *vaobj*, `uint` *buffer*);

vaobj is zero, indicating the default vertex array object, or the name of the vertex array object, and *buffer* is zero or the name of the buffer object. If *buffer* is zero, any existing element array buffer binding to *vaobj* is removed.

Errors

An INVALID_OPERATION error is generated if *vaobj* is not the name of an existing vertex array object.

An INVALID_OPERATION error is generated if *buffer* is not zero or the name of an existing buffer object.

10.3.2 Specifying Arrays for Generic Vertex Attributes

To specify the organization of arrays storing generic vertex attributes of a vertex array object, use the commands

The commands

```
void VertexAttribFormat( uint attribindex, int size,
    enum type, boolean normalized, uint relativeoffset );
void VertexAttribIFormat( uint attribindex, int size,
    enum type, uint relativeoffset );
void VertexAttribLFormat( uint attribindex, int size,
    enum type, uint relativeoffset );
void VertexArrayAttribFormat( uint vaobj,
    uint attribindex, int size, enum type,
    boolean normalized, uint relativeoffset );
void VertexArrayAttribIFormat( uint vaobj,
    uint attribindex, int size, enum type, uint relativeoffset );
void VertexArrayAttribLFormat( uint vaobj,
    uint attribindex, int size, enum type, uint relativeoffset );
```

For **VertexAttrib*Format**, the vertex array object is that bound to VERTEX_ARRAY_BINDING. For **VertexArrayAttrib*Format**, *vaobj* is zero, indicating the default vertex array object, or the name of the vertex array object.

attribindex identifies the generic vertex attribute array. *size* indicates the number of values per vertex that are stored in the array, as well as their component ordering. *type* specifies the data type of the values stored in the array.

Table 10.3 indicates the allowable values for *size* and *type*. A *type* of BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, FLOAT, HALF_FLOAT, or DOUBLE indicates the corresponding GL data type shown in table 8.7. A *type* of FIXED indicates the data type fixed. A *type* of INT_2_10_10_10_REV or UNSIGNED_INT_2_10_10_10_REV indicates respectively, four signed or unsigned elements packed into a single uint. A *type*

Command	*size*s and Component Ordering	Integer Handling	*type*s
VertexAttribFormat	1, 2, 3, 4, BGRA	flag	byte, ubyte, short, ushort, int, uint, fixed, float, half, double, *packed*
VertexAttribIFormat	1, 2, 3, 4	integer	byte, ubyte, short, ushort, int, uint
VertexAttribLFormat	1, 2, 3, 4	n/a	double

Table 10.3: Vertex array sizes (values per vertex) and data types for generic vertex attributes. See the body text for a full description of each column.

of UNSIGNED_INT_10F_11F_11F_REV indicates two unsigned 11-bit floating-point elements and one unsigned 10-bit floating-point element packed into a single uint. Encoding of the unsigned 11- and 10-bit floating-point values is described in sections 2.3.4.3 and 2.3.4.4, respectively. The types INT_2_10_10_10_REV, UNSIGNED_INT_2_10_10_10_REV and UNSIGNED_INT_10F_11F_11F_REV all correspond to the term *packed* in table 10.3. The components are packed as shown in table 8.13. *packed* is not a GL type, but indicates commands accepting multiple components packed into a single uint.

The "Integer Handling" column in tables 10.3 and 10.4 indicates how integer and fixed-point data types are handled. "cast" means that they are converted to floating-point directly. "normalize" means that they are converted to floating-point by normalizing to $[0, 1]$ (for unsigned types) or $[-1, 1]$ (for signed types), as described in equations 2.1 and 2.2, respectively. "integer" means that they remain as integer values. "flag" means that either "cast" or "normalized" applies, depending on whether the *normalized* flag to the command is TRUE or FALSE, respectively.

The *normalized* flag is ignored for floating-point data types, including fixed, float, half, double, and any packed types that have floating-point components.

If *size* is BGRA, vertex array values are always normalized, irrespective of the "normalize" table entry.

If *type* is UNSIGNED_INT_10F_11F_11F_REV, vertex array values are never normalized, irrespective of the "normalize" table entry.

relativeoffset is a byte offset of the first element relative to the start of the vertex buffer binding this attribute fetches from.

Errors

An INVALID_OPERATION error is generated by **VertexArrayAttrib*Format** if *vaobj* is not zero or the name of an existing vertex array object.

An INVALID_VALUE error is generated if *attribindex* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

An INVALID_VALUE error is generated if *size* is not one of the values shown in table 10.3 for the corresponding command.

An INVALID_ENUM error is generated if *type* is not one of the parameter token names from table 8.7 corresponding to one of the allowed GL data types for that command as shown in table 10.3.

An INVALID_ENUM error is generated by **VertexAttribIFormat** and **VertexAttribLFormat** if *type* is UNSIGNED_INT_10F_11F_11F_REV.

An INVALID_OPERATION error is generated under any of the following conditions:

- *size* is BGRA and *type* is not UNSIGNED_BYTE, INT_2_10_10_10_REV or UNSIGNED_INT_2_10_10_10_REV;
- *type* is INT_2_10_10_10_REV or UNSIGNED_INT_2_10_10_10_REV, and *size* is neither 4 nor BGRA;
- *type* is UNSIGNED_INT_10F_11F_11F_REV and *size* is not 3;
- *size* is BGRA and *normalized* is FALSE.

An INVALID_VALUE error is generated if *relativeoffset* is larger than the value of MAX_VERTEX_ATTRIB_RELATIVE_OFFSET.

The source of data for a generic vertex attribute may be determined by attaching a buffer object to a vertex array object with the commands

```
void BindVertexBuffer( uint bindingindex, uint buffer,
    intptr offset, sizei stride );
void VertexArrayVertexBuffer( uint vaobj,
    uint bindingindex, uint buffer, intptr offset,
    sizei stride );
```

For **BindVertexBuffer**, the vertex array object is the currently bound vertex array object. For **VertexArrayVertexBuffer**, *vaobj* is zero, indicating the default vertex array object, or the name of the vertex array object.

buffer is either zero or an unused name.

If *buffer* is zero, any buffer object bound to *bindingindex* is detached.

If *buffer* is not the name of an existing buffer object, the GL first creates a new state vector, initialized with a zero-sized memory buffer and comprising all the state and with the same initial values listed in table 6.2, just as for **BindBuffer**. *buffer* is then attached to the specified *bindingindex* of the vertex array object.

When sourcing vertex data from the buffer object, *offset* specifies the offset in basic machine units of the first element in the vertex buffer. Pointers to the ith and $(i+1)$st elements of the array differ by *stride* basic machine units, the pointer to the $(i+1)$st element being greater.

If the operation is successful no change is made to the state of the newly bound buffer object, and any previous binding to *bindingindex* is broken.

Errors

An `INVALID_OPERATION` error is generated by **VertexArrayVertexBuffer** if *vaobj* is not zero or the name of an existing vertex array object.

An `INVALID_VALUE` error is generated if *bindingindex* is greater than or equal to the value of `MAX_VERTEX_ATTRIB_BINDINGS`.

An `INVALID_VALUE` error is generated if *stride* or *offset* is negative, or if *stride* is greater than the value of `MAX_VERTEX_ATTRIB_STRIDE`.

The source of data for multiple vertex attributes may be determined by attaching multiple existing buffer objects to a vertex array object with the commands

```
void BindVertexBuffers( uint first, sizei count, const
    uint *buffers, const intptr *offsets, const
    sizei *strides );
void VertexArrayVertexBuffers( uint vaobj, uint first,
    sizei count, const uint *buffers, const
    intptr *offsets, const sizei *strides );
```

For **BindVertexBuffers**, the vertex array object is the currently bound vertex array object. For **VertexArrayVertexBuffers**, *vaobj* is zero, indicating the default vertex array object, or the name of the vertex array object.

count existing buffer objects are bound to vertex buffer binding points numbered *first* through $first + count - 1$. If *buffers* is not `NULL`, it specifies an array of *count* values, each of which must be zero or the name of an existing buffer object. *offsets* and *strides* specify arrays of *count* values indicating the offset of the first element and stride between elements in each buffer, respectively. If *buffers* is `NULL`, each affected vertex buffer binding point from *first* through $first+count-1$ will be reset to have no bound buffer object. In this case, the offsets and strides

associated with the binding points are set to default values, ignoring *offsets* and *strides*.

BindVertexBuffers is equivalent (assuming no errors are generated) to:

```
for (i = 0; i < count; i++) {
   if (buffers == NULL) {
      BindVertexBuffer(first + i, 0, 0, 16);
   } else {
      BindVertexBuffer(first + i, buffers[i], offsets[i],
         strides[i]);
   }
}
```

except that buffers will not be created if they do not exist.

VertexArrayVertexBuffers is equivalent to the pseudocode above, but replacing **BindVertexBuffer**(*args*) with **VertexArrayVertexBuffers**(*vaobj*, *args*).

The values specified in *buffers*, *offsets*, and *strides* will be checked separately for each vertex buffer binding point. When a value for a specific vertex buffer binding point is invalid, the state for that binding point will be unchanged and an error will be generated. However, state for other vertex buffer binding points will still be changed if their corresponding values are valid.

Errors

An INVALID_OPERATION error is generated by **VertexArrayVertexBuffers** if *vaobj* is not zero or the name of an existing vertex array object.

An INVALID_OPERATION error is generated if $first + count$ is greater than the value of MAX_VERTEX_ATTRIB_BINDINGS.

An INVALID_VALUE error is generated if *count* is negative.

An INVALID_OPERATION error is generated if any value in *buffers* is not zero or the name of an existing buffer object (per binding).

An INVALID_VALUE error is generated if any value in *offsets* or *strides* is negative, or if any value in *strides* is greater than the value of MAX_VERTEX_ATTRIB_STRIDE (per binding).

The association between a vertex attribute and the vertex buffer binding used by that attribute is set by the command

```
void VertexAttribBinding( uint attribindex,
   uint bindingindex );
```

void **VertexArrayAttribBinding**(uint *vaobj*, *uint attribindex*, *uint bindingindex*);

For **VertexAttribBinding**, the vertex array object is the currently bound vertex array object. For **VertexArrayAttribBinding**, *vaobj* is zero, indicating the default vertex array object, or the name of the vertex array object.

Errors

An INVALID_OPERATION error is generated by **VertexArrayAttrib-Binding** if *vaobj* is not zero or the name of an existing vertex array object.

An INVALID_VALUE error is generated if *attribindex* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

An INVALID_VALUE error is generated if *bindingindex* is greater than or equal to the value of MAX_VERTEX_ATTRIB_BINDINGS.

The one, two, three, or four values in an array that correspond to a single vertex comprise an array *element*. When size is BGRA, it indicates four values. The values within each array element are stored sequentially in memory. However, if *size* is BGRA, the first, second, third, and fourth values of each array element are taken from the third, second, first, and fourth values in memory respectively.

The commands

void **VertexAttribPointer**(uint *index*, int *size*, enum *type*, boolean *normalized*, sizei *stride*, const void **pointer*);
void **VertexAttribIPointer**(uint *index*, int *size*, enum *type*, sizei *stride*, const void **pointer*);
void **VertexAttribLPointer**(uint *index*, int *size*, enum *type*, sizei *stride*, const void **pointer*);

control vertex attribute state, a vertex buffer binding, and the mapping between a vertex attribute and a vertex buffer binding. They are equivalent (assuming no errors are generated) to:

VertexAttrib*Format(*index*, *size*, *type*, {*normalized*, }, 0);
VertexAttribBinding(*index*, *index*);
if (*stride* != 0) {
    effectiveStride = *stride*;
} else {
    compute effectiveStride based on *size* and *type*;

```
}
VERTEX_ATTRIB_ARRAY_STRIDE[index] = stride;
// This sets VERTEX_BINDING_STRIDE to effectiveStride
VERTEX_ATTRIB_ARRAY_POINTER[index] = pointer;
BindVertexBuffer(index, buffer bound to ARRAY_BUFFER,
    (char *)pointer - (char *)NULL, effectiveStride);
```

If *stride* is specified as zero, then array elements are stored sequentially.

Errors

An `INVALID_VALUE` error is generated if *stride* is negative.

An `INVALID_VALUE` error is generated if *stride* is greater than the value of `MAX_VERTEX_ATTRIB_STRIDE`.

An `INVALID_OPERATION` error is generated if a non-zero vertex array object is bound, no buffer is bound to `ARRAY_BUFFER`, and *pointer* is not `NULL`.

In addition, any of the errors defined by **VertexAttrib*Format** and **VertexAttribBinding** may be generated if the parameters passed to those commands in the equivalent code above would generate those errors.

An individual generic vertex attribute array in a vertex array object is enabled with the commands

```
void EnableVertexAttribArray( uint index );
void EnableVertexArrayAttrib( uint vaobj, uint index );
```

and is disabled with the commands

```
void DisableVertexAttribArray( uint index );
void DisableVertexArrayAttrib( uint vaobj, uint index );
```

index identifies the generic vertex attribute array to enable or disable. For **EnableVertexAttribArray** and **DisableVertexAttribArray**, the vertex array object is the currently bound vertex array object. For **EnableVertexArrayAttrib** and **DisableVertexArrayAttrib**, *vaobj* is zero, indicating the default vertex array object, or the name of the vertex array object.

Errors

An `INVALID_OPERATION` error is generated by **EnableVertexArrayAttrib** and **DisableVertexArrayAttrib** if *vaobj* is not zero or the name of an

existing vertex array object.

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

10.3.3 Specifying Arrays for Fixed-Function Attributes

The commands

```
void VertexPointer( int size, enum type, sizei stride,
    const void *pointer );
void NormalPointer( enum type, sizei stride, const
    void *pointer );
void ColorPointer( int size, enum type, sizei stride,
    const void *pointer );
void SecondaryColorPointer( int size, enum type,
    sizei stride, const void *pointer );
void IndexPointer( enum type, sizei stride, const
    void *pointer );
void EdgeFlagPointer( sizei stride, const void *pointer );
void FogCoordPointer( enum type, sizei stride, const
    void *pointer );
void TexCoordPointer( int size, enum type, sizei stride,
    const void *pointer );
```

specify the location and organization of arrays to store vertex coordinates, normals, colors, secondary colors, color indices, edge flags, fog coordinates, and the value of MAX_TEXTURE_COORDS multiple texture coordinate sets, respectively.

size, *type*, *stride*, and *pointer* have the same meaning as in **VertexAttrib*Pointer** , and table 10.4 indicates the allowable values of *size* and *type* for these commands in the same fashion as table 10.3. *type* UNSIGNED_INT_10F_11F_11F_REV is not permitted for any of these commands.

Because edge flags are always type boolean, **EdgeFlagPointer** has no *type* argument.

Because normals are always specified with three values, **NormalPointer** has no *size* argument. Likewise, because color indices and edge flags are always specified with a single value, **IndexPointer** and **EdgeFlagPointer** also have no *size* argument.

Errors

Command	*size*s and Component Ordering	Integer Handling	*type*s
VertexPointer	2, 3, 4	cast	`short`, `int`, `float`, `half`, `double`, *packed*
NormalPointer	3	normalize	`byte`, `short`, `int`, `float`, `half`, `double`, *packed*
ColorPointer	3, 4, `BGRA`	normalize	`byte`, `ubyte`, `short`, `ushort`, `int`, `uint`, `float`, `half`, `double`, *packed*
SecondaryColorPointer	3, `BGRA`	normalize	`byte`, `ubyte`, `short`, `ushort`, `int`, `uint`, `float`, `half`, `double`, *packed*
IndexPointer	1	cast	`ubyte`, `short`, `int`, `float`, `double`
FogCoordPointer	1	n/a	`float`, `half`, `double`
TexCoordPointer	1, 2, 3, 4	cast	`short`, `int`, `float`, `half`, `double`, *packed*
EdgeFlagPointer	1	integer	`boolean`

Table 10.4: Fixed-function vertex array sizes (values per vertex) and data types. Columns are interpreted in the same fashion as table 10.3.

Errors for these commands are the same as for **VertexAttrib*Pointer**, although not all error conditions described apply since these commands do not take all of the same parameters as **VertexAttrib*Pointer**.

An INVALID_VALUE error is generated if *stride* is negative.

An INVALID_ENUM error is generated if *type* is UNSIGNED_INT_10F_11F_11F_REV.

An individual array is enabled or disabled by calling one of

void **EnableClientState**(enum *array*);
void **DisableClientState**(enum *array*);

with *array* set to VERTEX_ARRAY, NORMAL_ARRAY, COLOR_ARRAY, SECONDARY_COLOR_ARRAY, INDEX_ARRAY, EDGE_FLAG_ARRAY, FOG_COORD_ARRAY, or TEXTURE_COORD_ARRAY, for the vertex, normal, color, secondary color, color index, edge flag, fog coordinate, or texture coordinate array, respectively.

Errors

An INVALID_VALUE error is generated if *array* is not one of the values listed above describing a fixed-function attribute array.

The command

void **ClientActiveTexture**(enum *texture*);

is used to select the vertex array client state parameters to be modified by the **TexCoordPointer** command and the array affected by **EnableClientState** and **DisableClientState** with parameter TEXTURE_COORD_ARRAY. This command sets the client state variable CLIENT_ACTIVE_TEXTURE. Each texture coordinate set has a client state vector which is selected when this command is invoked. This state vector includes the vertex array state. This call also selects the texture coordinate set state used for queries of client state.

Errors

An INVALID_ENUM error is generated if *texture* is not one of the tokens TEXTUREi, where i is in the range zero to the value of MAX_TEXTURE_COORDS (the implementation-dependent number of texture coordinate sets) minus one.

10.3.4 Vertex Attribute Divisors

Each generic vertex attribute has a corresponding *divisor* which modifies the rate at which attributes advance, which is useful when rendering multiple instances of primitives in a single draw call. If the *divisor* is zero, the corresponding attributes advance once per vertex. Otherwise, attributes advance once per *divisor* instances of the set(s) of vertices being rendered. A generic attribute is referred to as *instanced* if its corresponding *divisor* value is non-zero.

The divisor value for attributes taken from a vertex array object is set with the commands

```
void VertexBindingDivisor( uint bindingindex,
    uint divisor );
void VertexArrayBindingDivisor( uint vaobj,
    uint bindingindex, uint divisor );
```

For **VertexBindingDivisor**, the vertex array object is the currently bound vertex array object. For **VertexArrayBindingDivisor**, *vaobj* is zero, indicating the default vertex array object, or the name of the vertex array object. These commands set the divisor for the buffer bound to the specified *bindingindex* of the vertex array object to *divisor*.

Errors

An `INVALID_OPERATION` error is generated by **VertexArrayBindingDivisor** if *vaobj* is not zero or the name of an existing vertex array object.

An `INVALID_VALUE` error is generated if *bindingindex* is greater than or equal to the value of `MAX_VERTEX_ATTRIB_BINDINGS`.

The command

```
void VertexAttribDivisor( uint index, uint divisor );
```

is equivalent to (assuming no errors are generated):

```
VertexAttribBinding(index, index);
VertexBindingDivisor(index, divisor);
```

Errors

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

10.3.5 Transferring Array Elements

When a vertex is transferred to the GL by **DrawArrays**, **DrawElements**, or the other **Draw*** commands described below, each generic attribute is expanded to four components. If *size* is one then the x component of the attribute is specified by the array; the y, z, and w components are implicitly set to 0, 0, and 1, respectively. If *size* is two then the x and y components of the attribute are specified by the array; the z and w components are implicitly set to 0 and 1, respectively. If *size* is three then x, y, and z are specified, and w is implicitly set to 1. If *size* is four then all components are specified.

10.3.6 Primitive Restart

Primitive restart is enabled or disabled by calling one of the commands

`void` **Enable**(`enum` *target*);

and

`void` **Disable**(`enum` *target*);

with *target* PRIMITIVE_RESTART. The command

`void` **PrimitiveRestartIndex**(`uint` *index*);

specifies the index of a vertex array element that is treated specially when primitive restart is enabled. This value is called the *primitive restart index*.

When one of the ***DrawElements*** commands transfers a set of generic attribute array elements to the GL, if the index within the vertex arrays corresponding to that set is equal to the primitive restart index, then the GL does not process those elements as a vertex. Instead, it is as if the drawing command ended with the immediately preceding transfer, and another drawing command is immediately started with the same parameters, but only transferring the immediately following element through the end of the originally specified elements.

When one of the ***BaseVertex** drawing commands specified in section 10.4 is used, the primitive restart comparison occurs before the *basevertex* offset is added to the array index.

Primitive restart also affects drawing between **Begin** and **End** as described in section 10.7.4.

Primitive restart can also be enabled or disabled with a *target* of PRIMITIVE_RESTART_FIXED_INDEX. In this case, the primitive restart index is equal to $2^N - 1$, where N is 8, 16 or 32 if the type is UNSIGNED_BYTE, UNSIGNED_SHORT, or UNSIGNED_INT, respectively, and the *index* value specified by **PrimitiveRestartIndex** is ignored.

If both PRIMITIVE_RESTART and PRIMITIVE_RESTART_FIXED_INDEX are enabled, the *index* value determined by PRIMITIVE_RESTART_FIXED_INDEX is used.

Note that primitive restart is not performed for array elements transferred by any drawing command not taking a *type* parameter, including **ArrayElement** and all of the ***Draw*** commands other than ***DrawElements***.

Implementations are not required to support primitive restart for separate patch primitives (primitive type PATCHES). Support can be queried by calling **GetBooleanv** with *pname* PRIMITIVE_RESTART_FOR_PATCHES_SUPPORTED. A value of FALSE indicates that primitive restart is treated as disabled when drawing patches, no matter the value of the enables. A value of TRUE indicates that primitive restart behaves normally for patches.

10.3.7 Robust Buffer Access

Robust buffer access is enabled by creating a context with robust access enabled through the window system binding APIs. When enabled, indices within the element array (see section 10.3.10) that reference vertex data that lies outside the enabled attribute's vertex buffer object result in reading zero. It is not possible to read vertex data from outside the enabled vertex buffer objects or from another GL context, and these accesses do not result in abnormal program termination.

10.3.8 Packed Vertex Data Formats

Vertex data formats UNSIGNED_INT_2_10_10_10_REV and INT_2_10_10_10_REV describe packed, 4 component formats stored in a single 32-bit word.

For UNSIGNED_INT_2_10_10_10_REV, the first (x), second (y), and third (z) components are represented as 10-bit unsigned integer values and the fourth (w) component is represented as a 2-bit unsigned integer value.

For INT_2_10_10_10_REV, the x, y and z components are represented as 10-bit signed two's complement integer values and the w component is represented as a 2-bit signed two's complement integer value.

The *normalized* value is used to indicate whether to normalize the data to $[0, 1]$ (for unsigned types) or $[-1, 1]$ (for signed types). During normalization, the conversion rules specified in equations 2.1 and 2.2 are followed.

Tables 10.5 and 10.6 describe how these components are laid out in a 32-bit word.

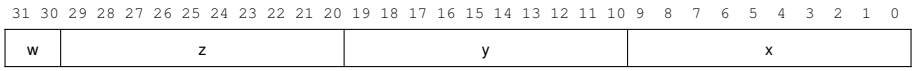


Table 10.5: Packed component layout for non-BGRA formats. Bit numbers are indicated for each component.

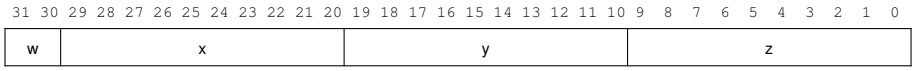


Table 10.6: Packed component layout for BGRA format. Bit numbers are indicated for each component.

Vertex data format UNSIGNED_INT_10F_11F_11F_REV describes a packed, 3-component format that is stored in a single 32-bit word. The first (x), and second (y) components are represented as 11-bit unsigned floating-point values, and the third (z) component is represented as a 10-bit unsigned floating-point value. Table 10.7 describes how these components are laid out in a 32-bit word.

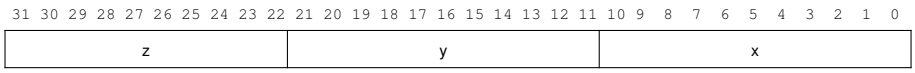


Table 10.7: Packed component layout for UNSIGNED_INT_10F_11F_11F_REV format. Bit numbers are indicated for each component.

10.3.9 Vertex Arrays in Buffer Objects

Blocks of vertex array data may be stored in buffer objects with the same format and layout options described in section 10.3. GL implementations are expected to (at minimum) be optimized for data with all components represented as floats, as well as for color data with components represented as either floats or unsigned bytes.

A buffer object binding point is added to the client state associated with each vertex array type and index. The commands that specify the locations and orga-

nizations of vertex arrays copy the buffer object name that is bound to ARRAY_BUFFER to the binding point corresponding to the vertex array type or index being specified. For example, the **VertexAttribPointer** command copies the value of ARRAY_BUFFER_BINDING (the queriable name of the buffer binding corresponding to the target ARRAY_BUFFER) to the client state variable VERTEX_ATTRIB_ARRAY_BUFFER_BINDING for the specified *index*.

The drawing commands using vertex arrays described in section 10.4 and section 10.7 operate as previously defined, except that data for enabled vertex and attrib arrays are sourced from buffers if the array's buffer binding is non-zero.

When an array is sourced from a buffer object for a vertex attribute, the *bindingindex* set with **VertexAttribBinding** for that attribute indicates which vertex buffer binding is used. The sum of the *relativeoffset* set for the attribute with **VertexAttrib*Format** and the *offset* set for the vertex buffer with **BindVertexBuffer** is used as the offset in basic machine units of the first element in that buffer's data store.

When a generic attribute array is sourced from client memory, the vertex attribute binding state is ignored. Instead, the *pointer*, *size*, *type*, and *stride* parameters set with **VertexAttrib*Pointer** for that attribute indicate the location in client memory of attribute values and their size, type, and stride, respectively.

It is acceptable for vertex or attribute arrays to be sourced from any combination of client memory and various buffer objects during a single rendering operation.

10.3.10 Array Indices in Buffer Objects

Blocks of array indices may be stored in buffer objects with the same format options that are supported for client-side index arrays. Initially zero is bound to ELEMENT_ARRAY_BUFFER, indicating that **DrawElements**, **DrawRangeElements**, and **DrawElementsInstanced** are to source their indices from arrays passed as their *indices* parameters, and that **MultiDrawElements** is to source its indices from the array of pointers to arrays passed in as its *indices* parameter.

A buffer object is bound to ELEMENT_ARRAY_BUFFER by calling **BindBuffer** with *target* set to ELEMENT_ARRAY_BUFFER, and *buffer* set to the name of the buffer object. If no corresponding buffer object exists, one is initialized as defined in section 6.

While a non-zero buffer object name is bound to ELEMENT_ARRAY_BUFFER, **DrawElements**, **DrawRangeElements**, and **DrawElementsInstanced** source their indices from that buffer object, using their *indices* parameters as offsets into the buffer object in the same fashion as described in section 10.3.9. **DrawElementsBaseVertex**, **DrawRangeElementsBaseVertex**, and **DrawEle-**

Indirect Command Name	Indirect Buffer *target*
DrawArraysIndirect	DRAW_INDIRECT_BUFFER
DrawElementsIndirect	DRAW_INDIRECT_BUFFER
MultiDrawArraysIndirect	DRAW_INDIRECT_BUFFER
MultiDrawElementsIndirect	DRAW_INDIRECT_BUFFER
DispatchComputeIndirect	DISPATCH_INDIRECT_BUFFER

Table 10.8: Indirect commands and corresponding indirect buffer targets.

mentsInstancedBaseVertex also source their indices from that buffer object, adding the *basevertex* offset to the appropriate vertex index as a final step before indexing into the vertex buffer; this does not affect the calculation of the base pointer for the index array. Finally, **MultiDrawElements** and **MultiDrawElementsBaseVertex** also source their indices from that buffer object, using its *indices* parameter as a pointer to an array of pointers that represent offsets into the buffer object.

In some cases performance will be optimized by storing indices and array data in separate buffer objects, and by creating those buffer objects with the corresponding binding points.

10.3.11 Indirect Commands in Buffer Objects

Arguments to the *indirect commands* **DrawArraysIndirect**, **DrawElementsIndirect**, **MultiDrawArraysIndirect**, and **MultiDrawElementsIndirect** (see section 10.4), and to **DispatchComputeIndirect** (see section 19) may be sourced from the buffer object currently bound to the corresponding indirect buffer *target* (see table 10.8), using the command's *indirect* parameter as an offset into the buffer object in the same fashion as described in section 10.3.9. Buffer objects are created and/or bound to a *target* as described in section 6.1. Initially zero is bound to each *target*.

Arguments are stored in buffer objects as structures (for ***Draw*Indirect**) or arrays (for **DispatchComputeIndirect**) of tightly packed 32-bit integers.

If zero is bound to DRAW_INDIRECT_BUFFER, the corresponding ***Draw*Indirect** commands instead source their arguments directly from the *indirect* pointer in client memory.

10.4 Drawing Commands Using Vertex Arrays

The command

```
void DrawArraysOneInstance( enum mode, int first,
    sizei count, int instance, uint baseinstance );
```

does not exist in the GL, but is used to describe functionality in the rest of this section. This command constructs a sequence of geometric primitives by successively transferring elements for *count* vertices. Elements *first* through $first + count - 1$ of each enabled non-instanced array are transferred to the GL. If *count* is zero, no elements are transferred.

mode specifies what kind of primitives are constructed, and must be one of the primitive types defined in section 10.1.

If an enabled vertex attribute array is instanced (it has a non-zero *divisor* as specified by **VertexAttribDivisor**), the element index that is transferred to the GL, for all vertices, is given by

$$\left\lfloor \frac{instance}{divisor} \right\rfloor + baseinstance$$

If an array corresponding to an attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current attribute state (see section 10.2).

If an array is enabled, the corresponding current vertex attribute value is unaffected by the execution of **DrawArraysOneInstance**.

The index of any element transferred to the GL by **DrawArraysOneInstance** is referred to as its *vertex ID*, and may be read by a vertex shader as `gl_VertexID`. The vertex ID of the ith element transferred is *first* + i.

The value of *instance* may be read by a vertex shader as `gl_InstanceID`, as described in section 11.1.3.9.

Errors

An `INVALID_ENUM` error is generated if *mode* is not one of the primitive types defined in section 10.1.

Specifying $first < 0$ results in undefined behavior. Generating an `INVALID_VALUE` error is recommended in this case.

An `INVALID_VALUE` error is generated if *count* is negative.

The command

```
void DrawArrays( enum mode, int first, sizei count );
```

is equivalent to

DrawArraysOneInstance(*mode*, *first*, *count*, 0, 0);

The command

```
void DrawArraysInstancedBaseInstance( enum mode,
   int first, sizei count, sizei instancecount,
   uint baseinstance );
```

behaves identically to **DrawArrays** except that *instancecount* instances of the range of elements are executed and the value of *instance* advances for each iteration. Those attributes that have non-zero values for *divisor*, as specified by **VertexAttribDivisor**, advance once every *divisor* instances. Additionally, the first element within those instanced vertex attributes is specified in *baseinstance*.

DrawArraysInstancedBaseInstance is equivalent (assuming no errors are generated) to:

```
if (mode, count, or instancecount is invalid)
   generate appropriate error
else {
   for (i = 0; i < instancecount; i++) {
      DrawArraysOneInstance(mode, first, count, i,
         baseinstance);
   }
}
```

The command

```
void DrawArraysInstanced( enum mode, int first,
   sizei count, sizei instancecount );
```

is equivalent to

DrawArraysInstancedBaseInstance(*mode*, *first*, *count*, *instancecount*, 0);

The command

```
void DrawArraysIndirect( enum mode, const
   void *indirect );
```

is equivalent to

```
typedef struct {
    uint count;
    uint instanceCount;
    uint first;
    uint baseInstance;
} DrawArraysIndirectCommand;

DrawArraysIndirectCommand *cmd =
    (DrawArraysIndirectCommand *)indirect;
DrawArraysInstancedBaseInstance(mode, cmd->first, cmd->count,
    cmd->instanceCount, cmd->baseInstance);
```

As with **DrawArraysInstanced**, vertex attributes may be sourced from client arrays or vertex buffer objects. Unlike **DrawArraysInstanced**, *first* is unsigned and cannot cause an error.

Errors

An `INVALID_OPERATION` error is generated if the command would source data beyond the end of the buffer object.

An `INVALID_VALUE` error is generated if *indirect* is not a multiple of the size, in basic machine units, of `uint`.

All elements of `DrawArraysIndirectCommand` are tightly packed 32-bit values.

The command

```
void MultiDrawArrays( enum mode, const int *first,
    const sizei *count, sizei drawcount );
```

behaves identically to **DrawArrays** except that *drawcount* separate ranges of elements are specified instead, all elements are treated as though they are not instanced, and the value of *instance* remains zero. It is equivalent (assuming no errors are generated) to:

```
if (mode or drawcount is invalid)
    generate appropriate error
else {
    for (i = 0; i < drawcount; i++) {
        if (count[i] > 0)
            DrawArraysOneInstance(mode, first[i], count[i],
```

```
                0, 0);
        }
    }
```

The index of the draw (*i* in the above pseudo-code) may be read by a vertex shader as `gl_DrawID`, as described in section 11.1.3.9.

The command

`void` **MultiDrawArraysIndirect**(`enum` *mode*, `const` `void` **indirect*, `sizei` *drawcount*, `sizei` *stride*);

behaves identically to **DrawArraysIndirect** except that *indirect* is treated as an array of *drawcount* `DrawArraysIndirectCommand` structures. *indirect* contains the offset of the first element of the array within the buffer currently bound to the `DRAW_INDIRECT` buffer binding. *stride* specifies the distance, in basic machine units, between the elements of the array. If *stride* is zero, the array elements are treated as tightly packed.

It is equivalent (assuming no errors are generated) to:

```
if (mode is invalid)
    generate appropriate error
else {
    const ubyte *ptr = (const ubyte *)indirect;
    for (i = 0; i < drawcount; i++) {
        DrawArraysIndirect(mode, (DrawArraysIndirectCommand*)ptr);
        if (stride == 0) {
            ptr += sizeof(DrawArraysIndirectCommand);
        } else {
            ptr += stride;
        }
    }
}
```

The index of the draw (*i* in the above pseudo-code) may be read by a vertex shader as `gl_DrawID`, as described in section 11.1.3.9.

Errors

In addition to errors that would be generated by **DrawArraysIndirect**:
An `INVALID_VALUE` error is generated if *stride* is negative.
An `INVALID_VALUE` error is generated if *stride* is neither zero nor a mul-

tiple of four.

An INVALID_VALUE error is generated if *drawcount* is not positive.

The command

```
void MultiDrawArraysIndirectCount( enum mode, const
    void *indirect, intptr drawcount, intptr maxdrawcount,
    sizei stride );
```

behaves similarly to **MultiDrawArraysIndirect**, except that *drawcount* defines an offset (in bytes) into the buffer object bound to the PARAMETER_BUFFER binding point at which a single *sizei* typed value is stored, which contains the draw count. *maxdrawcount* specifies the maximum number of draws that are expected to be stored in the buffer. If the value stored at *drawcount* into the buffer is greater than *maxdrawcount*, the implementation stops processing draws after *maxdrawcount* parameter sets. *drawcount* must be a multiple of four.

Errors

In addition to errors that would be generated by **MultiDrawArraysIndirect**:

An INVALID_OPERATION error is generated if no buffer is bound to the PARAMETER_BUFFER binding point.

An INVALID_VALUE error is generated if *drawcount* is not a multiple of four.

An INVALID_OPERATION error is generated if reading a *sizei* typed value from the buffer bound to the PARAMETER_BUFFER target at the offset specified by *drawcount* would result in an out-of-bounds access.

The command

```
void DrawElementsOneInstance( enum mode, sizei count,
    enum type, const void *indices, int instance,
    int basevertex, uint baseinstance );
```

does not exist in the GL, but is used to describe functionality in the rest of this section. This command constructs a sequence of geometric primitives by successively transferring elements for *count* vertices to the GL.

The index of any element transferred to the GL by **DrawElementsOneInstance** is referred to as its *vertex ID*, and may be read by a vertex shader as gl_VertexID. If no element array buffer is bound, the vertex ID of the ith element

transferred is $indices[i] + basevertex$. Otherwise, the vertex ID of the ith element transferred is the sum of $basevertex$ and the value stored in the currently bound element array buffer at offset $indices + i \times sizeof(type)$. If the vertex ID is larger than the maximum value representable by *type*, it should behave as if the calculation were upconverted to 32-bit unsigned integers (with wrapping on overflow conditions). Behavior of **DrawElementsOneInstance** is undefined if the vertex ID is negative for any element, and should be handled as described in section 6.4.

type must be one of `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, or `UNSIGNED_INT`, indicating that the index values are of GL type `ubyte`, `ushort`, or `uint` respectively. *mode* specifies what kind of primitives are constructed, and must be one of the primitive types defined in section 10.1.

If an enabled vertex attribute array is instanced (it has a non-zero *divisor* as specified by **VertexAttribDivisor**), the element index that is transferred to the GL, for all vertices, is given by

$$\left\lfloor \frac{instance}{divisor} \right\rfloor + baseinstance$$

If an array corresponding to an attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current attribute state (see section 10.2).

GL implementations do not restrict index values; any value representable in a `uint` may be used. However, for compatibility with OpenGL ES implementations, the maximum representable index vaue may be queried by calling **GetInteger64v** with *pname* `MAX_ELEMENT_INDEX`, and will return $2^{32} - 1$.

If an array is enabled, the corresponding current vertex attribute value is unaffected by the execution of **DrawElementsOneInstance**.

The value of *instance* may be read by a vertex shader as `gl_InstanceID`, as described in section 11.1.3.9.

Errors

An `INVALID_ENUM` error is generated if *mode* is not one of the primitive types defined in section 10.1.

An `INVALID_VALUE` error is generated if *count* is negative.

An `INVALID_ENUM` error is generated if *type* is not `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, or `UNSIGNED_INT`.

The command

```
void DrawElements( enum mode, sizei count, enum type,
    const void *indices );
```

behaves identically to **DrawElementsOneInstance** with *instance*, *basevertex* and *baseinstance* set to zero; the effect of calling

DrawElements(*mode*, *count*, *type*, *indices*);

is equivalent to

```
if (mode, count or type is invalid)
    generate appropriate error
else
    DrawElementsOneInstance(mode, count, type, indices,
        0, 0, 0);
```

The command

```
void DrawElementsInstancedBaseInstance( enum mode,
    sizei count, enum type, const void *indices,
    sizei instancecount, uint baseinstance );
```

behaves identically to **DrawElements** except that *instancecount* instances of the set of elements are executed and the value of *instance* advances between each set. Instanced attributes are advanced as they do during execution of **DrawArraysInstancedBaseInstance**, and *baseinstance* has the same effect. It is equivalent (assuming no errors are generated) to:

```
if (mode, count, type, or instancecount is invalid)
    generate appropriate error
else {
    for (int i = 0; i < instancecount; i++) {
        DrawElementsOneInstance(mode, count, type, indices,
            i, 0, baseinstance);
    }
}
```

The command

```
void DrawElementsInstanced( enum mode, sizei count,
    enum type, const void *indices, sizei instancecount );
```

behaves identically to **DrawElementsInstancedBaseInstance** except that *baseinstance* is zero. It is equivalent to

DrawElementsInstancedBaseInstance(*mode*, *count*, *type*, *indices*, *instancecount*, 0);

The command

```
void MultiDrawElements( enum mode, const
    sizei *count, enum type, const void *
    const *indices, sizei drawcount );
```

behaves identically to **DrawElementsInstanced** except that *drawcount* separate sets of elements are specified instead, all elements are treated as though they are not instanced, and the value of *instance* remains zero. It is equivalent (assuming no errors are generated) to:

```
if (mode, drawcount, or type is invalid)
    generate appropriate error
else {
    for (int i = 0; i < drawcount; i++)
        DrawElementsOneInstance(mode, count[i], type,
            indices[i], 0, 0, 0);
}
```

The index of the draw (*i* in the above pseudo-code) may be read by a vertex shader as `gl_DrawID`, as described in section 11.1.3.9.

The command

```
void DrawRangeElements( enum mode, uint start,
    uint end, sizei count, enum type, const
    void *indices );
```

is a restricted form of **DrawElements**. *mode*, *count*, *type*, and *indices* match the corresponding arguments to **DrawElements**, with the additional constraint that all index values identified by *indices* must lie between *start* and *end* inclusive.

Implementations denote recommended maximum amounts of vertex and index data, which may be queried by calling **GetIntegerv** with *pname*s `MAX_ELEMENTS_VERTICES` and `MAX_ELEMENTS_INDICES`. If $end - start + 1$ is greater than the value of `MAX_ELEMENTS_VERTICES`, or if *count* is greater than the value of `MAX_ELEMENTS_INDICES`, then the call may operate at reduced performance. There is no requirement that all vertices in the range $[start, end]$ be referenced. However, the implementation may partially process unused vertices, reducing performance from what could be achieved with an optimal index set.

Errors

An `INVALID_VALUE` error is generated if $end < start$.

An `INVALID_VALUE` error is generated if *count* is negative.

Invalid *mode*, *count*, or *type* parameters generate the same errors as would the corresponding call to **DrawElements**.

It is an error for index values (other than the primitive restart index, when primitive restart is enabled) to lie outside the range $[start, end]$, but implementations are not required to check for this. Such indices will cause implementation-dependent behavior.

The commands

```
void DrawElementsBaseVertex( enum mode, sizei count,
    enum type, const void *indices, int basevertex );
void DrawRangeElementsBaseVertex( enum mode,
    uint start, uint end, sizei count, enum type, const
    void *indices, int basevertex );
void DrawElementsInstancedBaseVertex( enum mode,
    sizei count, enum type, const void *indices,
    sizei instancecount, int basevertex );
void DrawElementsInstancedBaseVertexBaseInstance(
    enum mode, sizei count, enum type, const
    void *indices, sizei instancecount, int basevertex,
    uint baseinstance );
```

are equivalent to the commands with the same base name (without the **BaseVertex** suffix), except that the *basevertex* value passed to **DrawElementsOneInstance** is the *basevertex* value of these commands, instead of zero.

For **DrawRangeElementsBaseVertex**, the values taken from *indices* for each element transferred must be in the range $[start, end]$, prior to adding the *basevertex* offset. Index values lying outside this range are treated in the same way as **DrawRangeElements**.

The command

```
void DrawElementsIndirect( enum mode, enum type, const
    void *indirect );
```

is equivalent to

```
typedef struct {
```

```
    uint count;
    uint instanceCount;
    uint firstIndex;
    int baseVertex;
    uint baseInstance;
} DrawElementsIndirectCommand;

if (no element array buffer is bound) {
    generate appropriate error
} else {
    DrawElementsIndirectCommand *cmd =
(DrawElementsIndirectCommand *)indirect;

    DrawElementsInstancedBaseVertexBaseInstance(mode,
        cmd->count, type,
        cmd->firstIndex * size-of-type,
        cmd->instanceCount, cmd->baseVertex,
        cmd->baseInstance);
}
```

As with **DrawElementsInstancedBaseVertex**, vertex attributes may be sourced from client arrays or vertex buffer objects. Unlike **DrawElementsInstancedBaseVertex**, indices may not come from a client array and must come from an index buffer.

Errors

An `INVALID_OPERATION` error is generated if no element array buffer is bound.

An `INVALID_OPERATION` error is generated if the command would source data beyond the end of the buffer object.

An `INVALID_VALUE` error is generated if *indirect* is not a multiple of the size, in basic machine units, of `uint`.

All elements of `DrawElementsIndirectCommand` are tightly packed. The command

```
void MultiDrawElementsIndirect( enum mode, enum type,
    const void *indirect, sizei drawcount, sizei stride );
```

behaves identically to **DrawElementsIndirect** except that *indirect* is treated as an array of *drawcount* `DrawElementsIndirectCommand` structures. *indirect* con-

tains the offset of the first element of the array within the buffer currently bound to the DRAW_INDIRECT buffer binding. *stride* specifies the distance, in basic machine units, between the elements of the array. If *stride* is zero, the array elements are treated as tightly packed.

It is equivalent (assuming no errors are generated) to:

```
if (mode or type is invalid)
    generate appropriate error
else {
    const ubyte *ptr = (const ubyte *)indirect;
    for (i = 0; i < drawcount; i++) {
        DrawElementsIndirect(mode, type,
            (DrawElementsIndirectCommand*)ptr);
        if (stride == 0) {
            ptr += sizeof(DrawElementsIndirectCommand);
        } else {
            ptr += stride;
        }
    }
}
```

The index of the draw (*i* in the above pseudo-code) may be read by a vertex shader as gl_DrawID, as described in section 11.1.3.9.

Errors

In addition to errors that would be generated by **DrawElementsIndirect**:
An INVALID_VALUE error is generated if *stride* is negative.
An INVALID_VALUE error is generated if *stride* is neither zero nor a multiple of four.
An INVALID_VALUE error is generated if *drawcount* is not positive.

The command

```
void MultiDrawElementsIndirectCount( enum mode,
    enum type, const void *indirect, intptr drawcount,
    sizei maxdrawcount, sizei stride );
```

behaves similarly to **MultiDrawElementsIndirect**, except that *drawcount* defines an offset (in bytes) into the buffer object bound to the PARAMETER_BUFFER binding point at which a single *sizei* typed value is stored, which contains the draw

count. *maxdrawcount* specifies the maximum number of draws that are expected to be stored in the buffer. If the value stored at *drawcount* into the buffer is greater than *maxdrawcount*, the implementation stops processing draws after *maxdrawcount* parameter sets. *drawcount* must be a multiple of four.

Errors

In addition to errors that would be generated by **MultiDrawElementsIndirect**:

An `INVALID_OPERATION` error is generated if no buffer is bound to the `PARAMETER_BUFFER` binding point.

An `INVALID_VALUE` error is generated if *drawcount* is not a multiple of four.

An `INVALID_VALUE` error is generated if *maxdrawcount* is negative.

An `INVALID_OPERATION` error is generated if reading a *sizei* typed value from the buffer bound to the `PARAMETER_BUFFER` target at the offset specified by *drawcount* would result in an out-of-bounds access.

The command

```
void MultiDrawElementsBaseVertex( enum mode, const
    sizei *count, enum type, const void *
    const *indices, sizei drawcount, const
    int *basevertex );
```

behaves identically to **DrawElementsBaseVertex**, except that *drawcount* separate lists of elements are specified instead. It is equivalent (assuming no errors are generated) to:

```
if (mode or drawcount is invalid)
    generate appropriate error
else {
    for (int i = 0; i < drawcount; i++)
        if (count[i] > 0)
            DrawElementsBaseVertex(mode, count[i], type,
                indices[i], basevertex[i]);
}
```

The index of the draw (*i* in the above pseudo-code) may be read by a vertex shader as `gl_DrawID`, as described in section 11.1.3.9.

Errors

In addition to errors that would be generated by **DrawElementsBaseVertex**:

An `INVALID_VALUE` error is generated if *drawcount* is negative.

10.4.1 Interleaved Arrays

The command

```
void InterleavedArrays( enum format, sizei stride, const
    void *pointer );
```

efficiently initializes the six arrays and their enables to one of 14 configurations. *format* must be one of the formats in table 10.9.

The effect of

InterleavedArrays$(format,\ stride,\ pointer)$;

is the same as the effect of the command sequence

```
if (format or stride is invalid)
    generate appropriate error
else {
    set e_t, e_c, e_n, s_t, s_c, s_v, t_c, p_c, p_n, p_v, and s as a function
        of table 10.9 and the value of format.
    if (stride != 0)
        effectiveStride = stride;
    else
        effectiveStride = s;
    DisableClientState(EDGE_FLAG_ARRAY);
    DisableClientState(INDEX_ARRAY);
    DisableClientState(SECONDARY_COLOR_ARRAY);
    DisableClientState(FOG_COORD_ARRAY);
    if (e_t) {
        EnableClientState(TEXTURE_COORD_ARRAY);
        TexCoordPointer(s_t, FLOAT, effectiveStride, pointer);
    } else
        DisableClientState(TEXTURE_COORD_ARRAY);
    if (e_c) {
        EnableClientState(COLOR_ARRAY);
```

format	e_t	e_c	e_n	s_t	s_c	s_v	t_c
V2F	*False*	*False*	*False*			2	
V3F	*False*	*False*	*False*			3	
C4UB_V2F	*False*	*True*	*False*		4	2	UNSIGNED_BYTE
C4UB_V3F	*False*	*True*	*False*		4	3	UNSIGNED_BYTE
C3F_V3F	*False*	*True*	*False*		3	3	FLOAT
N3F_V3F	*False*	*False*	*True*			3	
C4F_N3F_V3F	*False*	*True*	*True*		4	3	FLOAT
T2F_V3F	*True*	*False*	*False*	2		3	
T4F_V4F	*True*	*False*	*False*	4		4	
T2F_C4UB_V3F	*True*	*True*	*False*	2	4	3	UNSIGNED_BYTE
T2F_C3F_V3F	*True*	*True*	*False*	2	3	3	FLOAT
T2F_N3F_V3F	*True*	*False*	*True*	2		3	
T2F_C4F_N3F_V3F	*True*	*True*	*True*	2	4	3	FLOAT
T4F_C4F_N3F_V4F	*True*	*True*	*True*	4	4	4	FLOAT

format	p_c	p_n	p_v	s
V2F			0	$2f$
V3F			0	$3f$
C4UB_V2F	0		c	$c + 2f$
C4UB_V3F	0		c	$c + 3f$
C3F_V3F	0		$3f$	$6f$
N3F_V3F		0	$3f$	$6f$
C4F_N3F_V3F	0	$4f$	$7f$	$10f$
T2F_V3F			$2f$	$5f$
T4F_V4F			$4f$	$8f$
T2F_C4UB_V3F	$2f$		$c + 2f$	$c + 5f$
T2F_C3F_V3F	$2f$		$5f$	$8f$
T2F_N3F_V3F		$2f$	$5f$	$8f$
T2F_C4F_N3F_V3F	$2f$	$6f$	$9f$	$12f$
T4F_C4F_N3F_V4F	$4f$	$8f$	$11f$	$15f$

Table 10.9: Variables that direct the execution of **InterleavedArrays**. f is `sizeof(float)`. c is 4 times `sizeof(ubyte)`, rounded up to the nearest multiple of f. All pointer arithmetic is performed in units of `sizeof(ubyte)`.

```
        ColorPointer(s_c, t_c, effectiveStride, pointer + p_c);
    } else
        DisableClientState(COLOR_ARRAY);
    if (e_n) {
        EnableClientState(NORMAL_ARRAY);
        NormalPointer(FLOAT, effectiveStride, pointer + p_n);
    } else
        DisableClientState(NORMAL_ARRAY);
    EnableClientState(VERTEX_ARRAY);
    VertexPointer(s_v, FLOAT, effectiveStride, pointer + p_v);
}
```

Errors

An INVALID_ENUM error is generated if *format* is not one of the formats in table 10.9.

An INVALID_VALUE error is generated if *stride* is negative.

10.5 Vertex Array and Vertex Array Object Queries

To query parameters of a vertex array object, use the command

```
void GetVertexArrayiv( uint vaobj, enum pname,
    int *param );
```

vaobj is zero, indicating the default vertex array object, or the name of the vertex array object. The value of parameter *pname* of *vaobj* is returned in *param*. *pname* must be ELEMENT_ARRAY_BUFFER_BINDING.

Errors

An INVALID_OPERATION error is generated if *vaobj* is not zero or the name of an existing vertex array object.

An INVALID_ENUM error is generated if *pname* is not ELEMENT_ARRAY_BUFFER_BINDING.

To query parameters of an attribute of a vertex array object, use the commands

```
void GetVertexArrayIndexediv( uint vaobj, uint index,
    enum pname, int *param );
```

```
void GetVertexArrayIndexed64iv( uint vaobj, uint index,
    enum pname, int64 *param );
```

vaobj is zero, indicating the default vertex array object, or the name of the vertex array object. The value of parameter *pname* for attribute *index* of *vaobj* is returned in *param*.

For **GetVertexArrayIndexediv**, *pname* must be one of VERTEX_ATTRIB_ARRAY_ENABLED, VERTEX_ATTRIB_ARRAY_SIZE, VERTEX_ATTRIB_ARRAY_STRIDE, VERTEX_ATTRIB_ARRAY_TYPE, VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER, VERTEX_ATTRIB_ARRAY_LONG, VERTEX_ATTRIB_ARRAY_DIVISOR, VERTEX_ATTRIB_RELATIVE_OFFSET, VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, VERTEX_BINDING_STRIDE, VERTEX_BINDING_DIVISOR, or VERTEX_BINDING_BUFFER.

For **GetVertexArrayIndexed64iv**, *pname* must be VERTEX_BINDING_OFFSET.

Errors

An INVALID_OPERATION error is generated if *vaobj* is not zero or the name of an existing vertex array object.

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIBS, and *pname* is one of VERTEX_ATTRIB_ARRAY_ENABLED, VERTEX_ATTRIB_ARRAY_SIZE, VERTEX_ATTRIB_ARRAY_STRIDE, VERTEX_ATTRIB_ARRAY_TYPE, VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER, VERTEX_ATTRIB_ARRAY_LONG, VERTEX_ATTRIB_ARRAY_DIVISOR, VERTEX_ATTRIB_RELATIVE_OFFSET, or VERTEX_ATTRIB_ARRAY_BUFFER_BINDING

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIB_BINDINGS, and *pname* is one of VERTEX_BINDING_OFFSET, VERTEX_BINDING_STRIDE, VERTEX_BINDING_DIVISOR, or VERTEX_BINDING_BUFFER.

An INVALID_ENUM error is generated if *pname* is not one of the valid values listed above for the corresponding command.

Queries of vertex array state variables are qualified by the value of VERTEX_ARRAY_BINDING to determine which vertex array object is queried. Tables 23.4-23.7 define the set of state stored in a vertex array object.

To query parameters of an attribute of the currently bound vertex array object, or current attribute values, use the commands

```
void GetVertexAttribdv( uint index, enum pname,
    double *params );
void GetVertexAttribfv( uint index, enum pname,
    float *params );
void GetVertexAttribiv( uint index, enum pname,
    int *params );
void GetVertexAttribIiv( uint index, enum pname,
    int *params );
void GetVertexAttribIuiv( uint index, enum pname,
    uint *params );
void GetVertexAttribLdv( uint index, enum pname,
    double *params );
```

The value of parameter *pname* for the attribute numbered *index* of the currently bound vertex array object is returned in *params*.

pname must be one of VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, VERTEX_ATTRIB_ARRAY_ENABLED, VERTEX_ATTRIB_ARRAY_SIZE, VERTEX_ATTRIB_ARRAY_STRIDE, VERTEX_ATTRIB_ARRAY_TYPE, VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER, VERTEX_ATTRIB_ARRAY_LONG, VERTEX_ATTRIB_ARRAY_DIVISOR, VERTEX_ATTRIB_BINDING, VERTEX_ATTRIB_RELATIVE_OFFSET, or CURRENT_VERTEX_ATTRIB. Note that all the queries except CURRENT_VERTEX_ATTRIB return values stored in the currently bound vertex array object (the value of VERTEX_ARRAY_BINDING). If the zero object is bound, these values are client state.

Queries of VERTEX_ATTRIB_ARRAY_BUFFER_BINDING and VERTEX_ATTRIB_ARRAY_DIVISOR map the requested attribute index to a binding index via the VERTEX_ATTRIB_BINDING state, and then return the value of VERTEX_BINDING_BUFFER or VERTEX_BINDING_DIVISOR, respectively.

All but CURRENT_VERTEX_ATTRIB return information about generic vertex attribute arrays. The enable state of a generic vertex attribute array is set by the command **EnableVertexAttribArray** and cleared by **DisableVertexAttribArray**. The size, stride, type, normalized flag, and unconverted integer flag are set by the commands **VertexAttribPointer** and **VertexAttribIPointer**. The normalized flag is always set to FALSE by **VertexAttribIPointer**. The unconverted integer flag is always set to FALSE by **VertexAttribPointer** and TRUE by **VertexAttribIPointer**.

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute *index*. **GetVertexAttribdv** and **GetVertexAttribfv** read and return the current attribute values as four floating-point values; **GetVertexAttribiv** reads them as floating-point values and converts them to four integer values;

GetVertexAttribIiv reads and returns them as four signed integers; **GetVertexAttribIuiv** reads and returns them as four unsigned integers; and **GetVertexAttribLdv** reads and returns them as four double-precision floating-point values. The results of the query are undefined if the current attribute values are read using one data type but were specified using a different one.

Errors

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

An INVALID_OPERATION error is generated if no vertex array object is bound (see section 10.3.1).

An INVALID_ENUM error is generated if *pname* is not one of the values listed above.

The command

```
void GetVertexAttribPointerv( uint index, enum pname,
    const void **pointer );
```

obtains the pointer named *pname* for the vertex attribute numbered *index* and places the information in the array *pointer*. *pname* must be VERTEX_ATTRIB_ARRAY_POINTER. The value returned is queried from the currently bound vertex array object. If the zero object is bound, the value is queried from client state.

Errors

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

An INVALID_OPERATION error is generated if no vertex array object is bound (see section 10.3.1).

Finally, the buffer bound to ELEMENT_ARRAY_BUFFER may be queried by calling **GetIntegerv** with *pname* ELEMENT_ARRAY_BUFFER_BINDING.

10.6 Required State

Let the number of supported generic vertex attributes (the value of MAX_VERTEX_ATTRIBS) be n. Let the number of supported texture units (the value of MAX_TEXTURE_COORDS) be m. Let the number of supported generic vertex attribute bindings (the value of MAX_VERTEX_ATTRIB_BINDINGS be k.

Then the state required to implement vertex arrays consists of $7+m+n$ boolean values, $7+m+n$ memory pointers, $7+m+n$ integer stride values, $6+m+n$ symbolic constants representing array types, $3+m+n$ integers representing values per element, $7+m$ unsigned integers representing vertex buffer bindings, n boolean values indicating normalization, n boolean values indicating whether the attribute values are pure integers, n boolean values indicating whether the attribute values are double precision, an integer for the client active texture unit selector, three integers for the current array buffer, current element array buffer, and current vertex array bindings, n unsigned integer vertex attribute binding indices, n unsigned integer relative offsets, k integers representing vertex attribute divisors, k unsigned integer vertex buffer bindings, k 64-bit integer vertex binding offsets, k integer vertex binding strides, an unsigned integer representing the primitive restart index, and two booleans representing the enable state of primitive restart and primitive restart with a fixed index.

In the initial state, the boolean values are each `FALSE`, the memory pointers are each `NULL`, the strides are each zero, the array types are each `FLOAT`, the integers representing values per element are each four, the normalized and pure integer flags are each `FALSE`, the divisors are each zero, the client active texture unit selector is `TEXTURE0`, the binding indices are i for each attribute i, the relative offsets are each zero, the vertex binding offsets are each zero, the vertex binding strides are each 16, the restart index is zero, and the restart enables are both `FALSE`.

10.7 Drawing Commands Using Begin and End

In addition to the drawing commands described in section 10.4, vertices defining a sequence of primitives may be transferred to the GL by enclosing commands defining attributes of those vertices between the two commands

```
void Begin( enum mode );
void End( void );
```

mode specifies the primitive type being defined and must be one of the primitive types defined in section 10.1. There is no limit on the number of vertices that may be specified between a **Begin** and an **End**.

The state required for **Begin** and **End** consists of a sixteen-valued integer indicating either one of the possible **Begin / End** *mode*s, or that no **Begin / End** mode is being processed.

Errors

An INVALID_ENUM error is generated by **Begin** if *mode* is not one of the primitive types defined in section 10.1.

An INVALID_FRAMEBUFFER_OPERATION error is generated by **Begin** if the framebuffer object bound to DRAW_FRAMEBUFFER_BINDING is not framebuffer complete (see section 9.4.2).

10.7.1 Transferring Vertices With Vertex Commands

A single vertex is specified between **Begin** and **End** by giving its coordinates in two, three, or four dimensions. This is done using one of several versions of the **Vertex** command:

```
void Vertex{234}{sifd}( T coords );
void Vertex{234}{sifd}v( const T coords );
```

Vertex coordinates may be stored as packed components within a larger natural type. Such data may be specified using

```
void VertexP{234}ui(enum type,uint coords);
void VertexP{234}uiv(enum type,const uint *coords);
```

These commands specify up to four coordinates as described above, packed into a single natural type as described in section 10.3.8. The *type* parameter must be INT_2_10_10_10_REV or UNSIGNED_INT_2_10_10_10_REV, specifying signed or unsigned data respectively. The first two (x, y), three (x, y, z), or four (x, y, z, w) components of the packed data are consumed by **VertexP2ui**, **VertexP3ui**, and **VertexP4ui**, respectively. For **VertexP*uiv**, *coords* contains the address of a single uint containing the packed coordinate components.

A call to any **Vertex** command specifies four coordinates: x, y, z, and w. The x coordinate is the first coordinate, y is second, z is third, and w is fourth. A call to **Vertex*2*** sets the x and y coordinates; the z coordinate is implicitly set to zero and the w coordinate to one. **Vertex*3*** sets x, y, and z to the provided values and w to one. **Vertex*4*** sets all four coordinates, allowing the specification of an arbitrary point in projective three-space.

Calling any of the **Vertex** commands above outside of a **Begin** / **End** pair results in undefined behavior.

10.7.2 Transferring Vertices With Vertex Attribute Zero

Setting generic vertex attribute zero using the **VertexAttrib** commands in section 10.2 also specifies a vertex and the value of its four coordinates; **Vertex2***, **Vertex3***, are **Vertex4*** commands are completely equivalent to the corresponding **VertexAttrib*** command with an *index* of zero.

10.7.3 Bundling Attributes With Vertex Commands

When one of the **Vertex** commands is called, values of generic attributes required by the vertex shader, or values of fixed-function attributes required by fixed-function vertex processing, are taken from the current values defined by the commands in section 10.2. When polygon edge flags are required, they are taken from the current edge flag defined by **EdgeFlag** (see section 10.1.17).

10.7.4 Transferring Vertices With ArrayElement

Vertices may also be transferred between **Begin** and **End** with attribute values specified in vertex arrays (see section 10.3). The command

```
void ArrayElement( int i );
```

transfers the *i*th element of every enabled, non-instanced array, and the first element of every enabled, instanced array to the GL. The effect of **ArrayElement** is the same as the effect of the command sequence

```
if (normal array enabled)
    Normal3[type]v(normal array element i);
if (color array enabled)
    Color[size][type]v(color array element i);
if (secondary color array enabled)
    SecondaryColor3[type]v(secondary color array element i);
if (fog coordinate array enabled)
    FogCoord[type]v(fog coordinate array element i);
for (j = 0; j < textureUnits; j++) {
    if (texture coordinate set j array enabled)
        MultiTexCoord[size][type]v(TEXTURE0 + j, texcoord(j, i));
}
if (color index array enabled)
    Index[type]v(color index array element i);
if (edge flag array enabled)
```

```
    EdgeFlagv(edge flag array element i);
for (j = 1; j < genericAttributes; j++) {
    if (generic vertex attribute j array enabled) {
        if (vertex attrib array divisor j > 0)
            k = 0;
        else
            k = i;
        VertexAttrib[size][type]v(j, genattrib(j, k));
    }
}
if (generic vertex attribute array 0 enabled) {
    if (vertex attrib array divisor 0 > 0)
        k = 0;
    else
        k = i;
    VertexAttrib[size][type]v(0, genattrib(0, k));
} else if (vertex array enabled) {
    Vertex[size][type]v(vertex array element i);
}
```

`genattrib(`*attrib*`, i)` represents the *i*th element of the vertex array for generic attribute *attrib*, and `texcoord(`*coord*`, i)` represents the *i*th element of the vertex array for texture coordinate set *coord*. *textureUnits* and *genericAttributes* give the number of texture coordinate sets and generic vertex attributes supported by the implementation, respectively. "[size]" and "[type]" correspond to the size and type of the corresponding array. For generic vertex attributes, it is assumed that a complete set of vertex attribute commands exists, even though not all such commands are provided by the GL.

When an array contains packed data, the pseudocode above will use the packed equivalent with the type of that data. For example, when a generic vertex attribute array contains packed data, the **VertexAttribP[size]uiv** command will be called instead of **VertexAttrib[size][type]v**.

Similarly, when a generic vertex attribute array contains pure integer data, **VertexAttribI[size][type]v** will be called; when an array contains fixed-point data, attribute values are specified in the signed two's complement 16.16 fixed-point `fixed` format; when an array contains double-precision data, **VertexAttrib[size][type]v** or **VertexAttribL[size][type]v** will be called, if the data was specified with **VertexAttribPointer** or **VertexAttribLPointer**, respectively; and when a generic attribute array normalization flag is set, and the array data type is not `FLOAT`, `HALF_FLOAT`, or `DOUBLE`, **VertexAttrib[size]N[type]v** will be called.

If **ArrayElement** is called while primitive restart is enabled (see section 10.3.6 and i is equal to the primitive restart index, then no vertex data is dereferenced, and no current vertex state is modified. Instead, it is as if **End** were called, followed by a call to **Begin** where *mode* is the same as the mode used by the previous **Begin**.

Changes made to array data between the execution of **Begin** and the corresponding execution of **End** may affect calls to **ArrayElement** that are made within the same **Begin / End** block in non-sequential ways. That is, a call to **ArrayElement** that precedes a change to array data may access the changed data, and a call that follows a change to array data may access original data.

Specifying $i < 0$ results in undefined behavior. Generating an `INVALID_VALUE` error is recommended in this case.

10.7.5 Commands Allowed Between Begin and End

The only GL commands that are allowed within any **Begin / End** pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, texture coordinates, generic vertex attributes, and fog coordinates (**Vertex**, **Color**, **SecondaryColor**, **Index**, **Normal**, **TexCoord** and **MultiTexCoord**, **VertexAttrib**, **FogCoord**), the **ArrayElement** command (see section 10.3), the **EvalCoord** and **EvalPoint** commands (see section 21.1), commands for specifying lighting material parameters (**Material** commands; see section 12.2.2), display list invocation commands (**CallList** and **CallLists**; see section 21.4), and the **EdgeFlag** command.

Errors

An `INVALID_OPERATION` error is generated if any other other GL command is called between the execution of **Begin** and the corresponding execution of **End**.

An `INVALID_OPERATION` error is generated if **Begin** is called after **Begin** has already been called, but before **End** is called.

An `INVALID_OPERATION` error is generated if **End** is called without a previous corresponding **Begin**.

Execution of the commands **EnableClientState**, **DisableClientState**, **PushClientAttrib**, **PopClientAttrib**, **ColorPointer**, **FogCoordPointer**, **EdgeFlagPointer**, **IndexPointer**, **NormalPointer**, **TexCoordPointer**, **SecondaryColorPointer**, **VertexPointer**, **VertexAttribPointer**, **ClientActiveTexture**, **InterleavedArrays**, and **PixelStore** is not allowed within any **Begin / End** pair, but an error may or may not be generated if such execution occurs. If an error is not

generated, GL operation is undefined. These commands are described in sections 10.3, 8.4.1, and chapter 22.

10.8 Rectangles

There is a set of GL commands to support efficient specification of rectangles as two corner vertices.

```
void Rect{sifd}( T x1, T y1, T x2, T y2 );
void Rect{sifd}v( const T v1[2], const T v2[2] );
```

Each command takes either four arguments organized as two consecutive pairs of (x, y) coordinates, or two pointers to arrays each of which contains an x value followed by a y value. The effect of the **Rect** command

Rect(x_1, y_1, x_2, y_2);

is exactly the same as the following sequence of commands:

```
Begin(POLYGON);
    Vertex2(x1, y1);
    Vertex2(x2, y1);
    Vertex2(x2, y2);
    Vertex2(x1, y2);
End();
```

The appropriate **Vertex2** command would be invoked depending on which of the **Rect** commands is issued.

10.9 Conditional Rendering

Conditional rendering can be used to discard rendering commands based on the result of an occlusion query. Conditional rendering is started and stopped using the commands

```
void BeginConditionalRender( uint id, enum mode );
void EndConditionalRender( void );
```

id specifies the name of an occlusion query object whose results are used to determine if the rendering commands are discarded. If the result (SAMPLES_PASSED) of the query is zero, or if the result (ANY_SAMPLES_PASSED, ANY_SAMPLES_PASSED_CONSERVATIVE, TRANSFORM_FEEDBACK_OVERFLOW, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW) is FALSE, all rendering commands described in section 2.4 are discarded and have no effect when issued between **BeginConditionalRender** and the corresponding **EndConditionalRender**.

The effect of commands setting current vertex state, such as **Color** or **VertexAttrib**, are undefined. If the result (SAMPLES_PASSED) of the query is non-zero, or if the result (ANY_SAMPLES_PASSED, ANY_SAMPLES_PASSED_CONSERVATIVE, TRANSFORM_FEEDBACK_OVERFLOW, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW) is TRUE, such commands are not discarded.

mode specifies how **BeginConditionalRender** interprets the results of the occlusion query given by *id*.

If *mode* is QUERY_WAIT, the GL waits for the results of the query to be available and then uses the results to determine if subsquent rendering commands are discarded.

If *mode* is QUERY_NO_WAIT, the GL may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If *mode* is QUERY_BY_REGION_WAIT, the GL will also wait for occlusion query results and discard rendering commands if the result of the occlusion query is zero. If the query result is non-zero, subsequent rendering commands are executed, but the GL may discard the results of the commands for any region of the framebuffer that did not contribute to the sample count in the specified occlusion query. Any such discarding is done in an implementation-dependent manner, but the rendering command results may not be discarded for any samples that contributed to the occlusion query sample count.

If *mode* is QUERY_BY_REGION_NO_WAIT, the GL operates as in QUERY_BY_REGION_WAIT, but may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If *mode* is QUERY_WAIT_INVERTED, QUERY_NO_WAIT_INVERTED, QUERY_BY_REGION_WAIT_INVERTED, or QUERY_BY_REGION_NO_WAIT_INVERTED then the condition used to determine whether or not to render subsequent drawing commands is negated with respect to QUERY_WAIT, QUERY_NO_WAIT, QUERY_BY_REGION_WAIT, or QUERY_BY_REGION_NO_WAIT, respectively.

If *mode* is QUERY_NO_WAIT_INVERTED or QUERY_BY_REGION_NO_WAIT_INVERTED, the GL may choose to unconditionally execute subsequent rendering commands without waiting for the query to complete.

Errors

An INVALID_OPERATION error is generated by **BeginConditionalRender** if called while conditional rendering is in progress.

An INVALID_VALUE error is generated by **BeginConditionalRender** if *id* is not the name of an existing query object.

An INVALID_OPERATION error is generated by **BeginConditionalRender** if *id* is the name of a query object with a target other than SAMPLES_PASSED, ANY_SAMPLES_PASSED, ANY_SAMPLES_PASSED_CONSERVATIVE, TRANSFORM_FEEDBACK_OVERFLOW, or TRANSFORM_FEEDBACK_STREAM_OVERFLOW; or if *id* is the name of a query currently in progress.

An INVALID_ENUM error is generated by **BeginConditionalRender** if *mode* is not QUERY_WAIT, QUERY_NO_WAIT, QUERY_BY_REGION_WAIT, QUERY_BY_REGION_NO_WAIT, QUERY_WAIT_INVERTED, QUERY_NO_WAIT_INVERTED, QUERY_BY_REGION_WAIT_INVERTED, or QUERY_BY_REGION_NO_WAIT_INVERTED.

An INVALID_OPERATION error is generated by **EndConditionalRender** if called while conditional rendering is not in progress.

10.10 Submission Queries

Submission queries use query objects to track the number of vertices and primitives that are issued to the GL using draw commands.

When **BeginQuery** is called with a target of VERTICES_SUBMITTED, the submitted vertices count maintained by the GL is set to zero. When a vertices submitted query is active, the submitted vertices count is incremented every time a vertex is transferred to the GL (see sections 10.3.5 and 10.4). In the case of primitive types with adjacency information (see sections 10.1.11, 10.1.12, 10.1.13, and 10.1.14), implementations may or may not count vertices not belonging to the main primitive. In the case of line loop primitives (see section 10.1.3), implementations are allowed to count the first vertex twice for the purposes of VERTICES_SUBMITTED queries. Additionally, vertices corresponding to incomplete primitives may or may not be counted.

When **BeginQuery** is called with a target of PRIMITIVES_SUBMITTED, the submitted primitives count maintained by the GL is set to zero. When a primitives submitted query is active, the submitted primitives count is incremented every time a point, line, triangle, or patch primitive is transferred to the GL (see sections 10.1, 10.3.5, and 10.3.6). Restarting a primitive topology using the primitive restart

index has no effect on the issued primitives count. Incomplete primitives may or may not be counted.

Chapter 11

Programmable Vertex Processing

When the program object currently in use for the vertex stage (see section 7.3) includes a vertex shader, its shader is considered *active* and is used to process vertices transferred to the GL (see section 11.1). Vertices may be further processed by tessellation and geometry shaders (see sections 11.2 and 11.3). The resulting transformed vertices are then processed as described in chapter 13.

If the current vertex stage program object has no vertex shader, or no program object is current for the vertex stage, then a fixed-function method for vertex processing is used instead. The transformation and lighting operations making up this method are described in chapter 12.

11.1 Vertex Shaders

Vertex shaders describe the operations that occur on vertex values and their associated data. When the program object currently in use for the vertex stage includes a vertex shader, its vertex shader is considered *active* and is used to process vertices.

Vertex attributes are per-vertex values available to vertex shaders, and are specified as described in section 10.2.

11.1.1 Vertex Attributes

Vertex shaders can define named attribute variables, which are bound to generic vertex attributes transferred by drawing commands. This binding can be specified in the shader text using the `location` qualifier, in a SPIR-V shader using the `Location` decoration, by the application before the program is linked, or automatically assigned by the GL when the program is linked.

Data type	`component layout` qualifier	Components used
scalar	0 or unspecified	x
scalar	1	y
scalar	2	z
scalar	3	w
two-component vector	0 or unspecified	(x, y)
two-component vector	1	(y, z)
two-component vector	2	(z, w)
three-component vector	0 or unspecified	(x, y, z)
three-component vector	1	(y, z, w)
four-component vector	0 or unspecified	(x, y, z, w)

Table 11.1: Generic attribute components accessed by attribute variables.

Vertex shaders can also access built-in variables corresponding to conventional vertex attributes such as vertex coordinates, normals, and colors.

When an attribute variable declared using one of the scalar or vector data types enumerated in table 11.3 is bound to a generic attribute index i, its value(s) are taken from the components of generic attribute i. The generic attribute components used depend on the type of the variable and value of the `component layout` qualifier (if any) specified in the variable declaration, as identified in table 11.1. An attribute variable declared using a combination of data type and `component layout` qualifier not listed in this table is not supported and will result in shader compilation errors.

When an attribute variable declared using a matrix type is bound to a generic attribute index i, its values are taken from consecutive generic attributes beginning with generic attribute i. Such matrices are treated as an array of column vectors with values taken from the generic attributes identified in table 11.2. Individual column vectors are taken from generic attribute components according to table 11.1, using the vector type from table 11.2.

When an attribute variable declared using an array type is bound to generic attribute index i, the active array elements are assigned to consecutive generic attributes beginning with generic attribute i. The number of attributes and components assigned to each element are determined according to the data type of array elements and `component layout` qualifier (if any) specified in the declaration of the array, as described above.

For the 64-bit double precision types listed in table 11.3, no default attribute

Data type	Column vector type `layout` qualifier	Generic attributes used
`mat2`, `dmat2`	two-component vector	$i, i+1$
`mat2x3`, `dmat2x3`	three-component vector	$i, i+1$
`mat2x4`, `dmat2x4`	four-component vector	$i, i+1$
`mat3x2`, `dmat3x2`	two-component vector	$i, i+1, i+2$
`mat3`, `dmat3`	three-component vector	$i, i+1, i+2$
`mat3x4`, `dmat3x4`	four-component vector	$i, i+1, i+2$
`mat4x2`, `dmat4x2`	two-component vector	$i, i+1, i+2, i+3$
`mat4x3`, `dmat4x3`	three-component vector	$i, i+1, i+2, i+3$
`mat4`, `dmat4`	four-component vector	$i, i+1, i+2, i+3$

Table 11.2: Generic attributes and vector types used by column vectors of matrix variables bound to generic attribute index i.

Data type	Command
`int`	**VertexAttribI1i**
`ivec2`	**VertexAttribI2i**
`ivec3`	**VertexAttribI3i**
`ivec4`	**VertexAttribI4i**
`uint`	**VertexAttribI1ui**
`uvec2`	**VertexAttribI2ui**
`uvec3`	**VertexAttribI3ui**
`uvec4`	**VertexAttribI4ui**
`float`	**VertexAttrib1***
`vec2`	**VertexAttrib2***
`vec3`	**VertexAttrib3***
`vec4`	**VertexAttrib4***
`double`	**VertexAttribL1d**
`dvec2`	**VertexAttribL2d**
`dvec3`	**VertexAttribL3d**
`dvec4`	**VertexAttribL4d**

Table 11.3: Scalar and vector vertex attribute types and **VertexAttrib*** commands used to set the values of the corresponding generic attribute.

values are provided if the values of the vertex attribute variable are specified with fewer components than required for the attribute variable. For example, the fourth component of a variable of type `dvec4` will be undefined if specified using **VertexAttribL3dv**, or using a vertex array specified with **VertexAttribLPointer** and a size of three.

The command

> `void` **BindAttribLocation**(`uint` *program*`,` `uint` *index*`,` `const char *`*name* `);`

specifies that the attribute variable named *name* in program *program* should be bound to generic vertex attribute *index* when the program is next linked. If *name* was bound previously, its assigned binding is replaced with *index*, but the new binding becomes effective only when the program is next linked. *name* must be a null-terminated string. **BindAttribLocation** has no effect until the program is linked. In particular, it doesn't modify the bindings of active attribute variables in a program that has already been linked.

Built-in attribute variables are automatically bound to conventional attributes, and can not have an assigned binding.

When a program is linked, any active attributes without a binding specified either through **BindAttribLocation** or explicitly set within the shader text or SPIR-V binary will automatically be bound to vertex attributes by the GL. Such bindings may be queried using the command **GetAttribLocation**. **LinkProgram** will fail if the assigned binding of an active attribute variable would cause the GL to reference a non-existent generic attribute (one greater than or equal to the value of `MAX_VERTEX_ATTRIBS`). **LinkProgram** will fail if the attribute bindings specified either by **BindAttribLocation** or explicitly set within the shader text or SPIR-V binary do not leave enough space to assign a location for an active matrix attribute or an active attribute array, both of which require multiple contiguous generic attributes. If an active attribute has a binding explicitly set within the shader text or SPIR-V binary and a different binding assigned by **BindAttribLocation**, the assignment in the shader text or SPIR-V binary is used. **LinkProgram** will also fail if the vertex shaders used in the program object contain assignments (not removed during pre-processing) to an attribute variable bound to generic attribute zero and to the conventional vertex position (`gl_Vertex`).

BindAttribLocation may be issued before any vertex shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with `"gl_"`) to an index, including a name that is never used as an attribute in any vertex shader object. Assigned bindings for attribute variables that do not exist or are not active are ignored. **BindAttribLocation** has no effect on

SPIR-V shaders, since the locations must always be fully specified in the SPIR-V shader as described in section 11.1.1.1.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

An INVALID_OPERATION error is generated if *name* is a built-in attribute variable starting with the reserved "gl_" prefix.

To determine the set of active vertex attribute variables used by a program, applications can query the properties and active resources of the PROGRAM_INPUT interface of a program including a vertex shader.

Additionally, the command

```
void GetActiveAttrib( uint program, uint index,
    sizei bufSize, sizei *length, int *size, enum *type,
    char *name );
```

can be used to determine properties of the active input variable assigned the index *index* in program object *program*. If no error occurs, the command is equivalent (assuming no errors are generated) to:

```
const enum props[] = { ARRAY_SIZE, TYPE };
GetProgramResourceName(program, PROGRAM_INPUT,
    index, bufSize, length, name);
GetProgramResourceiv(program, PROGRAM_INPUT,
    index, 1, &props[0], 1, NULL, size);
GetProgramResourceiv(program, PROGRAM_INPUT,
    index, 1, &props[1], 1, NULL, (int *)type);
```

For **GetActiveAttrib**, all active vertex shader input variables are enumerated, including the special built-in inputs gl_BaseInstance, gl_BaseVertex, gl_DrawID, gl_InstanceID, and gl_VertexID.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_VALUE error is generated if *index* is not the index of an active input variable in *program*.

An INVALID_VALUE error is generated for all values of *index* if *program* does not include a vertex shader, as it has no active vertex attributes.

An INVALID_VALUE error is generated if *bufSize* is negative.

The command

`int` **GetAttribLocation**(`uint` *program*, `const char` **name*);

can be used to determine the location assigned to the active input variable named *name* in program object *program*.

Errors

If *program* has been linked successfully but contains no vertex shader, no error is generated but -1 will be returned.

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_OPERATION error is generated and -1 is returned if *program* has not been linked successfully.

Otherwise, the command is equivalent to

GetProgramResourceLocation(*program*, PROGRAM_INPUT, *name*);

There is an implementation-dependent limit on the number of active attribute variables (either conventional or generic) in a vertex shader. A program with more than the value of MAX_VERTEX_ATTRIBS active attribute variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources. For the purposes of this test, attribute variables of the type dvec3, dvec4, dmat2x3, dmat2x4, dmat3, dmat3x4, dmat4x3, and dmat4 may count as consuming twice as many attributes as equivalent single-precision types. While these types use the same number of generic attributes

as their single-precision equivalents, implementations are permitted to consume two single-precision vectors of internal storage for each three- or four-component double-precision vector.

The values of generic attributes sent to generic attribute index i are part of current state, just like the conventional attributes. If a new program object has been made active, then these values will be tracked by the GL in such a way that the same values will be observed by attributes in the new program object that are also bound to index i.

It is possible for an application to bind more than one attribute name to the same location. This is referred to as *aliasing*. This will only work if only one of the aliased attributes is active in the executable program, or if no path through the shader consumes more than one attribute of a set of attributes aliased to the same location. A link error can occur if the linker determines that every path through the shader consumes multiple aliased attributes, but implementations are not required to generate an error in this case. The compiler and linker are allowed to assume that no aliasing is done, and may employ optimizations that work only in the absence of aliasing. It is not possible to alias generic attributes with conventional ones.

11.1.1.1 SPIR-V Vertex Input Interface

When a SPIR-V vertex stage is present, the vertex shader variables listed by `OpEntryPoint` with the `Input` storage class form the vertex input attribute interface. These inputs must be decorated with a `Location` and can also be decorated with a `Component` decoration. These correspond to the `location` and `component` layout discussed in section 11.1.1.

The vertex shader input variables are matched only by the `Location` and `Component` decorations, and must have a corresponding attribute and binding in the pipeline.

11.1.2 Vertex Shader Variables

Vertex shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Vertex shaders also have access to samplers to perform texturing operations, as described in section 7.11.

11.1.2.1 Output Variables

A vertex shader may define one or more *output variables* or *outputs* (see the OpenGL Shading Language Specification).

The OpenGL Shading Language Specification also defines a set of built-in outputs that vertex shaders can write to (see section 7.1("Built-In Variables") of the OpenGL Shading Language Specification). These output variables are used as the mechanism to communicate values to the next active stage in the vertex processing pipeline: either the tessellation control shader, the tessellation evaluation shader, the geometry shader, or the fixed-function vertex processing stages leading to rasterization.

If the output variables are passed directly to the vertex processing stages leading to rasterization, the values of all outputs are expected to be interpolated across the primitive being rendered, unless flatshaded. Otherwise the values of all outputs are collected by the primitive assembly stage and passed on to the subsequent pipeline stage once enough data for one primitive has been collected.

The number of components (individual scalar numeric values) of output variables that can be written by the vertex shader, whether or not a tessellation control, tessellation evaluation, or geometry shader is active, is given by the value of the implementation-dependent constant `MAX_VERTEX_OUTPUT_COMPONENTS`. For the purposes of counting input and output components consumed by a shader, variables declared as vectors, matrices, and arrays will all consume multiple components. Each component of variables declared as double-precision floating-point scalars, vectors, or matrices may be counted as consuming two components.

When a program is linked, all components of any outputs written by a vertex shader will count against this limit. A program whose vertex shader writes more than the value of `MAX_VERTEX_OUTPUT_COMPONENTS` components worth of outputs may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Additionally, when linking a program containing only a vertex and fragment shader, there is a limit on the total number of components used as vertex shader outputs or fragment shader inputs. This limit is given by the value of the implementation-dependent constant `MAX_VARYING_COMPONENTS`. The implementation-dependent constant `MAX_VARYING_VECTORS` has a value equal to the value of `MAX_VARYING_COMPONENTS` divided by four. Each output variable component used as either a vertex shader output or fragment shader input counts against this limit, except for the components of `gl_Position`. A program containing only a vertex and fragment shader that accesses more than this limit's worth of components of outputs may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Each program object can specify a set of output variables from one shader to be recorded in transform feedback mode (see section 13.3). The variables that can be recorded are those emitted by the **first** active shader, in order, from the following list:

- geometry shader
- tessellation evaluation shader
- tessellation control shader
- vertex shader

The set of variables to record can be specified in shader text using the `xfb_buffer`, `xfb_offset`, or `xfb_stride` `layout` qualifiers. For SPIR-V shaders, these are specified by the `XfbBuffer`, `Offset`, and `XfbStride` decorations, respectively. When a SPIR-V entry point is using any of these for transform feedback, it must declare the `Xfb` Execution Mode.

When recording output variables of each vertex in transform feedback mode, a fixed amount of memory is reserved in the buffer bound to each transform feedback buffer binding point. Each output variable recorded is associated with a binding point, specified by the `xfb_buffer` `layout` qualifier. Each output variable is written to its associated transform feedback binding point at an offset specified by the `xfb_offset` `layout` qualifier, in basic machine units, relative to the base of the memory reserved for its vertex. The amount of memory reserved in each transform feedback binding point for a single vertex can be specified using the `xfb_stride` `layout` qualifier. If no `xfb_stride` qualifier is specified for a binding point, the stride is derived by identifying the variable associated with the binding point having the largest offset, and then adding the offset and the size of the variable, in basic machine units. If any variable associated with the binding point contains double-precision floating-point components, the derived stride is aligned to the next multiple of eight basic machine units. If a binding point has no `xfb_stride` qualifier and no associated output variables, its stride is zero.

When no `xfb_buffer`, `xfb_offset`, or `xfb_stride` `layout` qualifiers are specified, the set of variables to record is specified with the command

```
void TransformFeedbackVaryings( uint program,
    sizei count, const char * const *varyings,
    enum bufferMode );
```

program specifies the program object. *count* specifies the number of output variables used for transform feedback. *varyings* is an array of *count* zero-terminated strings specifying the names of the outputs to use for transform feedback. The variables specified in *varyings* can be either built-in (beginning with `"gl_"`) or user-defined variables. Each variable can either be a basic type or an array of basic types. Structure, array of array and array of structure types cannot be captured

directly. Base-level members of aggregates can be captured by specifying the fully qualified path identifying the member, using the same rules with which active resource lists are enumerated for program interfaces as described in section 7.3.1.1, with one exception. To allow capturing whole arrays or individual elements of an array, there are additional rules for array variables. To capture a single element, the name of the output array is specified with a constant-integer index "`name[x]`" where `name` is the name of the array variable and `x` is the constant-integer index of the array element. To capture the whole of the output array, `name` is specified without the array index or square brackets. Output variables are written out in the order they appear in the array *varyings*. *bufferMode* is either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS`, and identifies the mode used to capture the outputs when transform feedback is active.

The variables in *varyings* are assigned binding points and offsets sequentially, as though each were specified using the `xfb_buffer` and `xfb_offset` `layout` qualifiers. The strides associated with each binding point are derived by adding the offset and size of the last variable associated with that binding point. The first variable in *varyings* is assigned a binding point and offset of zero. When *bufferMode* is `INTERLEAVED_ATTRIBS`, each subsequent variable is assigned to the same binding point as the previous variable and an offset equal to the sum of the offset and size of the previous variable. When *bufferMode* is `SEPARATE_ATTRIBS`, each subsequent variable is assigned to the binding point following the binding point of the previous variable with an offset of zero.

Several special identifiers are supported when *bufferMode* is `INTERLEAVED_ATTRIBS`. These identifiers do not identify output variables captured in transform feedback mode, but can be used to modify the binding point and offsets assigned to subsequent variables. If a string in *varyings* is `gl_NextBuffer`, the next variable in *varyings* will be assigned to the next binding point, with an offset of zero. If a string in *varyings* is `gl_SkipComponents1`, `gl_SkipComponents2`, `gl_SkipComponents3`, or `gl_SkipComponents4`, the variable is treated as specifying a one- to four-component floating-point output variable with undefined values. No data will be recorded for such strings, but the offset assigned to the next variable in *varyings* and the stride of the assigned binding point will be affected.

Errors

An `INVALID_VALUE` error is generated if *program* is not the name of either a program or shader object.

An `INVALID_OPERATION` error is generated if *program* is the name of a shader object.

An INVALID_VALUE error is generated if *count* is negative.

An INVALID_ENUM error is generated if *bufferMode* is not SEPARATE_ATTRIBS or INTERLEAVED_ATTRIBS.

An INVALID_VALUE error is generated if *bufferMode* is SEPARATE_ATTRIBS and *count* is greater than the value of the implementation-dependent limit MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS.

An INVALID_OPERATION error is generated if any pointer in *varyings* identifies the special names gl_NextBuffer, gl_SkipComponents1, gl_SkipComponents2, gl_SkipComponents3, or gl_SkipComponents4 and *bufferMode* is not INTERLEAVED_ATTRIBS, or if the number of gl_NextBuffer pointers in *varyings* is greater than or equal to the value of MAX_TRANSFORM_FEEDBACK_BUFFERS.

The state set by **TransformFeedbackVaryings** or using transform feedback layout qualifiers has no effect on the execution of the program until *program* is subsequently linked. When **LinkProgram** is called, the program is linked so that the values of the specified outputs for the vertices of each primitive generated by the GL are written to one or more buffer objects. If the set of output variables to record in transform feedback mode is specified by **TransformFeedbackVaryings**, a program will fail to link if:

- the *count* specified by **TransformFeedbackVaryings** is non-zero, but the program object has no vertex, tessellation control, tessellation evaluation, or geometry shader;
- any variable name specified in the *varyings* array is not one of gl_NextBuffer, gl_SkipComponents1, gl_SkipComponents2, gl_SkipComponents3, or gl_SkipComponents4, and is not declared as a built-in or user-defined output variable in the shader stage whose outputs can be recorded;
- any two entries in the *varyings* array specify the same output variable or include the same elements from an array variable (different elements from the same array are permitted);
- the total number of components to capture in any output in *varyings* is greater than the value of MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS and the buffer mode is SEPARATE_ATTRIBS;
- the total number of components to capture is greater than the value of MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS and the buffer mode is INTERLEAVED_ATTRIBS; or

- the set of outputs to capture to any single binding point includes outputs from more than one vertex stream.

If the set of output variables to record in transform feedback mode is specified using `layout` qualifiers, a program will fail to link if:

- any pair of variables associated with the same binding point overlap in memory (where the offset of the first variable is less than or equal to the offset of the second, but the sum of the offset and size of the first variable is greater than the offset of the second);
- any binding point has a stride declared using the `xfb_stride layout` qualifier and the sum of the offset and size of any variable associated with that binding point exceeds the value of this stride;
- any variable containing double-precision floating-point components
  - has an `xfb_offset layout` qualifier that is not a multiple of eight; or
  - is associated with a binding point with an `xfb_stride layout` qualifier that is not a multiple of eight;
- the sum of the offset and size of any variable exceeds the maximum stride supported by the implementation (four times the value of `MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS`); or
- the `xfb_stride layout` qualifier for any binding point exceeds the maximum stride supported by the implementation.

For transform feedback purposes, each component of outputs declared as double-precision floating-point scalars, vectors, or matrices is considered to consume eight basic machine units, and each component of any other type is considered to consume four basic machine units.

To determine the set of output variables in a linked program object that will be captured in transform feedback mode and the binding points to which those variables are written, applications can query the properties and active resources of the `TRANSFORM_FEEDBACK_VARYING` and `TRANSFORM_FEEDBACK_BUFFER` interfaces.

If the shader used to record output variables for transform feedback varyings uses the `xfb_buffer`, `xfb_offset`, or `xfb_stride layout` qualifiers, or its SPIR-V entry point declares the `Xfb` Execution Mode, the values specified

by **TransformFeedbackVaryings** are ignored, and the set of variables captured for transform feedback is instead derived from the specified `layout` qualifiers or SPIR-V decorations. Outputs specifying both an `XfbBuffer` and an `Offset` are captured, while outputs not specifying both of these are not captured. Values are captured each time the shader writes to such a decorated object.

Additionally, the command

```
void GetTransformFeedbackVarying( uint program,
    uint index, sizei bufSize, sizei *length, sizei *size,
    enum *type, char *name );
```

can be used to enumerate properties of a single output variable captured in transform feedback mode, and is equivalent (assuming no errors are generated) to:

```
const enum props[] = { ARRAY_SIZE, TYPE };
GetProgramResourceName(program, TRANSFORM_FEEDBACK_VARYING,
    index, bufSize, length, name);
GetProgramResourceiv(program, TRANSFORM_FEEDBACK_VARYING,
    index, 1, &props[0], 1, NULL, size);
GetProgramResourceiv(program, TRANSFORM_FEEDBACK_VARYING,
    index, 1, &props[1], 1, NULL, (int *)type);
```

Special output names (e.g., `gl_NextBuffer`, `gl_SkipComponents1`) passed to **TransformFeedbackVaryings** in the *varyings* array are counted as outputs to be recorded for the purposes of determining the value of `TRANSFORM_FEEDBACK_VARYINGS` and for determining the variable selected by *index* in **GetTransformFeedbackVarying**. If *index* identifies `gl_NextBuffer`, the values zero and `NONE` will be written to *size* and *type*, respectively. If *index* is of the form `gl_SkipComponents`*n*, the value `NONE` will be written to *type* and the number of components *n* will be written to *size*.

GetTransformFeedbackVarying may be used to query any transform feedback output variable, not just those specified with **TransformFeedbackVarying**.

11.1.3 Shader Execution

If there is an active program object present for the vertex, tessellation control, tessellation evaluation, or geometry shader stages, the executable code for these active programs is used to process incoming vertex values, instead of the fixed-function method described in chapter 12.

The following sequence of operations is performed:

- Vertices are processed by the vertex shader (see section 11.1) and assembled into primitives as described in sections 10.1 through 10.3.
- If the current program contains a tessellation control shader, each individual patch primitive is processed by the tessellation control shader (section 11.2.1). Otherwise, primitives are passed through unmodified. If active, the tessellation control shader consumes its input patch and produces a new patch primitive, which is passed to subsequent pipeline stages.
- If the current program contains a tessellation evaluation shader, each individual patch primitive is processed by the tessellation primitive generator (section 11.2.2) and tessellation evaluation shader (see section 11.2.3). Otherwise, primitives are passed through unmodified. When a tessellation evaluation shader is active, the tessellation primitive generator produces a new collection of point, line, or triangle primitives to be passed to subsequent pipeline stages. The vertices of these primitives are processed by the tessellation evaluation shader. The patch primitive passed to the tessellation primitive generator is consumed by this process.
- If the current program contains a geometry shader, each individual primitive is processed by the geometry shader (section 11.3). Otherwise, primitives are passed through unmodified. If active, the geometry shader consumes its input patch primitive. However, each geometry shader invocation may emit new vertices, which are arranged into primitives and passed to subsequent pipeline stages.

Implementations are allowed to skip the execution of certain shader invocations, and to execute additional shader invocations for any shader type during programmable vertex processing due to implementation dependent reasons, including the execution of shader invocations that do not have an active program object present for the particular shader stage, as long as the results of rendering otherwise remain unchanged.

Following shader execution, the fixed-function operations described in chapter 13 are applied.

Special considerations for vertex shader execution are described in the following sections.

11.1.3.1 Shader Only Texturing

This section describes texture functionality that is only accessible through shaders (of all types). Also refer to chapter 8 and to section 8.9(“Texture Functions”) of the OpenGL Shading Language Specification.

11.1.3.2 Texel Fetches

The OpenGL Shading Language `texelFetch` built-ins provide the ability to extract a single texel from a specified texture image. Texel fetches cannot access cube maps.

The integer coordinates (i, j, k) passed to `texelFetch` are used to point-sample the texture image. The level-of-detail accessed is computed by adding the specified level-of-detail parameter *lod* to the base level of the texture, $level_{base}$.

Texel fetch proceeds similarly to the steps described for texture access in section 11.1.3.5, with the exception that none of the operations controlled by sampler object state are performed, including:

- level-of-detail clamping;
- texture wrap mode application;
- filtering (however, a mipmapped minification filter is required to access any level-of-detail other than the base level);
- depth comparison.

The steps that are performed are:

- validation of texel coordinates as described below, including the computed level-of-detail, (i, j, k), the specified level for array textures, and texture completeness;
- sRGB conversion of fetched values as described in section 8.24;
- conversion to base color C_b;
- component swizzling.

The results of `texelFetch` built-ins are undefined if any of the following conditions hold:

- the computed level-of-detail is less than the texture's base level ($level_{base}$) or greater than the maximum defined level, q (see section 8.14.3)
- the computed level-of-detail is not the texture's base level and the texture's minification filter is `NEAREST` or `LINEAR`
- the layer specified for array textures is negative, or greater than or equal to the number of layers in the array texture

- the texel coordinates (i, j, k) refer to a texel outside the defined extents of the computed level-of-detail, where any of

$$\begin{aligned} i &< -b_s & \qquad i &\geq w_s - b_s \\ j &< -b_s & \qquad j &\geq h_s - b_s \\ k &< -b_s & \qquad k &\geq d_s - b_s \end{aligned}$$

  and b_s, w_s, h_s, and d_s refer to the border size, width, height, and depth of the image, as defined in section 8.5.3.

- the texture being accessed is not complete, as defined in section 8.17

- the texture being accessed is not bound.

In all the above cases, if the context was created with robust buffer access enabled (see section 10.3.7), the result of the texture fetch is zero, or a texture source color of $(0, 0, 0, 1)$ in the case of a texel fetch from an incomplete texture. If robust buffer access is not enabled, the result of the texture fetch is undefined in each case.

11.1.3.3 Multisample Texel Fetches

Multisample buffers do not have mipmaps, and there is no level-of-detail parameter for multisample texel fetches. Instead, an integer parameter selects the sample number to be fetched from the buffer. The number identifying the sample is the same as the value used to query the sample location using **GetMultisamplefv**. Multisample textures are not filtered when samples are fetched, and filter state is ignored.

The results of a multisample texel fetch are undefined if any of the following conditions hold:

- the texel coordinates (i, j, k) refer to a texel outside the extents of the multisample texture image, where any of

$$\begin{aligned} i &< 0 & \qquad i &\geq w_s \\ j &< 0 & \qquad j &\geq h_s \\ k &< 0 & \qquad k &\geq d_s \end{aligned}$$

  and the size parameters w_s, h_s, and d_s refer to the width, height, and depth of the image

- the specified *sample* number does not exist (is negative, or greater than or equal to the number of samples in the texture).

Additionally, these fetches may only be performed on a multisample texture sampler. No other sample or fetch commands may be performed on a multisample texture sampler.

11.1.3.4 Texture Queries

The `textureSize` functions provide the ability to query the size of a texture image. The level-of-detail value *lod* passed in as an argument to the texture size functions is added to the $level_{base}$ of the texture to determine a texture image level. The dimensions of that image level (excluding a possible border) are then returned. The value returned is undefined if:

- the texture is not complete;
- the texture is not mipmap complete, and the computed texture level is not $level_{base}$; or
- the computed texture image level is outside the range $[level_{base}, q]$, where q is defined in section 8.14.3.

When querying the size of an array texture, both the dimensions and the layer count are returned.

The `textureQueryLevels` functions provide the ability to query the number of accessible mipmap levels in a texture object associated with a sampler uniform. If the sampler is associated with an immutable-format texture object (see section 8.19), the value returned will be:

$$\min(level_{immut} - 1, level_{max}) - level_{base} + 1.$$

Otherwise, the value returned will be an implementation-dependent value between zero and $q - level_{base} + 1$. The value returned in that case must satisfy the following constraints:

- if all levels of the texture have zero size, zero must be returned
- if the texture is complete, a non-zero value must be returned
- if the texture is complete and is accessed with a minification filter requiring mipmaps, $q - level_{base} + 1$ must be returned.

11.1.3.5 Texture Access

Shaders have the ability to do a lookup into a texture map. The maximum number of texture image units available to shaders are the values of the implementation-dependent constants

- `MAX_VERTEX_TEXTURE_IMAGE_UNITS` (for vertex shaders),
- `MAX_TESS_CONTROL_TEXTURE_IMAGE_UNITS` (for tessellation control shaders),
- `MAX_TESS_EVALUATION_TEXTURE_IMAGE_UNITS` (for tessellation evaluation shaders),
- `MAX_GEOMETRY_TEXTURE_IMAGE_UNITS` (for geometry shaders),
- `MAX_TEXTURE_IMAGE_UNITS` (for fragment shaders), and
- `MAX_COMPUTE_TEXTURE_IMAGE_UNITS` (for compute shaders).

All active shaders, and fixed-function fragment processing if no fragment shader is active, combined cannot use more than the value of `MAX_COMBINED_TEXTURE_IMAGE_UNITS` texture image units. If more than one pipeline stage accesses the same texture image unit, each such access counts separately against the `MAX_COMBINED_TEXTURE_IMAGE_UNITS` limit.

When a texture lookup is performed in a shader, the filtered texture value τ is computed in the manner described in sections 8.14 and 8.15, and converted to a texture base color C_b as shown in table 16.1, followed by application of the texture swizzle as described in section 16.1 to compute the texture source color C_s and A_s.

The resulting four-component vector (R_s, G_s, B_s, A_s) is returned to the shader. Texture lookup functions (see section 8.9("Texture Functions") of the OpenGL Shading Language Specification) may return floating-point, signed, or unsigned integer values depending on the function and the internal format of the texture.

In shaders other than fragment shaders, it is not possible to perform automatic level-of-detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 8.14. Hence, there is no automatic selection of an image array level. Minification or magnification of a texture map is controlled by a level-of-detail value optionally passed as an argument in the texture lookup functions. If the texture lookup function supplies an explicit level-of-detail value l, then the pre-bias level-of-detail value $\lambda_{base}(x, y) = l$ (replacing equation 8.8). If the texture lookup function does not supply an explicit level-of-detail value, then $\lambda_{base}(x, y) = 0$. The scale factor $\rho(x, y)$ and its approximation function $f(x, y)$ (see equation 8.12) are ignored.

Texture lookups involving textures with depth component data generate a texture base color C_b either using depth data directly or by performing a comparison with the D_{ref} value used to perform the lookup, as described in section 8.23.1, and expanding the resulting value R_t to a color $C_b = (R_t, 0, 0, 1)$. In either case, swizzling of C_b is then performed as described above, but only the first component $C_s[0]$ is returned to the shader. The comparison operation is requested in the shader by using any of the shadow sampler types (`sampler*Shadow`), and in the texture using the `TEXTURE_COMPARE_MODE` parameter. These requests must be consistent; the results of a texture lookup are undefined if any of the following conditions are true:

- The sampler used in a texture lookup function is not one of the shadow sampler types, the texture object's base internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the `TEXTURE_COMPARE_MODE` is not `NONE`.
- The sampler used in a texture lookup function is one of the shadow sampler types, the texture object's base internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the `TEXTURE_COMPARE_MODE` is `NONE`.
- The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's base internal format is not `DEPTH_COMPONENT` or `DEPTH_STENCIL`.
- The sampler used in a texture lookup function is one of the shadow sampler types, the texture object's base internal format is `DEPTH_STENCIL`, and the `DEPTH_STENCIL_TEXTURE_MODE` is not `DEPTH_COMPONENT`.

The stencil index texture internal component is ignored if the base internal format is `DEPTH_STENCIL` and the value of `DEPTH_STENCIL_TEXTURE_MODE` is not `STENCIL_INDEX`.

Texture lookups involving texture objects with an internal format of `DEPTH_STENCIL` can read the stencil value as described in section 8.23 by setting the `DEPTH_STENCIL_TEXTURE_MODE` to `STENCIL_INDEX`. Textures with a `STENCIL_INDEX` base internal format may also be used to read stencil data. The stencil value is read as an integer and assigned to R_t. An unsigned integer sampler should be used to lookup the stencil component, otherwise the results are undefined.

If a sampler is used in a shader and the sampler's associated texture is not complete, as defined in section 8.17, $(0.0, 0.0, 0.0, 1.0)$, in floating-point, will be returned for a non-shadow sampler and 0 for a shadow sampler. In this case, if the sampler is declared in the shader as a signed or unsigned integer sampler type,

undefined values are returned as specified in section 9.9(“Texture Functions”) of the OpenGL Shading Language Specification when the texture format and sampler type are unsupported combinations.

11.1.3.6 Atomic Counter Access

Shaders have the ability to set and get atomic counters. The maximum number of atomic counters available to shaders are the values of the implementation-dependent constants

- `MAX_VERTEX_ATOMIC_COUNTERS` (for vertex shaders),
- `MAX_TESS_CONTROL_ATOMIC_COUNTERS` (for tessellation control shaders),
- `MAX_TESS_EVALUATION_ATOMIC_COUNTERS` (for tessellation evaluation shaders),
- `MAX_GEOMETRY_ATOMIC_COUNTERS` (for geometry shaders),
- `MAX_FRAGMENT_ATOMIC_COUNTERS` (for fragment shaders), and
- `MAX_COMPUTE_ATOMIC_COUNTERS` (for compute shaders).

All active shaders combined cannot use more than the value of `MAX_COMBINED_ATOMIC_COUNTERS` atomic counters. If more than one pipeline stage accesses the same atomic counter, each such access counts separately against the `MAX_COMBINED_ATOMIC_COUNTERS` limit.

11.1.3.7 Image Access

Shaders have the ability to read and write to textures using image uniforms. The maximum number of image uniforms available to individual shader stages are the values of the implementation-dependent constants

- `MAX_VERTEX_IMAGE_UNIFORMS` (for vertex shaders),
- `MAX_TESS_CONTROL_IMAGE_UNIFORMS` (for tessellation control shaders),
- `MAX_TESS_EVALUATION_IMAGE_UNIFORMS` (for tessellation evaluation shaders),
- `MAX_GEOMETRY_IMAGE_UNIFORMS` (for geometry shaders),

- MAX_FRAGMENT_IMAGE_UNIFORMS (for fragment shaders), and
- MAX_COMPUTE_IMAGE_UNIFORMS (for compute shaders).

All active shaders combined cannot use more than the value of MAX_COMBINED_IMAGE_UNIFORMS image uniforms. If more than one shader stage accesses the same image uniform, each such access counts separately against the MAX_COMBINED_IMAGE_UNIFORMS limit.

11.1.3.8 Shader Storage Buffer Access

Shaders have the ability to read and write to buffer memory via buffer variables in shader storage blocks. The maximum number of shader storage blocks available to shaders are the values of the implementation-dependent constants

- MAX_VERTEX_SHADER_STORAGE_BLOCKS (for vertex shaders)
- MAX_TESS_CONTROL_SHADER_STORAGE_BLOCKS (for tessellation control shaders)
- MAX_TESS_EVALUATION_SHADER_STORAGE_BLOCKS (for tessellation evaluation shaders)
- MAX_GEOMETRY_SHADER_STORAGE_BLOCKS (for geometry shaders)
- MAX_FRAGMENT_SHADER_STORAGE_BLOCKS (for fragment shaders)
- MAX_COMPUTE_SHADER_STORAGE_BLOCKS (for compute shaders)

All active shaders combined cannot use more than the value of MAX_COMBINED_SHADER_STORAGE_BLOCKS shader storage blocks. If more than one pipeline stage accesses the same shader storage block, each such access counts separately against this combined limit.

11.1.3.9 Shader Inputs

Besides having access to vertex attributes and uniform variables, vertex shaders can access the read-only built-in variables gl_BaseInstance, gl_BaseVertex, gl_DrawID, gl_InstanceID, and gl_VertexID.

gl_BaseInstance holds the integer value passed to the *baseInstance* parameter to the command that resulted in the current shader invocation. In the case where the command has no *baseInstance* parameter, the value of gl_BaseInstance is zero.

`gl_BaseVertex` holds the integer value passed to the *baseVertex* parameter to the command that resulted in the current shader invocation. In the case where the command has no *baseVertex* parameter, the value of `gl_BaseVertex` is zero.

`gl_DrawID` holds the integer draw number the current draw being processed by the shader invocation. It is dynamically uniform. In **MultiDraw*** variants, this is the zero-based index of the draw within the list of draws processed by the command. In non-**MultiDraw*** commands, the value of `gl_DrawID` is always zero.

`gl_InstanceID` holds the integer instance number of the current primitive in an instanced draw call (see section 10.4).

`gl_VertexID` holds the integer index i explicitly passed to **ArrayElement** to specify the vertex, or implicitly passed by **DrawArrays** or one of the other drawing commands defined in section 10.4. The value of `gl_VertexID` is defined if and only if:

- the vertex comes from a vertex array command that specifies a complete primitive (a vertex array drawing command other than **ArrayElement**).
- all enabled vertex arrays have non-zero buffer object bindings, and
- the vertex does not come from a display list, even if the display list was compiled using one of the vertex array commands described above with data sourced from buffer objects.

Section 7.1("Built-In Variables") of the OpenGL Shading Language Specification also describes these variables.

11.1.3.10 Shader Outputs

A vertex shader can write to built-in as well as user-defined output variables. Interpolation of these values across the primitive is determined by the fragment shader input variable interpolation qualifier. section 13.6 and sections 4.3.6("Output Variables"), 4.5("Interpolation Qualifiers"), and 7.1("Built-In Variables") of the OpenGL Shading Language Specification for more detail.

The built-in outputs `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor`, and `gl_BackSecondaryColor` hold the front and back colors for the primary and secondary colors for the current vertex.

The built-in output `gl_TexCoord`[] is an array and holds the set of texture coordinates for the current vertex.

The built-in output gl_FogFragCoord is used as the c value described in section 16.4.

The built-in output gl_Position is intended to hold the homogeneous vertex position. Writing gl_Position is optional.

The built-in output variables gl_ClipVertex, gl_ClipDistance and gl_CullDistance respectively hold the vertex coordinate, and the clip distance and cull distance used in the clipping stage, as described in section 13.7. If clipping is enabled, only one of gl_ClipVertex and gl_ClipDistance should be written.

The built-in output gl_PointSize, if written, holds the size of the point to be rasterized, measured in pixels.

11.1.3.11 Position Invariance

If a vertex shader uses the built-in function ftransform to generate a vertex position, then this generally guarantees that the transformed position will be the same whether using this vertex shader or the fixed-function pipeline. This allows for correct multi-pass rendering algorithms, where some passes use fixed-function vertex transformation and other passes use a vertex shader. If a vertex shader does not use ftransform to generate a position, transformed positions are not guaranteed to match, even if the sequence of instructions used to compute the position match the sequence of transformations described in section 12.1.

11.1.3.12 Validation

It is not always possible to determine at link time if a program object can execute successfully, given that **LinkProgram** can not know the state of the remainder of the pipeline. Therefore validation is done when the first rendering command which triggers shader invocations is issued, to determine if the set of active program objects can be executed[1].

Errors

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL or launches compute work if the current set of active program objects cannot be executed, for reasons including:

[1]The OpenGL Specification differs from the OpenGL ES Specification in what validation takes place at draw-time. In particular, the OpenGL Specification does not require the implementation to detect mismatched interfaces between the shader stages as this might add extra draw-time overhead. Depending on the implementation, developers can activate more draw-time validation by enabling DEBUG_OUTPUT or using a debug context.

- A program object is active for at least one, but not all of the shader stages that were present when the program was linked.
- One program object is active for at least two shader stages and a second program is active for a shader stage between two stages for which the first program was active. The active compute shader is ignored for the purposes of this test.
- There is an active program for tessellation control, tessellation evaluation, or geometry stages with corresponding executable shader, but there is no active program with executable vertex shader.
- There is no current program object specified by **UseProgram**, there is a current program pipeline object, and the current program for any shader stage has been relinked since being applied to the pipeline object via **UseProgramStages** with the `PROGRAM_SEPARABLE` parameter set to `FALSE`.
- There is no current program object specified by **UseProgram**, there is a current program pipeline object, and that object is empty (no executable code is installed for any stage).
- Any two active samplers in the set of active program objects are of different types, but refer to the same texture image unit.
- Any active sampler in the current program object refers to a texture image unit where fixed-function fragment processing accesses a texture target that does not match the sampler type.
- The sum of the number of active samplers in the program and the number of texture image units enabled for fixed-function fragment processing exceeds the combined limit on the total number of texture image units allowed.
- The sum of the number of active atomic counters, atomic counter buffers, image uniforms, shader output resources, shader storage blocks, texture image units, and uniform blocks used by the current program objects exceeds the corresponding combined limit (the value of `MAX_COMBINED_ATOMIC_COUNTERS`, `MAX_COMBINED_ATOMIC_COUNTER_BUFFERS`, `MAX_COMBINED_IMAGE_UNIFORMS`, `MAX_COMBINED_SHADER_OUTPUT_RESOURCES`, `MAX_COMBINED_SHADER_STORAGE_BLOCKS`, `MAX_COMBINED_TEXTURE_IMAGE_UNITS`, and `MAX_COMBINED_UNIFORM_BLOCKS`, respectively).

Fixed-function fragment processing operations will be performed if the program object in use has no fragment shader.

The INVALID_OPERATION error generated by these rendering commands may not provide enough information to find out why the currently active program object would not execute. No information at all is available about a program object that would still execute, but is inefficient or suboptimal given the current GL state. As a development aid, use the command

`void` **ValidateProgram**(`uint` *program*);

to validate the program object *program* against the current GL state. Each program object has a boolean status, VALIDATE_STATUS, that is modified as a result of validation. This status may be queried with **GetProgramiv** (see section 7.14). If validation succeeded this status will be set to TRUE, otherwise it will be set to FALSE. If validation succeeded, no INVALID_OPERATION validation error is generated if *program* is made current via **UseProgram**, given the current state. If validation failed, such errors are generated under the current state.

ValidateProgram will check for all the conditions described in this section, and may check for other conditions as well. For example, it could give a hint on how to optimize some piece of shader code. The information log of *program* is overwritten with information on the results of the validation, which could be an empty string. The results written to the information log are typically only useful during application development; an application should not expect different GL implementations to produce identical information.

A shader should not fail to compile, and a program object should not fail to link due to lack of instruction space or lack of temporary variables. Implementations should ensure that all valid shaders and program objects may be successfully compiled, linked and executed.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

Separable program objects may have validation failures that cannot be detected without the complete program pipeline. Mismatched interfaces, improper usage of program objects together, and the same state-dependent failures can result in validation errors for such program objects. As a development aid, use the command

`void` **ValidateProgramPipeline**(`uint` *pipeline*);

to validate the program pipeline object *pipeline* against the current GL state. Each program pipeline object has a boolean status, `VALIDATE_STATUS`, that is modified as a result of validation. This status may be queried with **GetProgramPipelineiv** (see section 7.14). If validation succeeded, no `INVALID_OPERATION` validation error is generated if *pipeline* is bound and no program is made current via **UseProgram**, given the current state. If validation failed, such errors are generated under the current state.

If *pipeline* is a name that has been generated (without subsequent deletion) by **GenProgramPipelines**, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when **BindProgramPipeline** creates a new program pipeline object.

Errors

An `INVALID_OPERATION` error is generated if *pipeline* is not a name returned from a previous call to **GenProgramPipelines** or if such a name has since been deleted by **DeleteProgramPipelines**.

11.1.3.13 Undefined Behavior

When using array, vector or matrix variables in a shader, it is possible to access a variable with an index computed at run time that is outside the declared extent of the variable. Such out-of-bounds accesses have undefined behavior, and system errors (possibly including program termination) may occur. The level of protection provided against such errors in the shader is implementation-dependent.

Robust buffer access can be enabled by creating a context with robust access enabled through the window system binding APIs. When enabled, out-of-bounds accesses will be bounded within the working memory of the active program, cannot access memory owned by other GL contexts, and will not result in abnormal program termination. Out-of-bounds access to local and global variables cannot read values from other program invocations. An out-of-bounds read may return another value from the active program's working memory or zero. An out-of-bounds write may overwrite a value from the active program's working memory or be discarded.

Out-of-bounds accesses to resources backed by buffer objects cannot read or modify data outside of the buffer object. For resources bound to buffer ranges, access is restricted within the buffer object from which the buffer range was created, and not within the buffer range itself.

Out-of-bounds reads may return any of the following values:

- Values from anywhere within the buffer object.
- Zero values, or $(0, 0, 0, x)$ vectors for vector reads where x is a valid value represented in the type of the vector components and may be any of
  - Zero, one, or the maximum representable positive integer value, for signed or unsigned integer components.
  - 0.0 or 1.0, for floating-point components.

Out-of-bounds writes may modify values within the buffer object or be discarded.

Out-of-bounds accesses to arrays of resources, such as an array of textures, can only access the data of bound resources. Reads from unbound resources return zero and writes are discarded. It is not possible to access data owned by other GL contexts.

Applications that require defined behavior for out-of-bounds accesses should range check all computed indices before dereferencing the array, vector or matrix.

11.1.4 Vertex Shader Queries

Vertex shader queries use query objects to track the number of vertex shader invocations.

When **BeginQuery** is called with a target of `VERTEX_SHADER_INVOCATIONS`, the vertex shader invocations count maintained by the GL is set to zero. When a vertex shader invocations query is active, the counter is incremented every time the vertex shader is invoked (see section 11.1).

The result of vertex shader queries may be implementation dependent due to reasons described in section 11.1.3.

11.2 Tessellation

Tessellation is a process that reads a patch primitive and generates new primitives used by subsequent pipeline stages. The generated primitives are formed by subdividing a single triangle or quad primitive according to fixed or shader-computed levels of detail and transforming each of the vertices produced during this subdivision.

Tessellation functionality is controlled by two types of tessellation shaders: tessellation control shaders and tessellation evaluation shaders. Tessellation is considered active if and only if there is an active tessellation control or tessellation evaluation program object.

The tessellation control shader is used to read an input patch provided by the application, and emit an output patch. The tessellation control shader is run once for each vertex in the output patch and computes the attributes of that vertex. Additionally, the tessellation control shader may compute additional per-patch attributes of the output patch. The most important per-patch outputs are the tessellation levels, which are used to control the number of subdivisions performed by the tessellation primitive generator. The tessellation control shader may also write additional per-patch attributes for use by the tessellation evaluation shader. If no tessellation control shader is active, the patch provided is passed through to the tessellation primitive generator stage unmodified.

If a tessellation evaluation shader is active, the tessellation primitive generator subdivides a triangle or quad primitive into a collection of points, lines, or triangles according to the tessellation levels of the patch and the set of `layout` declarations specified in the tessellation evaluation shader text. The tessellation levels used to control subdivision are normally written by the tessellation control shader. If no tessellation control shader is active, default tessellation levels are instead used.

When a tessellation evaluation shader is active, it is run on each vertex generated by the tessellation primitive generator to compute the final position and other attributes of the vertex. The tessellation evaluation shader can read the relative location of the vertex in the subdivided output primitive, given by an (u, v) or (u, v, w) coordinate, as well as the position and attributes of any or all of the vertices in the input patch.

Tessellation operates only on patch primitives. Patch primitives are not supported by pipeline stages below the tessellation evaluation shader.

A non-separable program object or program pipeline object that includes a tessellation shader of any kind must also include a vertex shader.

Errors

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if tessellation is active and the primitive mode is not `PATCHES`.

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if the primitive *mode* is `PATCHES` and there is no active tessellation evaluation program.

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if the current program state has a tessellation shader but no vertex shader.

11.2.1 Tessellation Control Shaders

The tessellation control shader consumes an input patch provided by the application and emits a new output patch. The input patch is an array of vertices with attributes corresponding to output variables written by the vertex shader. The output patch consists of an array of vertices with attributes corresponding to per-vertex output variables written by the tessellation control shader and a set of per-patch attributes corresponding to per-patch output variables written by the tessellation control shader. Tessellation control output variables are per-vertex by default, but may be declared as per-patch using the `patch` qualifier.

The number of vertices in the output patch is fixed when the program is linked, and is specified in tessellation control shader source code using the output `layout` qualifier `vertices`, as described in the OpenGL Shading Language Specification. A program will fail to link if the output patch vertex count is not specified by any tessellation control shader object attached to the program, if it is specified differently by multiple tessellation control shader objects, if it is less than or equal to zero, or if it is greater than the implementation-dependent maximum patch size. The output patch vertex count may be queried by calling **GetProgramiv** with *pname* `TESS_CONTROL_OUTPUT_VERTICES`.

Tessellation control shaders are created as described in section 7.1, using a *type* of `TESS_CONTROL_SHADER`. When a new input patch is received, the tessellation control shader is run once for each vertex in the output patch. The tessellation control shader invocations collectively specify the per-vertex and per-patch attributes of the output patch. The per-vertex attributes are obtained from the per-vertex output variables written by each invocation. Each tessellation control shader invocation may only write to per-vertex output variables corresponding to its own output patch vertex. The output patch vertex number corresponding to a given tessellation control shader invocation is given by the built-in variable `gl_InvocationID`. Per-patch attributes are taken from the per-patch output variables, which may be written by any tessellation control shader invocation. While tessellation control shader invocations may read any per-vertex and per-patch output variable and write any per-patch output variable, reading or writing output variables also written by other invocations has ordering hazards discussed below.

11.2.1.1 Tessellation Control Shader Variables

Tessellation control shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Tessellation control shaders also have access to samplers to perform texturing

operations, as described in section 7.11.

Tessellation control shaders can access the transformed attributes of all vertices for their input primitive using *input* variables. A vertex shader writing to output variables generates the values of these input variables. Values for any inputs that are not written by a vertex shader are undefined.

Additionally, tessellation control shaders can write to one or more *output* variables including per-vertex attributes for the vertices of the output patch and per-patch attributes of the patch. Tessellation control shaders can also write to a set of built-in per-vertex and per-patch outputs defined in the OpenGL Shading Language. The per-vertex and per-patch attributes of the output patch are used by the tessellation primitive generator (section 11.2.2) and may be read by a tessellation evaluation shader (section 11.2.3).

11.2.1.2 Tessellation Control Shader Execution Environment

If there is an active program for the tessellation control stage, the executable version of the program's tessellation control shader is used to process patches resulting from the primitive assembly stage. When tessellation control shader execution completes, the input patch is consumed. A new patch is assembled from the per-vertex and per-patch output variables written by the shader and is passed to subsequent pipeline stages.

There are several special considerations for tessellation control shader execution described in the following sections.

11.2.1.2.1 Texture Access Section 11.1.3.1 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to tessellation control shaders.

11.2.1.2.2 Tessellation Control Shader Inputs Section 7.1("Built-In Variables") of the OpenGL Shading Language Specification describes the built-in variable array `gl_in` available as input to a tessellation control shader. `gl_in` receives values from equivalent built-in output variables written by the vertex shader (section 11.1.3). Each array element of `gl_in` is a structure holding values for a specific vertex of the input patch. The length of `gl_in` is equal to the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). Behavior is undefined if `gl_in` is indexed with a vertex index greater than or equal to the current patch size. The members of each element of the `gl_in` array are `gl_Position`, `gl_PointSize`, `gl_CullDistance`, `gl_ClipDistance`, `gl_`

ClipVertex, gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, gl_BackSecondaryColor, gl_TexCoord, and gl_FogFragCoord.

Tessellation control shaders have available several other built-in input variables not replicated per-vertex and not contained in gl_in, including:

- The variable gl_PatchVerticesIn holds the number of vertices in the input patch being processed by the tessellation control shader.
- The variable gl_PrimitiveID is filled with the number of primitives processed since the last time **Begin** was called (directly or indirectly via vertex array functions). The first primitive generated after a **Begin** is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.
- The variable gl_InvocationID holds an invocation number for the current tessellation control shader invocation. Tessellation control shaders are invoked once per output patch vertex, and invocations are numbered beginning with zero.

Similarly to the built-in inputs, each user-defined input variable has a value for each vertex and thus needs to be declared as an array, or inside an input block declared as an array. Declaring an array size is optional. If no size is specified, it will be taken from the implementation-dependent maximum patch size (gl_MaxPatchVertices). If a size is specified, it must match the maximum patch size; otherwise, a compile or link error will occur. Since the array size may be larger than the number of vertices found in the input patch, behavior is undefined if a per-vertex input variable is accessed using an index greater than or equal to the number of vertices in the input patch.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of input variables that can be read by the tessellation control shader, given by the value of the implementation-dependent constant MAX_TESS_CONTROL_INPUT_COMPONENTS.

When a program is linked, all components of any input variable read by a tessellation control shader will count against this limit. A program whose tessellation control shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for MAX_VERTEX_OUTPUT_COMPONENTS (see section 11.1.2.1).

11.2.1.2.3 Tessellation Control Shader Outputs Section 7.1("Built-In Variables") of the OpenGL Shading Language Specification describes the built-in variable array `gl_out` available as an output for a tessellation control shader. `gl_out` passes values to equivalent built-in input variables read by subsequent shader stages or to subsequent fixed functionality vertex processing pipeline stages. Each array element of `gl_out` is a structure holding values for a specific vertex of the output patch. The length of `gl_out` is equal to the output patch size specified in the tessellation control shader output `layout` declaration. The members of each element of the `gl_out` array are `gl_Position`, `gl_PointSize`, `gl_ClipDistance`, `gl_ClipVertex`, `gl_CullDistance`, `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor`, `gl_BackSecondaryColor`, `gl_TexCoord`, and `gl_FogFragCoord`, and behave identically to equivalently named vertex shader outputs (section 11.1.3).

Tessellation control shaders additionally have two built-in per-patch output arrays, `gl_TessLevelOuter` and `gl_TessLevelInner`. These arrays are not replicated for each output patch vertex and are not members of `gl_out`. `gl_TessLevelOuter` is an array of four floating-point values specifying the approximate number of segments that the tessellation primitive generator should use when subdividing each outer edge of the primitive it subdivides. `gl_TessLevelInner` is an array of two floating-point values specifying the approximate number of segments used to produce a regularly-subdivided primitive interior. The values written to `gl_TessLevelOuter` and `gl_TessLevelInner` need not be integers, and their interpretation depends on the type of primitive the tessellation primitive generator will subdivide and other tessellation parameters, as discussed in the following section.

A tessellation control shader may also declare user-defined per-vertex output variables. User-defined per-vertex output variables are declared with the qualifier `out` and have a value for each vertex in the output patch. Such variables must be declared as arrays or inside output blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be taken from the output patch size declared in the shader. If a size is specified, it must match the maximum patch size; otherwise, a compile or link error will occur.

While per-vertex output variables are declared as arrays indexed by vertex number, each tessellation control shader invocation may write only to those outputs corresponding to its output patch vertex. Tessellation control shaders must use the input variable `gl_InvocationID` as the vertex number index when writing to per-vertex output variables.

Additionally, a tessellation control shader may declare per-patch output variables using the qualifier `patch out`. Unlike per-vertex outputs, per-patch outputs do not correspond to any specific vertex in the patch, and are not indexed by vertex

number. Per-patch outputs declared as arrays have multiple values for the output patch; similarly declared per-vertex outputs would indicate a single value for each vertex in the output patch. User-defined per-patch outputs are not used by the tessellation primitive generator, but may be read by tessellation evaluation shaders.

There are several limits on the number of components of output variables that can be written by the tessellation control shader. The number of components of active per-vertex output variables may not exceed the value of `MAX_TESS_CONTROL_OUTPUT_COMPONENTS`. The number of components of active per-patch output variables may not exceed the value of `MAX_TESS_PATCH_COMPONENTS`. The built-in outputs `gl_TessLevelOuter` and `gl_TessLevelInner` are not counted against the per-patch limit. The total number of components of active per-vertex and per-patch outputs is derived by multiplying the per-vertex output component count by the output patch size and then adding the per-patch output component count. The total component count may not exceed `MAX_TESS_CONTROL_TOTAL_OUTPUT_COMPONENTS`.

When a program is linked, all components of any output variable written by a tessellation control shader will count against this limit. A program exceeding any of these limits may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS` (see section 11.1.2.1).

11.2.1.2.4 Tessellation Control Shader Execution Order For tessellation control shaders with a declared output patch size greater than one, the shader is invoked more than once for each input patch. The order of execution of one tessellation control shader invocation relative to the other invocations for the same input patch is largely undefined. The built-in function `barrier` provides some control over relative execution order. When a tessellation control shader calls the `barrier` function, its execution pauses until all other invocations have also called the same function. Output variable assignments performed by any invocation executed prior to calling `barrier` will be visible to any other invocation after the call to `barrier` returns. Shader output values read in one invocation but written by another may be undefined without proper use of `barrier`; full rules are found in the OpenGL Shading Language Specification.

The `barrier` function may only be called inside the main entry point of the tessellation control shader and may not be called in code containing potentially divergent flow of control. In particular, `barrier` may not be called inside a `switch` statement, in either sub-statement of an `if` statement, inside a `do`, `for`, or `while` loop, or at any point after a return statement in the function `main`.

11.2.2 Tessellation Primitive Generation

If a tessellation evaluation shader is present, the tessellation primitive generator consumes the input patch and produces a new set of basic primitives (points, lines, or triangles). These primitives are produced by subdividing a geometric primitive (rectangle or triangle) according to the per-patch tessellation levels written by the tessellation control shader, if present, or taken from default patch parameter values. This subdivision is performed in an implementation-dependent manner. If no tessellation evaluation shader is present, the tessellation primitive generator passes incoming primitives through without modification.

The type of subdivision performed by the tessellation primitive generator is specified by an input `layout` declaration in the tessellation evaluation shader using one of the identifiers `triangles`, `quads`, and `isolines`. For `triangles`, the primitive generator subdivides a triangle primitive into smaller triangles. For `quads`, the primitive generator subdivides a rectangle primitive into smaller triangles. For `isolines`, the primitive generator subdivides a rectangle primitive into a collection of line segments arranged in strips stretching horizontally across the rectangle. Each vertex produced by the primitive generator has an associated (u, v, w) or (u, v) position in a normalized parameter space, with parameter values in the range $[0, 1]$, as illustrated in figure 11.1. For `triangles`, the vertex position is a barycentric coordinate (u, v, w), where $u + v + w = 1$, and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For `quads` and `isolines`, the position is a (u, v) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

When no tessellation control shader is present, the tessellation levels are taken from default patch tessellation levels. These default levels are set by calling

```
void PatchParameterfv( enum pname, const
    float *values );
```

If *pname* is `PATCH_DEFAULT_OUTER_LEVEL`, *values* specifies an array of four floating-point values corresponding to the four outer tessellation levels for each subsequent patch. If *pname* is `PATCH_DEFAULT_INNER_LEVEL`, *values* specifies an array of two floating-point values corresponding to the two inner tessellation levels.

A patch is discarded by the tessellation primitive generator if any relevant outer tessellation level is less than or equal to zero. Patches will also be discarded if any relevant outer tessellation level corresponds to a floating-point NaN (not a number) in implementations supporting NaN. When patches are discarded, no new primitives will be generated and the tessellation evaluation program will not be run.

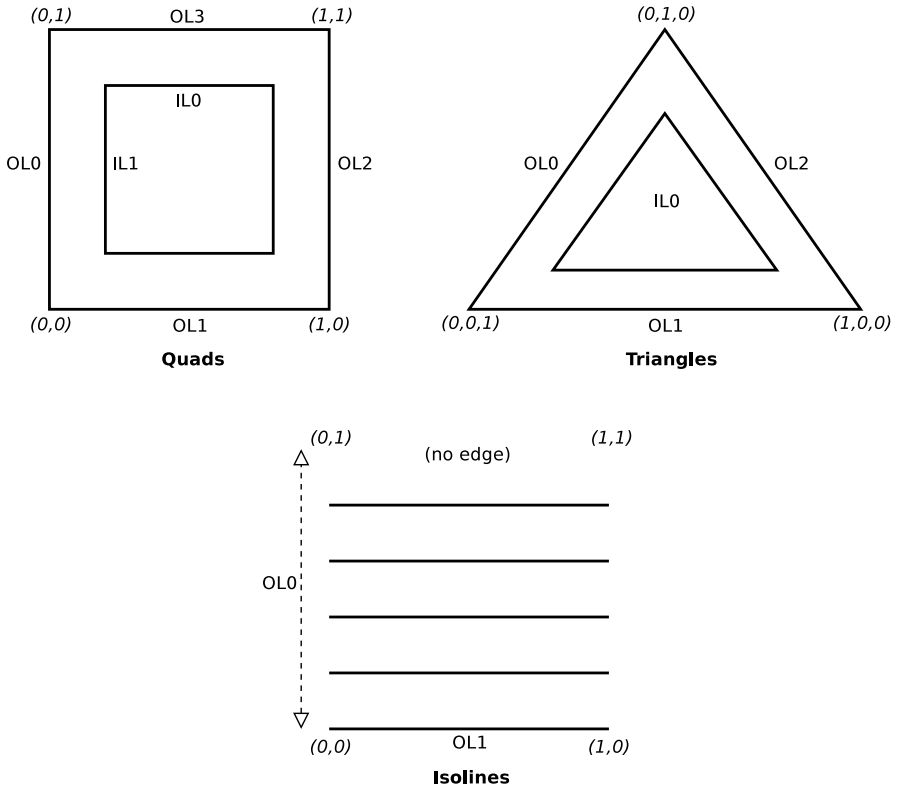


Figure 11.1. Domain parameterization for tessellation generator primitive modes (triangles, quads, or isolines). The coordinates illustrate the value of `gl_TessCoord` at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.

For quads, all four outer levels are relevant. For triangles and isolines, only the first three or two outer levels, respectively, are relevant. Negative inner levels will not cause a patch to be discarded; they will be clamped as described below.

Each of the tessellation levels is used to determine the number and spacing of segments used to subdivide a corresponding edge. The method used to derive the number and spacing of segments is specified by an input layout declaration in the tessellation evaluation shader using one of the identifiers equal_spacing, fractional_even_spacing, or fractional_odd_spacing. If no spacing is specified in the tessellation evaluation shader, equal_spacing will be used.

If equal_spacing is used, the floating-point tessellation level is first clamped to the range $[1, max]$, where *max* is the implementation-dependent maximum tessellation level (the value of MAX_TESS_GEN_LEVEL). The result is rounded up to the nearest integer n, and the corresponding edge is divided into n segments of equal length in (u, v) space.

If fractional_even_spacing is used, the tessellation level is first clamped to the range $[2, max]$ and then rounded up to the nearest even integer n. If fractional_odd_spacing is used, the tessellation level is clamped to the range $[1, max - 1]$ and then rounded up to the nearest odd integer n. If n is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into $n - 2$ segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with the value of $n - f$, where f is the clamped floating-point tessellation level. When $n - f$ is zero, the additional segments will have equal length to the other segments. As $n - f$ approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments should be placed symmetrically on opposite sides of the subdivided edge. The relative location of these two segments is undefined, but must be identical for any pair of subdivided edges with identical values of f.

When the tessellation primitive generator produces triangles (in the triangles or quads modes), the orientation of all triangles can be specified by an input layout declaration in the tessellation evaluation shader using the identifiers cw and ccw. If the order is cw, the vertices of all generated triangles will have a clockwise ordering in (u, v) or (u, v, w) space, as illustrated in figure 11.1. If the order is ccw, the vertices will be specified in counter-clockwise order. If no layout is specified, ccw will be used.

For all primitive modes, the tessellation primitive generator is capable of generating points instead of lines or triangles. If an input layout declaration in the tessellation evaluation shader specifies the identifier point_mode, the primitive generator will generate one point for each distinct vertex produced by tessellation. Otherwise, the primitive generator will produce a collection of line segments or

triangles according to the primitive mode. When tessellating triangles or quads in point mode with fractional odd spacing, the tessellation primitive generator may produce "interior" vertices that are positioned on the edge of the patch if an inner tessellation level is less than or equal to one. Such vertices are considered distinct from vertices produced by subdividing the outer edge of the patch, even if there are pairs of vertices with identical coordinates.

The points, lines, or triangles produced by the tessellation primitive generator are passed to subsequent pipeline stages in an implementation-dependent order.

Errors

An `INVALID_ENUM` error is generated if *pname* is not `PATCH_DEFAULT_OUTER_LEVEL` or `PATCH_DEFAULT_INNER_LEVEL`.

11.2.2.1 Triangle Tessellation

If the tessellation primitive mode is `triangles`, an equilateral triangle is subdivided into a collection of triangles covering the area of the original triangle. First, the original triangle is subdivided into a collection of concentric equilateral triangles. The edges of each of these triangles are subdivided, and the area between each triangle pair is filled by triangles produced by joining the vertices on the subdivided edges. The number of concentric triangles and the number of subdivisions along each triangle except the outermost is derived from the first inner tessellation level. The edges of the outermost triangle are subdivided independently, using the first, second, and third outer tessellation levels to control the number of subdivisions of the $u = 0$ (left), $v = 0$ (bottom), and $w = 0$ (right) edges, respectively. The second inner tessellation level and the fourth outer tessellation level have no effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with (u, v, w) coordinates of $(0, 0, 1)$, $(1, 0, 0)$, and $(0, 1, 0)$ is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as $1 + \epsilon$ and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce "inner" vertices positioned on the edge of the triangle.

If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating n segments. For

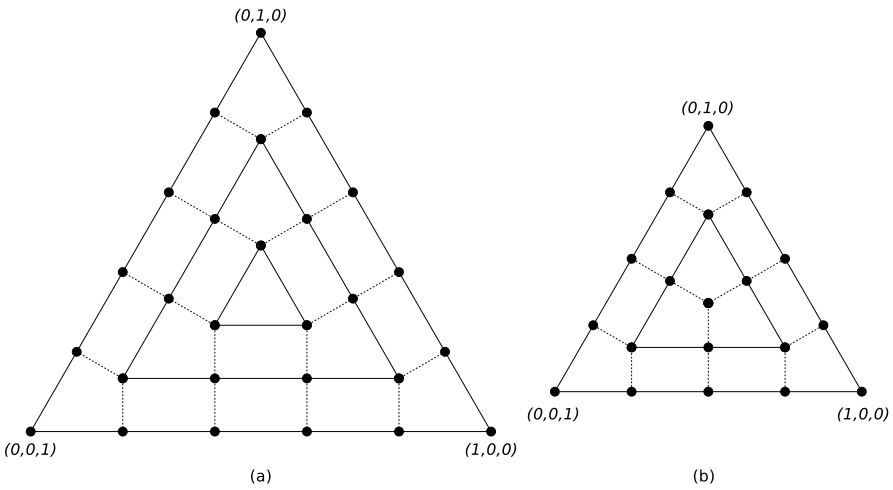


Figure 11.2. Inner triangle tessellation with inner tessellation levels of (a) five and (b) four, respectively (not to scale). Solid black circles depict vertices along the edges of the concentric triangles. The edges of inner triangles are subdivided by intersecting the edge with segments perpendicular to the edge passing through each inner vertex of the subdivided outer edge. Dotted lines depict edges connecting corresponding vertices on the inner and outer triangle edges.

the outermost inner triangle, the inner triangle is degenerate – a single point at the center of the triangle – if n is two. Otherwise, for each corner of the outer triangle, an inner triangle corner is produced at the intersection of two lines extended perpendicular to the corner's two adjacent edges running through the vertex of the subdivided outer edge nearest that corner. If n is three, the edges of the inner triangle are not subdivided and it is the final triangle in the set of concentric triangles. Otherwise, each edge of the inner triangle is divided into $n - 2$ segments, with the $n - 1$ vertices of this subdivision produced by intersecting the inner edge with lines perpendicular to the edge running through the $n - 1$ innermost vertices of the subdivision of the outer edge. Once the outermost inner triangle is subdivided, the previous subdivision process repeats itself, using the generated triangle as an outer triangle. This subdivision process is illustrated in figure 11.2.

Once all the concentric triangles are produced and their edges are subdivided, the area between each pair of adjacent inner triangles is filled completely with a set of non-overlapping triangles. In this subdivision, two of the three vertices of

each triangle are taken from adjacent vertices on a subdivided edge of one triangle; the third is one of the vertices on the corresponding edge of the other triangle. If the innermost triangle is degenerate (i.e., a point), the triangle containing it is subdivided into six triangles by connecting each of the six vertices on that triangle with the center point. If the innermost triangle is not degenerate, that triangle is added to the set of generated triangles as-is.

After the area corresponding to any inner triangles is filled, the primitive generator generates triangles to cover area between the outermost triangles and the outermost inner triangles. To do this, the temporary subdivision of the outer triangle edges above is discarded. Instead, the $u = 0$, $v = 0$, and $w = 0$ edges are subdivided according to the first, second, and third outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the first inner triangle is retained. The area between the outer and first inner triangles is completely filled by non-overlapping triangles as described above. If the first (and only) inner triangle is degenerate, a set of triangles is produced by connecting each vertex on the outer triangle edges with the center point.

After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric (u, v, w) coordinate based on its location relative to the three vertices of the outer triangle.

The algorithm used to subdivide the triangular domain in (u, v, w) space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles. The order in which the generated triangles are passed to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to figure 11.2, the order of the vertices in the generated triangles will be either all clockwise or all counter-clockwise, according to the vertex order `layout` declaration.

11.2.2.2 Quad Tessellation

If the tessellation primitive mode is `quads`, a rectangle is subdivided into a collection of triangles covering the area of the original rectangle. First, the original rectangle is subdivided into a regular mesh of rectangles, where the number of rectangles along the $u = 0$ and $u = 1$ (vertical) and $v = 0$ and $v = 1$ (horizontal) edges are derived from the first and second inner tessellation levels, respectively. All rectangles, except those adjacent to one of the outer rectangle edges, are decomposed into triangle pairs. The outermost rectangle edges are subdivided independently, using the first, second, third, and fourth outer tessellation levels to control the number of subdivisions of the $u = 0$ (left), $v = 0$ (bottom), $u = 1$

(right), and $v = 1$ (top) edges, respectively. The area between the inner rectangles of the mesh and the outer rectangle edges is filled by triangles produced by joining the vertices on the subdivided outer edges to the vertices on the edges of the inner rectangle mesh.

If both clamped inner tessellation levels and all four clamped outer tessellation levels are exactly one, only a single triangle pair covering the outer rectangle is generated. Otherwise, if either clamped inner tessellation level is one, that tessellation level is treated as though it were originally specified as $1 + \epsilon$, and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce "inner" vertices positioned on the edge of the rectangle.

If any tessellation level is greater than one, tessellation begins by subdividing the $u = 0$ and $u = 1$ edges of the outer rectangle into m segments using the clamped and rounded first inner tessellation level and the tessellation spacing. The $v = 0$ and $v = 1$ edges are subdivided into n segments using the second inner tessellation level. Each vertex on the $u = 0$ and $v = 0$ edges is joined with the corresponding vertex on the $u = 1$ and $v = 1$ edges to produce a set of vertical and horizontal lines that divide the rectangle into a grid of smaller rectangles. The primitive generator emits a pair of non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the edges of which are subdivided by the grid vertices that lie on the edge. If either m or n is two, the inner rectangle is degenerate, and one or both of the rectangle's "edges" consist of a single point. This subdivision is illustrated in figure 11.3.

After the area corresponding to the inner rectangle is filled, the primitive generator must produce triangles to cover area between the inner and outer rectangles. To do this, the subdivision of the outer rectangle edges above is discarded. Instead, the $u = 0$, $v = 0$, $u = 1$, and $v = 1$ edges are subdivided according to the first, second, third, and fourth outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the inner rectangle is retained. The area between the outer and inner rectangles is completely filled by non-overlapping triangles. Two of the three vertices of each triangle are adjacent vertices on a subdivided edge of one rectangle; the third is one of the vertices on the corresponding edge of the other rectangle. If either edge of the innermost rectangle is degenerate, the area near the corresponding outer edge is filled by connecting each vertex on the outer edge with the single vertex making up the inner "edge".

The algorithm used to subdivide the rectangular domain in (u, v) space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles. The order in which the generated triangles are

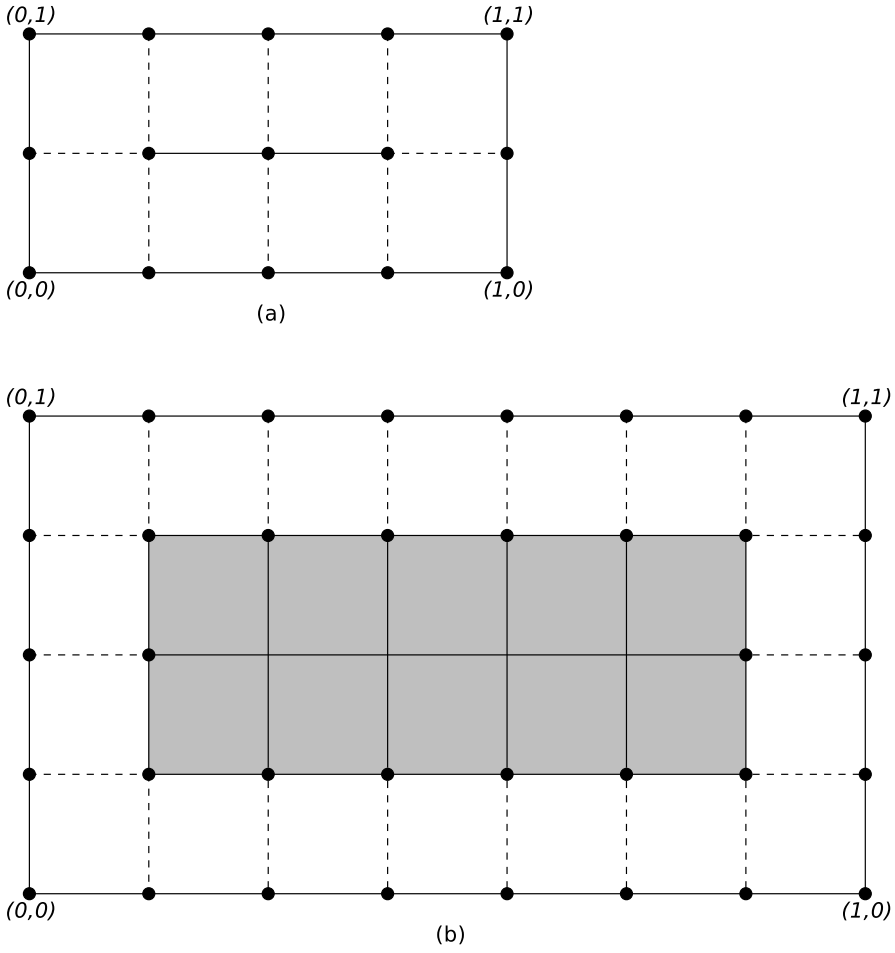


Figure 11.3. Inner quad tessellation with inner tessellation levels of (a) $(4, 2)$ and (b) $(7, 4)$, respectively. Gray regions on the bottom figure depict the 10 inner rectangles, each of which will be subdivided into two triangles. Solid black circles depict vertices on the boundary of the outer and inner rectangles, where the inner rectangle on the top figure is degenerate (a single line segment). Dotted lines depict the horizontal and vertical edges connecting corresponding vertices on the inner and outer rectangle edges.

passed to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to figure 11.3, the order of the vertices in the generated triangles will be either all clockwise or all counter-clockwise, according to the vertex order `layout` declaration.

11.2.2.3 Isoline Tessellation

If the tessellation primitive mode is `isolines`, a set of independent horizontal line segments is drawn. The segments are arranged into connected strips called *isolines*, where the vertices of each isoline have a constant v coordinate and u coordinates covering the full range $[0, 1]$. The number of isolines generated is derived from the first outer tessellation level; the number of segments in each isoline is derived from the second outer tessellation level. Both inner tessellation levels and the third and fourth outer tessellation levels have no effect in this mode.

As with quad tessellation above, isoline tessellation begins with a rectangle. The $u = 0$ and $u = 1$ edges of the rectangle are subdivided according to the first outer tessellation level. For the purposes of this subdivision, the tessellation spacing is ignored and treated as `equal_spacing`. An isoline is drawn connecting each vertex on the $u = 0$ rectangle edge with the corresponding vertex on the $u = 1$ rectangle edge, except that no line is drawn between $(0, 1)$ and $(1, 1)$. If the number of isolines on the subdivided $u = 0$ and $u = 1$ edges is n, this process will result in n equally spaced lines with constant v coordinates of $0, \frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$.

Each of the n isolines is then subdivided according to the second outer tessellation level and the tessellation spacing, resulting in m line segments. Each segment of each line is emitted by the tessellation primitive generator, as illustrated in figure 11.4.

The order in which the generated line segments are passed to subsequent pipeline stages and the order of the vertices in each generated line segment are both implementation-dependent.

11.2.3 Tessellation Evaluation Shaders

If active, the tessellation evaluation shader takes the (u, v) or (u, v, w) location of each vertex in the primitive subdivided by the tessellation primitive generator, and generates a vertex with a position and associated attributes. The tessellation evaluation shader can read any of the vertices of its input patch, which is the output patch produced by the tessellation control shader (if present) or provided by the application and transformed by the vertex shader (if no control shader is used). Evaluating the bivariate polynomials described in section 21.1 using the vertices

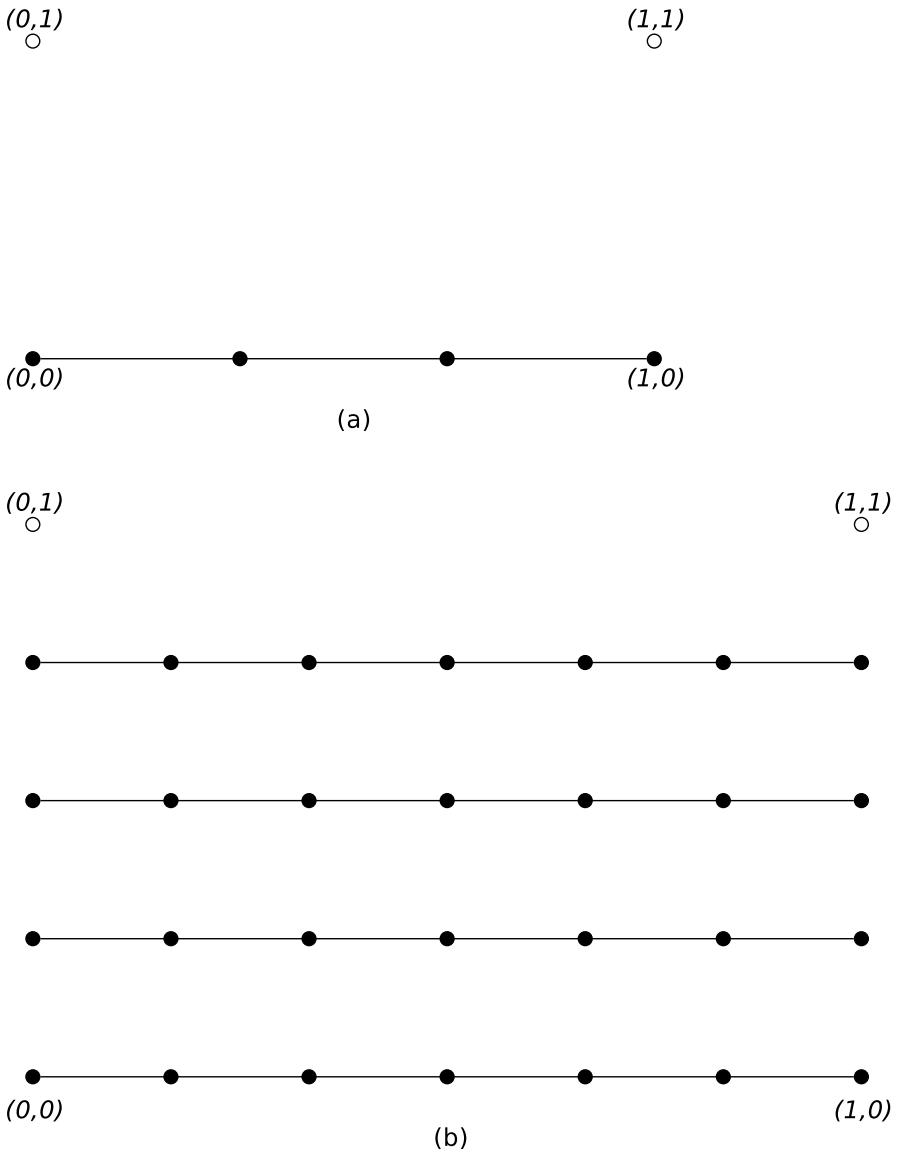


Figure 11.4. Isoline tessellation with the first two outer tessellation levels of (a) $(1, 3)$ and (b) $(4, 6)$, respectively. Line segments connecting the vertices marked with solid black circles are emitted by the primitive generator. Vertices marked with empty circles correspond to (u, v) coordinates of $(0, 1)$ and $(1, 1)$, where no line segments are generated.

of the provided patch as control points is one example of the type of computations that a tessellation evaluation shader might be expected to perform. Tessellation evaluation shaders are created as described in section 7.1, using a *type* of `TESS_EVALUATION_SHADER`.

Each invocation of the tessellation evaluation shader writes the attributes of exactly one vertex. The number of vertices evaluated per patch depends on the tessellation level values computed by the tessellation control shaders (if present) or specified as patch parameters. Tessellation evaluation shader invocations run independently, and no invocation can access the variables belonging to another invocation. All invocations are capable of accessing all the vertices of their corresponding input patch.

If a tessellation control shader is present, the number of the vertices in the input patch is fixed and is equal to the tessellation control shader output patch size parameter in effect when the program was last linked. If no tessellation control shader is present, the input patch is provided by the application and can have a variable number of vertices, as specified by **PatchParameteri**.

11.2.3.1 Tessellation Evaluation Shader Variables

Tessellation evaluation shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Tessellation evaluation shaders also have access to samplers to perform texturing operations, as described in section 7.11.

Tessellation evaluation shaders can access the transformed attributes of all vertices for their input primitive using input variables. If active, a tessellation control shader writing to output variables generates the values of these input variables. If no tessellation control shader is active, input variables will be obtained from vertex shader outputs. Values for any input variables that are not written by a vertex or tessellation control shader are undefined.

Additionally, tessellation evaluation shaders can write to one or more output variables that will be passed to subsequent programmable shader stages or fixed functionality vertex pipeline stages.

11.2.3.2 Tessellation Evaluation Shader Execution Environment

If there is an active program for the tessellation evaluation stage, the executable version of the program's tessellation evaluation shader is used to process vertices produced by the tessellation primitive generator. During this processing, the shader may access the input patch processed by the primitive generator. When tessellation

evaluation shader execution completes, a new vertex is assembled from the output variables written by the shader and is passed to subsequent pipeline stages.

There are several special considerations for tessellation evaluation shader execution described in the following sections.

11.2.3.2.1 Texture Access Section 11.1.3.1 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to tessellation evaluation shaders.

11.2.3.3 Tessellation Evaluation Shader Inputs

Section 7.1("Built-In Variables") of the OpenGL Shading Language Specification describes the built-in variable array `gl_in` available as input to a tessellation evaluation shader. `gl_in` receives values from equivalent built-in output variables written by a previous shader (section 11.1.3). If a tessellation control shader is active, the values of `gl_in` will be taken from tessellation control shader outputs. Otherwise, they will be taken from vertex shader outputs. Each array element of `gl_in` is a structure holding values for a specific vertex of the input patch. The length of `gl_in` is equal to the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). Behavior is undefined if `gl_in` is indexed with a vertex index greater than or equal to the current patch size. The members of each element of the `gl_in` array are `gl_Position`, `gl_PointSize`, `gl_ClipDistance`, `gl_ClipVertex`, `gl_CullDistance`, `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor`, `gl_BackSecondaryColor`, `gl_TexCoord`, and `gl_FogFragCoord`.

Tessellation evaluation shaders have available several other built-in input variables not replicated per-vertex and not contained in `gl_in`, including:

- The variables `gl_PatchVerticesIn` and `gl_PrimitiveID` are filled with the number of the vertices in the input patch and a primitive number, respectively. They behave exactly as the identically named inputs for tessellation control shaders.
- The variable `gl_TessCoord` is a three-component floating-point vector consisting of the (u, v, w) coordinate of the vertex being processed by the tessellation evaluation shader. The values of u, v, and w are in the range $[0, 1]$, and vary linearly across the primitive being subdivided. For tessellation primitive modes of `quads` or `isolines`, the w value is always zero.

The (u, v, w) coordinates are generated by the tessellation primitive generator in a manner dependent on the primitive mode, as described in section 11.2.2. `gl_TessCoord` is not an array; it specifies the location of the vertex being processed by the tessellation evaluation shader, not of any vertex in the input patch.

- The variables `gl_TessLevelOuter` and `gl_TessLevelInner` are arrays holding outer and inner tessellation levels of the patch, as used by the tessellation primitive generator. If a tessellation control shader is active, the tessellation levels will be taken from the corresponding outputs of the tessellation control shader. Otherwise, the default levels provided as patch parameters are used. Tessellation level values loaded in these variables will be prior to the clamping and rounding operations performed by the primitive generator as described in section 11.2.2. For triangular tessellation, `gl_TessLevelOuter[3]` and `gl_TessLevelInner[1]` will be undefined. For isoline tessellation, `gl_TessLevelOuter[2]`, `gl_TessLevelOuter[3]`, and both values in `gl_TessLevelInner` are undefined.

A tessellation evaluation shader may also declare user-defined per-vertex input variables. User-defined per-vertex input variables are declared with the qualifier `in` and have a value for each vertex in the input patch. User-defined per-vertex input variables have a value for each vertex and thus need to be declared as arrays or inside input blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be taken from the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). If a size is specified, it must match the maximum patch size; otherwise, a compile or link error will occur. Since the array size may be larger than the number of vertices found in the input patch, behavior is undefined if a per-vertex input variable is accessed using an index greater than or equal to the number of vertices in the input patch.

Additionally, a tessellation evaluation shader may declare per-patch input variables using the qualifier `patch in`. Unlike per-vertex inputs, per-patch inputs do not correspond to any specific vertex in the patch, and are not indexed by vertex number. Per-patch inputs declared as arrays have multiple values for the input patch; similarly declared per-vertex inputs would indicate a single value for each vertex in the output patch. User-defined per-patch input variables are filled with corresponding per-patch output values written by the tessellation control shader. If no tessellation control shader is active, all such variables are undefined.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of per-vertex and

per-patch input variables that can be read by the tessellation evaluation shader, given by the values of the implementation-dependent constants MAX_TESS_EVALUATION_INPUT_COMPONENTS and MAX_TESS_PATCH_COMPONENTS, respectively. The built-in inputs gl_TessLevelOuter and gl_TessLevelInner are not counted against the per-patch limit.

When a program is linked, all components of any input variable read by a tessellation evaluation shader will count against this limit. A program whose tessellation evaluation shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for MAX_VERTEX_OUTPUT_COMPONENTS (see section 11.1.2.1).

11.2.3.4 Tessellation Evaluation Shader Outputs

Tessellation evaluation shaders have a number of built-in output variables used to pass values to equivalent built-in input variables read by subsequent shader stages or to subsequent fixed functionality vertex processing pipeline stages. These variables are gl_Position, gl_PointSize, gl_ClipDistance, gl_ClipVertex, gl_CullDistance, gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, gl_BackSecondaryColor, gl_TexCoord, and gl_FogFragCoord, and all behave identically to equivalently named vertex shader outputs (see section 11.1.3). A tessellation evaluation shader may also declare user-defined per-vertex output variables.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of output variables that can be written by the tessellation evaluation shader, given by the values of the implementation-dependent constant MAX_TESS_EVALUATION_OUTPUT_COMPONENTS.

When a program is linked, all components of any output variable written by a tessellation evaluation shader will count against this limit. A program whose tessellation evaluation shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for MAX_VERTEX_OUTPUT_COMPONENTS (see section 11.1.2.1).

11.2.4 Tessellation Shader Queries

Tessellation shader queries use query objects to track the number of tessellation control shader and tessellation evaluation shader invocations.

When **BeginQuery** is called with a target of TESS_CONTROL_SHADER_PATCHES, the tessellation control shader patches count maintained by the GL is set to zero. When a tessellation control shader patches query is active, the counter is incremented every time a patch is processed by the tessellation control shader stage (see section 11.2.1).

When **BeginQuery** is called with a target of TESS_EVALUATION_SHADER_INVOCATIONS, the tessellation evaluation shader invocations count maintained by the GL is set to zero. When a tessellation evaluation shader invocations query is active, the counter is incremented every time the tessellation evaluation shader is invoked (see section 11.2.3).

The result of tessellation shader queries may be implementation dependent due to reasons described in section 11.1.3.

11.3 Geometry Shaders

After vertices are processed, they are arranged into primitives, as described in section 10.1. This section describes optional *geometry shaders*, an additional pipeline stage defining operations to further process those primitives. Geometry shaders operate on a single primitive at a time and emit one or more output primitives, all of the same type, which are then processed like an equivalent OpenGL primitive specified by the application. The original primitive is discarded after geometry shader execution. The inputs available to a geometry shader are the transformed attributes of all the vertices that belong to the primitive. Additional *adjacency primitives* are available which also make the transformed attributes of neighboring vertices available to the shader. The results of the shader are a new set of transformed vertices, arranged into primitives by the shader.

The geometry shader pipeline stage is inserted after primitive assembly, prior to transform feedback (section 13.3).

A geometry shader only applies when the GL is in RGBA mode. Its operation in color index mode is undefined.

Geometry shaders are created as described in section 7.1 using a *type* of GEOMETRY_SHADER. They are attached to and used in program objects as described in section 7.3. When the program object currently in use includes a geometry shader, its geometry shader is considered active, and is used to process primitives. If the program object has no geometry shader, or no program object is in use, this stage is bypassed.

A non-separable program object or program pipeline object that includes a geometry shader must also include a vertex shader.

Errors

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL if the current program state has a geometry shader but no vertex shader.

11.3.1 Geometry Shader Input Primitives

A geometry shader can operate on one of five input primitive types. Depending on the input primitive type, one to six input vertices are available when the shader is executed. Each input primitive type supports a subset of the primitives provided by the GL.

Errors

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL if a geometry shader is active and the primitive *mode* parameter is incompatible with the input primitive type of the geometry shader of the active geometry program object, as discussed below. If a tessellation evaluation shader is not active, the *mode* parameter passed to drawing commands is used for purposes of this error check. Otherwise, the type of primitive emitted by that shader is used.

A geometry shader that accesses more input vertices than are available for a given input primitive type can be successfully compiled, because the input primitive type is not part of the shader object. However, a program object containing a shader object that accesses more input vertices than are available for the input primitive type of the program object will not link.

The input primitive type is specified in the geometry shader source code using an input layout qualifier, as described in the OpenGL Shading Language Specification. A program will fail to link if the input primitive type is not specified by any geometry shader object attached to the program, or if it is specified differently by multiple geometry shader objects. The input primitive type may be queried by calling **GetProgramiv** with *pname* GEOMETRY_INPUT_TYPE. The supported types and the corresponding OpenGL Shading Language input layout qualifier keywords are:

Points (points)

Geometry shaders that operate on points are valid only for the POINTS primitive type. There is only a single vertex available for each geometry shader invocation.

Lines (`lines`)

Geometry shaders that operate on line segments are valid only for the `LINES`, `LINE_STRIP`, and `LINE_LOOP` primitive types. There are two vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment. See also section 11.3.4.

Lines with Adjacency (`lines_adjacency`)

Geometry shaders that operate on line segments with adjacent vertices are valid only for the `LINES_ADJACENCY` and `LINE_STRIP_ADJACENCY` primitive types. There are four vertices available for each program invocation. The second vertex refers to attributes of the vertex at the beginning of the line segment and the third vertex refers to the vertex at the end of the line segment. The first and fourth vertices refer to the vertices adjacent to the beginning and end of the line segment, respectively.

Triangles (`triangles`)

Geometry shaders that operate on triangles are valid for the `TRIANGLES`, `TRIANGLE_STRIP` and `TRIANGLE_FAN` primitive types. There are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively.

Triangles with Adjacency (`triangles_adjacency`)

Geometry shaders that operate on triangles with adjacent vertices are valid for the `TRIANGLES_ADJACENCY` and `TRIANGLE_STRIP_ADJACENCY` primitive types. There are six vertices available for each program invocation. The first, third and fifth vertices refer to attributes of the first, second and third vertex of the triangle, respectively. The second, fourth and sixth vertices refer to attributes of the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

11.3.2 Geometry Shader Output Primitives

A geometry shader can generate primitives of one of three types. The supported output primitive types are points (`POINTS`), line strips (`LINE_STRIP`), and triangle strips (`TRIANGLE_STRIP`). The vertices output by the geometry shader are assembled into points, lines, or triangles based on the output primitive type in the manner described in section 10.7. The resulting primitives are then further processed as described in section 11.3.4. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, nothing is drawn. The number of

vertices output by the geometry shader is limited to a maximum count specified in the shader.

The output primitive type and maximum output vertex count are specified in the geometry shader source code using an output `layout` qualifier, as described in section 4.4.2.2("Geometry Outputs") of the OpenGL Shading Language Specification. A program will fail to link if either the output primitive type or maximum output vertex count are not specified by any geometry shader object attached to the program, or if they are specified differently by multiple geometry shader objects. The output primitive type and maximum output vertex count of a linked program may be queried by calling **GetProgramiv** with *pname*s `GEOMETRY_OUTPUT_TYPE` and `GEOMETRY_VERTICES_OUT`, respectively.

11.3.3 Geometry Shader Variables

Geometry shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Geometry shaders also have access to samplers to perform texturing operations, as described in section 7.11.

Geometry shaders can access the transformed attributes of all vertices for their input primitive type using input variables. A vertex or tessellation shader (the *upstream shader* for the geometry shader) writing to output variables generates the values of these input variables. Values for any inputs that are not written by a shader are undefined. Additionally, a geometry shader has access to a built-in input that holds the ID of the current primitive. This ID is generated by the primitive assembly stage preceding the geometry shader.

Additionally, geometry shaders can write to one or more output variables for each vertex they output. These values are optionally flatshaded (using the OpenGL Shading Language qualifier `flat`) and clipped, then the clipped values interpolated across the primitive (if not flatshaded). The results of these interpolations are available to the fragment shader, if one is active. Geometry shaders can also write to a set of built-in output variables defined in the OpenGL Shading Language, corresponding to the values required for fixed-function processing that occurs after geometry processing.

11.3.4 Geometry Shader Execution Environment

If there is an active program for the geometry stage, the executable version of the program's geometry shader is used to process primitives resulting from the primitive assembly stage.

There are several special considerations for geometry shader execution described in the following sections.

11.3.4.1 Texture Access

Section 11.1.3.1 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to geometry shaders.

11.3.4.2 Instanced Geometry Shaders

For each input primitive received by the geometry shader pipeline stage, the geometry shader may be run once or multiple times. The number of times a geometry shader should be executed for each input primitive may be specified using a `layout` qualifier in a geometry shader of a linked program. If the invocation count is not specified in any `layout` qualifier, the invocation count will be one.

Each separate geometry shader invocation is assigned a unique invocation number. For a geometry shader with N invocations, each input primitive spawns N invocations, numbered 0 through $N - 1$. The built-in input variable `gl_InvocationID` may be used by a geometry shader invocation to determine its invocation number.

When executing instanced geometry shaders, the output primitives generated from each input primitive are passed to subsequent pipeline stages using the shader invocation number to order the output. The first primitives received by the subsequent pipeline stages are those emitted by the shader invocation numbered zero, followed by those from the shader invocation numbered one, and so forth. Additionally, all output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.

11.3.4.3 Geometry Shader Vertex Streams

Geometry shaders may emit primitives to multiple independent vertex streams. Each vertex emitted by the geometry shader is directed at one of the vertex streams. As vertices are received on each stream, they are arranged into primitives of the type specified by the geometry shader output primitive type. The shading language built-in functions `EndPrimitive` and `EndStreamPrimitive` may be used to end the primitive being assembled on a given vertex stream and start a new empty primitive of the same type. If an implementation supports N vertex streams, the individual streams are numbered 0 through $N - 1$. There is no requirement on the

order of the streams to which vertices are emitted, and the number of vertices emitted to each stream may be completely independent, subject only to implementation-dependent output limits.

The primitives emitted to all vertex streams are passed to the transform feedback stage to be captured and written to buffer objects in the manner specified by the transform feedback state. The primitives emitted to all streams but stream zero are discarded after transform feedback. Primitives emitted to stream zero are passed to subsequent pipeline stages for clipping, rasterization, and subsequent fragment processing.

Geometry shaders that emit vertices to multiple vertex streams are currently limited to using only the `points` output primitive type. A program will fail to link if it includes a geometry shader that calls the `EmitStreamVertex` built-in function and has any other output primitive type parameter.

11.3.4.4 Geometry Shader Inputs

Section 7.1("Built-In Variables") of the OpenGL Shading Language Specification describes the built-in variable array `gl_in[]` available as input to a geometry shader. `gl_in[]` receives values from equivalent built-in output variables written by the upstream shader, and each array element of `gl_in[]` is a structure holding values for a specific vertex of the input primitive. The length of `gl_in[]` is determined by the geometry shader input primitive type (see section 11.3.1). The members of each element of the `gl_in[]` array are:

- Structure member `gl_ClipDistance[]` holds the per-vertex array of clip distances, as written by the upstream shader to the built-in output variable `gl_ClipDistance[]`.
- Structure member `gl_CullDistance[]` holds the per-vertex array of cull distances, as written by the upstream shader to the built-in output variable `gl_CullDistance[]`.
- Structure member `gl_ClipVertex` holds the per-vertex position in clip coordinates, as written by the upstream shader to the built-in output variable `gl_ClipVertex`.
- Structure members `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor` and `gl_BackSecondaryColor` hold the per-vertex front and back colors of the primary and secondary colors, as written by the upstream shader to the corresponding built-in output variables.

- Structure member `gl_FogFragCoord` holds the per-vertex fog coordinate, as written by the upstream shader to the built-in output variable `gl_FogFragCoord`.
- Structure member `gl_TexCoord[]` holds the per-vertex array of texture coordinates written by the upstream shader to the built-in output variable `gl_TexCoord[]`.
- Structure member `gl_PointSize` holds the per-vertex point size written by the upstream shader to the built-in output variable `gl_PointSize`. If the upstream shader does not write `gl_PointSize`, the value of `gl_PointSize` is undefined, regardless of the value of the enable `PROGRAM_POINT_SIZE`.
- Structure member `gl_Position` holds the per-vertex position written by the upstream shader to the built-in output variable `gl_Position`. Note that writing to `gl_Position` from either the upstream or geometry shader is optional (also see section 7.1("Built-In Variables") of the OpenGL Shading Language Specification).

Geometry shaders also have available the built-in input variable `gl_PrimitiveIDIn`, which is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed since the last time **Begin** was called (directly or indirectly via vertex array functions). The first primitive generated after a **Begin** is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. For triangles drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may eventually be drawn. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Similarly to the built-in inputs, each user-defined input has a value for each vertex and thus needs to be declared as an array or inside an input block declared as an array. Declaring an array size is optional. If no size is specified, it will be inferred by the linker from the input primitive type. If a size is specified, it must match the number of vertices for the input primitive type; otherwise, a link error will occur.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of input variables that can be read by the geometry shader, given by the value of the implementation-dependent constant `MAX_GEOMETRY_INPUT_COMPONENTS`.

When a program is linked, all components of any input read by a geometry shader will count against this limit. A program whose geometry shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for MAX_VERTEX_OUTPUT_COMPONENTS (see section 11.1.2.1).

11.3.4.5 Geometry Shader Outputs

A geometry shader is limited in the number of vertices it may emit per invocation. The maximum number of vertices a geometry shader can possibly emit is specified in the geometry shader source and may be queried after linking by calling **GetProgramiv** with *pname* GEOMETRY_VERTICES_OUT. If a single invocation of a geometry shader emits more vertices than this value, the emitted vertices may have no effect.

There are two implementation-dependent limits on the value of GEOMETRY_VERTICES_OUT; it may not exceed the value of MAX_GEOMETRY_OUTPUT_VERTICES, and the product of the total number of vertices and the sum of all components of all active output variables may not exceed the value of MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS. **LinkProgram** will fail if it determines that the total component limit would be violated.

A geometry shader can write to built-in as well as user-defined output variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. To enable seamlessly inserting or removing a geometry shader from a program object, the rules, names and types of the built-in and user-defined output variables are the same as for the vertex shader. Refer to section 11.1.2.1, and to sections 4.3("Storage Qualifiers") and 7.1("Built-In Variables") of the OpenGL Shading Language Specification for more detail.

After a geometry shader emits a vertex, all output variables are undefined, as described in section 8.15("Geometry Shader Functions") of the OpenGL Shading Language Specification.

The built-in outputs gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor, and gl_BackSecondaryColor hold the front and back colors for the primary and secondary colors for the current vertex.

The built-in output gl_TexCoord[] is an array and holds the set of texture coordinates for the current vertex.

The built-in output gl_FogFragCoord is used as the c value, as described in section 16.4.

The built-in output gl_Position is intended to hold the homogeneous vertex position. Writing gl_Position is optional.

The built-in output gl_ClipVertex holds the vertex coordinate used in the clipping stage, as described in section 13.7.

The built-in outputs gl_ClipDistance and gl_CullDistance hold the clip distance and cull distance, respectively, used in the clipping stage, as described in section 13.7.

The built-in output gl_PointSize, if written, holds the size of the point to be rasterized, measured in pixels.

The built-in output gl_PrimitiveID holds the primitive ID counter read by the fragment shader, replacing the value of gl_PrimitiveID generated by drawing commands when no geometry shader is active. The geometry shader must write to gl_PrimitiveID for the provoking vertex (see section 13.6) of a primitive being generated, or the primitive ID counter read by the fragment shader for that primitive is undefined.

The built-in output gl_Layer is used in layered rendering, and discussed further in the next section.

The built-in output gl_ViewportIndex is used to direct rendering to one of several viewports and is discussed further in the next section.

Similarly to the limit on vertex shader output components (see section 11.1.2.1), there is a limit on the number of components of output variables that can be written by the geometry shader, given by the value of the implementation-dependent constant MAX_GEOMETRY_OUTPUT_COMPONENTS.

When a program is linked, all components of any output variable written by a geometry shader will count against this limit. A program whose geometry shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for MAX_VERTEX_OUTPUT_COMPONENTS (see section 11.1.2.1).

11.3.4.6 Layer and Viewport Selection

Geometry shaders can be used to render to one of several different layers of cube map textures, three-dimensional textures, or one- or two-dimensional texture arrays. This functionality allows an application to bind an entire complex texture to a framebuffer object, and render primitives to arbitrary layers computed at run time. For example, it can be used to project and render a scene onto all six faces of a cubemap texture in one pass. The layer to render to is specified by writing to the built-in output variable gl_Layer. Layered rendering requires the use of framebuffer objects (see section 9.8).

Geometry shaders may also select the destination viewport for each output primitive. The destination viewport for a primitive may be selected in the geometry shader by writing to the built-in output variable `gl_ViewportIndex`. This functionality allows a geometry shader to direct its output to a different viewport for each primitive, or to draw multiple versions of a primitive into several different viewports.

The specific vertex of a primitive that is used to select the rendering layer or viewport index is implementation-dependent and thus portable applications will assign the same layer and viewport index for all vertices in a primitive. The vertex conventions followed for `gl_Layer` and `gl_ViewportIndex` may be determined by calling **GetIntegerv** with *pname*s `LAYER_PROVOKING_VERTEX` and `VIEWPORT_INDEX_PROVOKING_VERTEX`, respectively. For either query, if the value returned is `PROVOKING_VERTEX`, then vertex selection follows the convention specified by **ProvokingVertex** (see section 13.6). If the value returned is `FIRST_VERTEX_CONVENTION`, selection is always taken from the first vertex of a primitive. If the value returned is `LAST_VERTEX_CONVENTION`, the selection is always taken from the last vertex of a primitive. If the value returned is `UNDEFINED_VERTEX`, the selection is not guaranteed to be taken from any specific vertex in the primitive. The vertex considered the provoking vertex for particular primitive types is given in table 13.2.

11.3.4.7 Primitive Type Mismatches and Drawing Commands

Errors

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL, and no fragments will be rendered, if a mismatch exists between the type of primitive being drawn and the input primitive type of a geometry shader. A mismatch exists under any of the following conditions:

- the input primitive type of the current geometry shader is `POINTS` and *mode* is not `POINTS`;
- the input primitive type of the current geometry shader is `LINES` and *mode* is not `LINES`, `LINE_STRIP`, or `LINE_LOOP`;
- the input primitive type of the current geometry shader is `TRIANGLES` and *mode* is not `TRIANGLES`, `TRIANGLE_STRIP` or `TRIANGLE_FAN`;
- the input primitive type of the current geometry shader is `LINES_ADJACENCY` and *mode* is not `LINES_ADJACENCY` or `LINE_STRIP_ADJACENCY`; or,

- the input primitive type of the current geometry shader is TRIANGLES_ADJACENCY and *mode* is not TRIANGLES_ADJACENCY or TRIANGLE_STRIP_ADJACENCY.

11.3.5 Geometry Shader Queries

Geometry shader queries use query objects to track the number of geometry shader invocations and the number of primitives those emitted.

When **BeginQuery** is called with a *target* of GEOMETRY_SHADER_INVOCATIONS, the geometry shader invocations count maintained by the GL is set to zero. When a geometry shader invocations query is active, the counter is incremented every time the geometry shader is invoked (see section 11.3). In case of instanced geometry shaders (see section 11.3.4.2), the geometry shader invocations count is incremented for each separate instanced invocation.

When **BeginQuery** is called with a *target* of GEOMETRY_SHADER_PRIMITIVES_EMITTED, the geometry shader output primitives count maintained by the GL is set to zero. When a geometry shader primitives emitted query is active, the counter is incremented every time the geometry shader emits a primitive to a vertex stream. Implementations may or may not count primitives emitted to a vertex stream that is not further processed by the GL (see section 11.3.2). Restarting primitive topology using the shading language built-in functions **EndPrimitive** or **EndStreamPrimitive** does not increment the geometry shader output primitives count.

The result of geometry shader queries may be implementation dependent due to reasons described in section 11.1.3.

Chapter 12

Fixed-Function Vertex Processing

When programmable vertex processing (see chapter 11) is not being performed, the fixed-function operations described in this chapter are performed instead. Vertices are first transformed as described in section 12.1, followed by lighting and coloring described described in section 12.2. The resulting transformed vertices are then processed as described in chapter 13.

12.1 Fixed-Function Vertex Transformations

Vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how this transformation is controlled.

Figure 12.1 diagrams the sequence of transformations that are applied to vertices. The vertex coordinates that are transferred to the GL are termed *object coordinates*. The *model-view* matrix is applied to these coordinates to yield *eye* coordinates. Then another matrix, called the *projection* matrix, is applied to eye coordinates to yield *clip* coordinates.

Object coordinates, eye coordinates, and clip coordinates are four-dimensional, consisting of x, y, z, and w coordinates (in that order). The model-view and projection matrices are thus 4×4.

If a vertex in object coordinates is given by $\begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$ and the model-view matrix

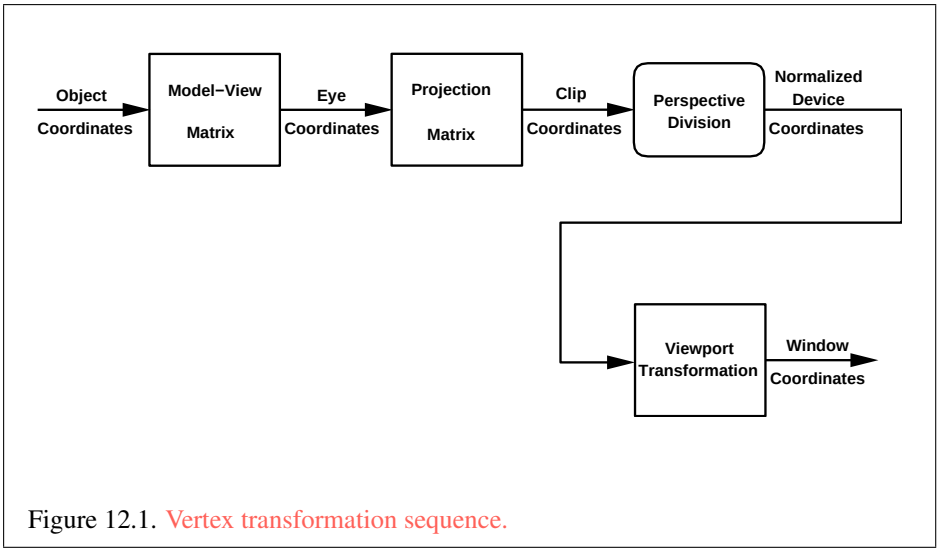


Figure 12.1. Vertex transformation sequence.

is M, then the vertex's eye coordinates are found as

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = M \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}.$$

Similarly, if P is the projection matrix, then the vertex's clip coordinates are

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = P \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}.$$

12.1.1 Matrices

The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

`void` **MatrixMode**(`enum` *mode*);

which takes one of the pre-defined constants `TEXTURE`, `MODELVIEW`, `COLOR`, or `PROJECTION` as the argument value. `TEXTURE` is described later in section 12.1.1,

and COLOR is described in section 8.4.3. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix.

The two basic commands for affecting the current matrix are

```
void LoadMatrix{fd}( const T m[16] );
void MultMatrix{fd}( const T m[16] );
```

LoadMatrix takes a pointer to a 4×4 matrix stored in column-major order as 16 consecutive floating-point values, i.e. as

$$\begin{pmatrix} a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{pmatrix}.$$

(This differs from the standard row-major C ordering for matrix elements. If the standard ordering is used, all of the subsequent transformation equations are transposed, and the columns representing vectors become rows.)

The specified matrix replaces the current matrix with the one pointed to. **MultMatrix** takes the same type argument as **LoadMatrix**, but multiplies the current matrix by the one pointed to and replaces the current matrix with the product. If C is the current matrix and M is the matrix pointed to by **MultMatrix**'s argument, then the resulting current matrix, C', is

$$C' = C \cdot M.$$

The commands

```
void LoadTransposeMatrix{fd}( const T m[16] );
void MultTransposeMatrix{fd}( const T m[16] );
```

take pointers to 4×4 matrices stored in row-major order as 16 consecutive floating-point values, i.e. as

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}.$$

The effect of

LoadTransposeMatrix[fd](m);

is the same as the effect of

LoadMatrix[fd](m^T);

The effect of

MultTransposeMatrix[fd](m);

is the same as the effect of

MultMatrix[fd](m^T);

The command

void **LoadIdentity**(void);

effectively calls **LoadMatrix** with the identity matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

There are a variety of other commands that manipulate matrices. **Rotate**, **Translate**, **Scale**, **Frustum**, and **Ortho** manipulate the current matrix. Each computes a matrix and then invokes **MultMatrix** with this matrix. In the case of

void **Rotate**{**fd**}(T θ, T x, T y, T z);

θ gives an angle of rotation in degrees; the coordinates of a vector $\mathbf{v}$ are given by $\mathbf{v} = (x\ y\ z)^T$. The computed matrix is a counter-clockwise rotation about the line through the origin with the specified axis when that axis is pointing up (i.e. the right-hand rule determines the sense of the rotation angle). The matrix is thus

$$\begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Let $\mathbf{u} = \mathbf{v}/||\mathbf{v}|| = \begin{pmatrix} x' & y' & z' \end{pmatrix}^T$. If

$$S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

then

$$R = \mathbf{u}\mathbf{u}^T + \cos\theta(I - \mathbf{u}\mathbf{u}^T) + \sin\theta S.$$

The arguments to

`void` **Translate**{**fd**}(`T` *x*, `T` *y*, `T` *z*);

give the coordinates of a translation vector as $(x\ y\ z)^T$. The resulting matrix is a translation by the specified vector:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

`void` **Scale**{**fd**}(`T` *x*, `T` *y*, `T` *z*);

produces a general scaling along the x-, y-, and z- axes. The corresponding matrix is

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

For

`void` **Frustum**(`double` *l*, `double` *r*, `double` *b*, `double` *t*,
`double` *n*, `double` *f*);

the coordinates $(l\ b\ -n)^T$ and $(r\ t\ -n)^T$ specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively (assuming that the eye is located at $(0\ 0\ 0)^T$). f gives the distance from the eye to the far clipping plane. The corresponding matrix is

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Errors

An INVALID_VALUE error is generated if $n \leq 0$, $f \leq 0$, $l = r$, $b = t$, or $n = f$.

```
void Ortho( double l, double r, double b, double t,
  double n, double f);
```

describes a matrix that produces parallel projection. $(l\ b\ -n)^T$ and $(r\ t\ -n)^T$ specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively. f gives the distance from the eye to the far clipping plane. The corresponding matrix is

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Errors

An INVALID_VALUE error is generated if $l = r$, $b = t$, or $n = f$.

For each texture coordinate set, a 4×4 matrix is applied to the corresponding texture coordinates. This matrix is applied as

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix} \begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix},$$

where the left matrix is the current texture matrix. The matrix is applied to the coordinates resulting from texture coordinate generation (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to TEXTURE causes the already described matrix operations to apply to the texture matrix.

The active texture unit selector (see section 8) specifies the texture coordinate set accessed by commands involving texture coordinate processing. Such commands include those accessing the current matrix stack (if MATRIX_MODE is TEXTURE), **TexEnv** commands controlling point sprite coordinate replacement (see section 14.4), **TexGen** (section 12.1.3), **Enable/Disable** (if any texture coordinate generation enum is selected), as well as queries of the current texture coordinates and current raster texture coordinates.

Errors

An INVALID_OPERATION error is generated by any such command if the texture coordinate set number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the value of MAX_TEXTURE_COORDS.

There is a stack of matrices for each of matrix modes MODELVIEW, PROJECTION, and COLOR, and for each texture unit. For MODELVIEW mode, the stack depth is at least 32 (that is, there is a stack of at least 32 model-view matrices). For the other modes, the depth is at least 2. Texture matrix stacks for all texture units have the same depth. The current matrix in any mode is the matrix on the top of the stack for that mode.

`void` **PushMatrix**(`void`);

pushes the stack down by one, duplicating the current matrix in both the top of the stack and the entry below it.

`void` **PopMatrix**(`void`);

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates a STACK_UNDERFLOW error; pushing a matrix onto a full stack generates a STACK_OVERFLOW error.

When the current matrix mode is TEXTURE, the texture matrix stack of the active texture unit is pushed or popped.

The state required to implement transformations consists of a four-valued integer indicating the current matrix mode, one stack of at least two 4×4 matrices for each of COLOR, PROJECTION, and each texture coordinate set, TEXTURE; and a stack of at least 32 4×4 matrices for MODELVIEW. Each matrix stack has an associated stack pointer. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW.

12.1.1.1 Matrix Queries

Matrices may be queried and returned in transposed form by calling **GetBooleanv**, **GetIntegerv**, **GetFloatv**, and **GetDoublev** with *pname* set to one of TRANSPOSE_MODELVIEW_MATRIX, TRANSPOSE_PROJECTION_MATRIX, TRANSPOSE_TEXTURE_MATRIX, or TRANSPOSE_COLOR_MATRIX. The effect of

GetFloatv(TRANSPOSE_MODELVIEW_MATRIX, m);

is the same as the effect of the command sequence

```
GetFloatv(MODELVIEW_MATRIX, m);
m = m^T;
```

Similar conversions occur when querying TRANSPOSE_PROJECTION_MATRIX, TRANSPOSE_TEXTURE_MATRIX, and TRANSPOSE_COLOR_MATRIX.

12.1.2 Normal Transformation

Finally, we consider how the model-view matrix and transformation state affect normals. Before use in lighting, normals are transformed to eye coordinates by a matrix derived from the model-view matrix. Rescaling and normalization operations are performed on the transformed normals to make them unit length prior to use in lighting. Rescaling and normalization are controlled by calling **Enable** and **Disable** with *target* RESCALE_NORMAL or NORMALIZE, respectively. This requires two bits of state. The initial state is for normals not to be rescaled or normalized.

If the model-view matrix is M, then the normal is transformed to eye coordinates by:

$$\begin{pmatrix} n_x' & n_y' & n_z' & q' \end{pmatrix} = \begin{pmatrix} n_x & n_y & n_z & q \end{pmatrix} \cdot M^{-1}$$

where, if $\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$ are the associated vertex coordinates, then

$$q = \begin{cases} 0, & w = 0, \\ \dfrac{-\begin{pmatrix} n_x & n_y & n_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}}{w}, & w \neq 0 \end{cases} \tag{12.1}$$

Implementations may choose instead to transform $\begin{pmatrix} n_x & n_y & n_z \end{pmatrix}$ to eye coordinates using

$$\begin{pmatrix} n_x' & n_y' & n_z' \end{pmatrix} = \begin{pmatrix} n_x & n_y & n_z \end{pmatrix} \cdot {M_u}^{-1}$$

where M_u is the upper leftmost 3x3 matrix taken from M.

Rescale multiplies the transformed normals by a scale factor

$$\begin{pmatrix} n_x'' & n_y'' & n_z'' \end{pmatrix} = f \begin{pmatrix} n_x' & n_y' & n_z' \end{pmatrix}$$

If rescaling is disabled, then $f = 1$. If rescaling is enabled, then f is computed as (m_{ij} denotes the matrix element in row i and column j of M^{-1}, numbering the topmost row of the matrix as row 1 and the leftmost column as column 1)

$$f = \frac{1}{\sqrt{{m_{31}}^2 + {m_{32}}^2 + {m_{33}}^2}}$$

Note that if the normals sent to GL were unit length and the model-view matrix uniformly scales space, then rescale makes the transformed normals unit length.

Alternatively, an implementation may choose f as

$$f = \frac{1}{\sqrt{{n_x'}^2 + {n_y'}^2 + {n_z'}^2}}$$

recomputing f for each normal. This makes all non-zero length normals unit length regardless of their input length and the nature of the model-view matrix.

After rescaling, the final transformed normal used in lighting, n_f, is computed as

$$n_f = m \begin{pmatrix} n_x'' & n_y'' & n_z'' \end{pmatrix}$$

If normalization is disabled, then $m = 1$. Otherwise

$$m = \frac{1}{\sqrt{{n_x''}^2 + {n_y''}^2 + {n_z''}^2}}$$

Because we specify neither the floating-point format nor the means for matrix inversion, we cannot specify behavior in the case of a poorly-conditioned (nearly singular) model-view matrix M. In case of an exactly singular matrix, the transformed normal is undefined. If the GL implementation determines that the model-view matrix is uninvertible, then the entries in the inverted matrix are arbitrary. In any case, neither normal transformation nor use of the transformed normal may lead to GL interruption or termination.

12.1.3 Generating Texture Coordinates

Texture coordinates associated with a vertex may either be taken from the current texture coordinates or generated according to a function dependent on vertex coordinates. The command

```
void TexGen{ifd}( enum coord, enum pname, T param );
void TexGen{ifd}v( enum coord, enum pname, const
   T *params );
```

controls texture coordinate generation. *coord* must be one of the constants S, T, R, or Q, indicating that the pertinent coordinate is the s, t, r, or q coordinate, respectively.

In the first form of the command, *param* specifies a single-valued texture generation parameter; in the second form, *params* is a pointer to an array of values that specify texture generation parameters.

pname must be one of TEXTURE_GEN_MODE, OBJECT_PLANE, or EYE_PLANE. If *pname* is TEXTURE_GEN_MODE, then either *params* points to or *param* is an integer that is one of OBJECT_LINEAR, EYE_LINEAR, SPHERE_MAP, REFLECTION_MAP, or NORMAL_MAP.

Data conversions are performed as specified in section 2.2.1.

If TEXTURE_GEN_MODE indicates OBJECT_LINEAR, then the generation function for the coordinate indicated by *coord* is

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o.$$

x_o, y_o, z_o, and w_o are the object coordinates of the vertex. $p_1, \ldots, p_4$ are specified by calling **TexGen** with *pname* set to OBJECT_PLANE in which case *params* points to an array containing $p_1, \ldots, p_4$. There is a distinct group of plane equation coefficients for each texture coordinate; *coord* indicates the coordinate to which the specified coefficients pertain.

If TEXTURE_GEN_MODE indicates EYE_LINEAR, then the function is

$$g = p'_1 x_e + p'_2 y_e + p'_3 z_e + p'_4 w_e$$

where

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 \end{pmatrix} M^{-1}$$

x_e, y_e, z_e, and w_e are the eye coordinates of the vertex. $p_1, \ldots, p_4$ are set by calling **TexGen** with *pname* set to EYE_PLANE in correspondence with setting the coefficients in the OBJECT_PLANE case. M is the model-view matrix in effect when $p_1, \ldots, p_4$ are specified. Computed texture coordinates may be inaccurate or undefined if M is poorly conditioned or singular.

When used with a suitably constructed texture image, calling **TexGen** with TEXTURE_GEN_MODE indicating SPHERE_MAP can simulate the reflected image of a spherical environment on a polygon. SPHERE_MAP texture coordinates are generated as follows. Denote the unit vector pointing from the origin to the vertex (in eye coordinates) by $\mathbf{u}$. Denote the current normal, after transformation to eye coordinates, by $\mathbf{n_f}$. Let $\mathbf{r} = \begin{pmatrix} r_x & r_y & r_z \end{pmatrix}^T$, the reflection vector, be given by

$$\mathbf{r} = \mathbf{u} - 2\mathbf{n_f}^T \left(\mathbf{n_f u} \right),$$

and let $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$. Then the value assigned to an s coordinate (the first **TexGen** argument value is S) is $s = \frac{r_x}{m} + \frac{1}{2}$; the value assigned to a t coordinate is $t = \frac{r_y}{m} + \frac{1}{2}$.

If TEXTURE_GEN_MODE indicates REFLECTION_MAP, compute the reflection vector **r** as described for the SPHERE_MAP mode. Then the value assigned to an s coordinate is $s = r_x$; the value assigned to a t coordinate is $t = r_y$; and the value assigned to an r coordinate is $r = r_z$.

If TEXTURE_GEN_MODE indicates NORMAL_MAP, compute the normal vector n_f as described in section 12.1.2. Then the value assigned to an s coordinate is $s = n_{f_x}$; the value assigned to a t coordinate is $t = n_{f_y}$; and the value assigned to an r coordinate is $r = n_{f_z}$ (the values n_{f_x}, n_{f_y}, and n_{f_z} are the components of n_f.)

Errors

An INVALID_ENUM error is generated if *coord* is not S, T, R, or Q.

An INVALID_ENUM error is generated if *pname* is not TEXTURE_GEN_MODE, OBJECT_PLANE, or EYE_PLANE.

An INVALID_ENUM error is generated if *coord* is either R or Q, and *pname* is SPHERE_MAP.

An INVALID_ENUM error is generated if *coord* is Q and *pname* is REFLECTION_MAP.

An INVALID_ENUM error is generated if *coord* is Q and *pname* is NORMAL_MAP.

A texture coordinate generation function is enabled or disabled by calling **Enable** and **Disable** with *target* TEXTURE_GEN_S, TEXTURE_GEN_T, TEXTURE_GEN_R, or TEXTURE_GEN_Q (each indicates the corresponding texture coordinate). When enabled, the specified texture coordinate is computed according to the current EYE_LINEAR, OBJECT_LINEAR or SPHERE_MAP specification, depending on the current setting of TEXTURE_GEN_MODE for that coordinate. When disabled, subsequent vertices will take the indicated texture coordinate from the current texture coordinates.

The state required for texture coordinate generation for each texture unit comprises a five-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR. The initial state has the texture generation function disabled for all texture coordinates. The initial values of p_i for s are all 0 except p_1 which is one; for t all the p_i are zero except p_2,

which is 1. The values of p_i for r and q are all 0. These values of p_i apply for both the EYE_LINEAR and OBJECT_LINEAR versions. Initially all texture generation modes are EYE_LINEAR.

12.1.3.1 Texture Coordinate Generation Queries

The command

```
void GetTexGen{ifd}v( enum coord, enum pname,
    T *params );
```

returns information about *pname* for *coord* in *params*. *coord* must be one of S, T, R, or Q. EYE_LINEAR coefficients are returned in the eye coordinates that were computed when the plane was specified; OBJECT_LINEAR coefficients are returned in object coordinates.

The same errors generated by **TexGen*** for invalid *light* and *pname* parameters are generated by **GetTexGen***.

12.2 Fixed-Function Vertex Lighting and Coloring

Figures 12.2 and 12.3 diagram the processing of RGBA colors and color indices before rasterization. Incoming colors arrive in one of several formats. R, G, B, and A components specified with unsigned and signed integer versions of the **Color** command are converted to floating-point as described in equations 2.1 and 2.2, respectively. As a result of limited precision, some converted values will not be represented exactly. In color index mode, a single-valued color index is not mapped.

Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or current color (primary color) and current secondary color are used in further processing. After lighting, RGBA colors may be clamped to the range $[0, 1]$ as described in section 13.1. A color index is converted to fixed-point and then its integer portion is masked (see section 13.1). After clamping or masking, a primitive may be *flatshaded*, indicating that all vertices of the primitive are to have the same colors. Finally, if a primitive is clipped, then colors (and texture coordinates) must be computed at the vertices introduced or modified by clipping.

12.2.1 Lighting

GL lighting computes colors for each vertex sent to the GL. This is accomplished by applying an equation defined by a client-specified lighting model to a collection

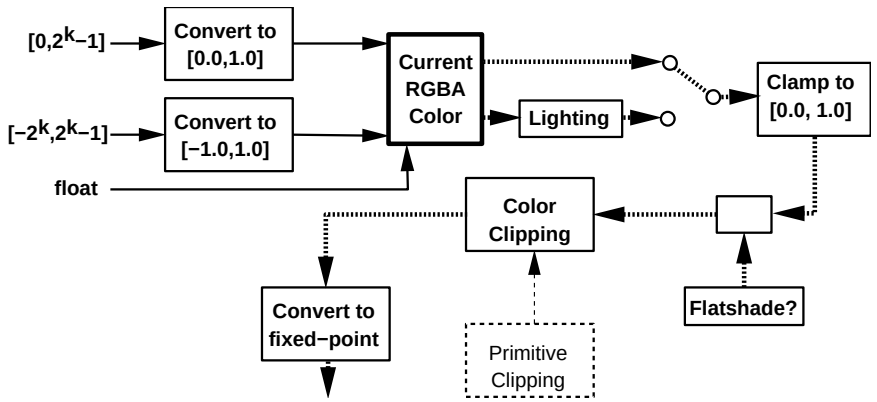


Figure 12.2. Processing of RGBA colors. The heavy dotted lines indicate both primary and secondary vertex colors, which are processed in the same fashion. k is the minimum required bit width of the integer type representing a color component.

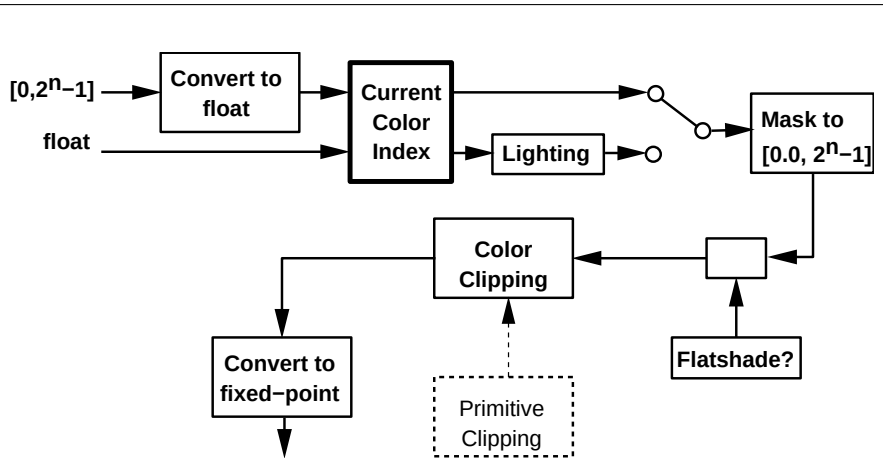


Figure 12.3. Processing of color indices. n is the number of bits in a color index.

of parameters that can include the vertex coordinates, the coordinates of one or more light sources, the current normal, and parameters defining the characteristics of the light sources and a current material. The following discussion assumes that the GL is in RGBA mode. (Color index lighting is described in section 12.2.6.)

Lighting is turned on or off by calling **Enable** or **Disable** with *target* `LIGHTING`. If lighting is off, the current color and current secondary color are assigned to the vertex primary and secondary color, respectively. If lighting is on, colors computed from the current lighting parameters are assigned to the vertex primary and secondary colors.

12.2.1.1 Lighting Operation

A lighting parameter is of one of five types: color, position, direction, real, or boolean. A color parameter consists of four floating-point values, one for each of R, G, B, and A, in that order. There are no restrictions on the allowable values for these parameters. A position parameter consists of four floating-point coordinates (x, y, z, and w) that specify a position in object coordinates (w may be zero, indicating a point at infinity in the direction given by x, y, and z). A direction parameter consists of three floating-point coordinates (x, y, and z) that specify a direction in object coordinates. A real parameter is one floating-point value. The various values and their types are summarized in table 12.1. The result of a lighting computation is undefined if a value for a parameter is specified that is outside the range given for that parameter in the table.

There are n light sources, indexed by $i = 0, \ldots, n-1$. (n is an implementation-dependent maximum that must be at least 8.) Note that the default values for $\mathbf{d}_{cli}$ and $\mathbf{s}_{cli}$ differ for $i = 0$ and $i > 0$.

Before specifying the way that lighting computes colors, we introduce operators and notation that simplify the expressions involved. If $\mathbf{c}_1$ and $\mathbf{c}_2$ are colors without alpha where $\mathbf{c}_1 = (r_1, g_1, b_1)$ and $\mathbf{c}_2 = (r_2, g_2, b_2)$, then define $\mathbf{c}_1 * \mathbf{c}_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$. Addition of colors is accomplished by addition of the components. Multiplication of colors by a scalar means multiplying each component by that scalar. If $\mathbf{d}_1$ and $\mathbf{d}_2$ are directions, then define

$$\mathbf{d}_1 \odot \mathbf{d}_2 = \max\{\mathbf{d}_1 \cdot \mathbf{d}_2, 0\}.$$

(Directions are taken to have three coordinates.) If $\mathbf{P}_1$ and $\mathbf{P}_2$ are (homogeneous, with four coordinates) points then let $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ be the unit vector that points from $\mathbf{P}_1$ to $\mathbf{P}_2$. Note that if $\mathbf{P}_2$ has a zero w coordinate and $\mathbf{P}_1$ has non-zero w coordinate, then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector corresponding to the direction specified by the x, y, and z coordinates of $\mathbf{P}_2$; if $\mathbf{P}_1$ has a zero w coordinate and $\mathbf{P}_2$ has a non-zero w coordinate then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector that is the negative of that corresponding

Parameter	Type	Default Value	Description
Material Parameters			
$\mathbf{a}_{cm}$	color	$(0.2, 0.2, 0.2, 1.0)$	ambient color of material
$\mathbf{d}_{cm}$	color	$(0.8, 0.8, 0.8, 1.0)$	diffuse color of material
$\mathbf{s}_{cm}$	color	$(0.0, 0.0, 0.0, 1.0)$	specular color of material
$\mathbf{e}_{cm}$	color	$(0.0, 0.0, 0.0, 1.0)$	emissive color of material
s_{rm}	real	0.0	specular exponent (range: $[0.0, 128.0]$)
a_m	real	0.0	ambient color index
d_m	real	1.0	diffuse color index
s_m	real	1.0	specular color index
Light Source Parameters			
$\mathbf{a}_{cli}$	color	$(0.0, 0.0, 0.0, 1.0)$	ambient intensity of light i
$\mathbf{d}_{cli}(i = 0)$	color	$(1.0, 1.0, 1.0, 1.0)$	diffuse intensity of light 0
$\mathbf{d}_{cli}(i > 0)$	color	$(0.0, 0.0, 0.0, 1.0)$	diffuse intensity of light i
$\mathbf{s}_{cli}(i = 0)$	color	$(1.0, 1.0, 1.0, 1.0)$	specular intensity of light 0
$\mathbf{s}_{cli}(i > 0)$	color	$(0.0, 0.0, 0.0, 1.0)$	specular intensity of light i
$\mathbf{P}_{pli}$	position	$(0.0, 0.0, 1.0, 0.0)$	position of light i
$\mathbf{s}_{dli}$	direction	$(0.0, 0.0, -1.0)$	direction of spotlight for light i
s_{rli}	real	0.0	spotlight exponent for light i (range: $[0.0, 128.0]$)
c_{rli}	real	180.0	spotlight cutoff angle for light i (range: $[0.0, 90.0]$, 180.0)
k_{0i}	real	1.0	constant attenuation factor for light i (range: $[0.0, \infty)$)
k_{1i}	real	0.0	linear attenuation factor for light i (range: $[0.0, \infty)$)
k_{2i}	real	0.0	quadratic attenuation factor for light i (range: $[0.0, \infty)$)
Lighting Model Parameters			
$\mathbf{a}_{cs}$	color	$(0.2, 0.2, 0.2, 1.0)$	ambient color of scene
v_{bs}	boolean	`FALSE`	viewer assumed to be at $(0, 0, 0)$ in eye coordinates (`TRUE`) or $(0, 0, \infty)$ (`FALSE`)
c_{es}	enum	`SINGLE_COLOR`	controls computation of colors
t_{bs}	boolean	`FALSE`	use two-sided lighting mode

Table 12.1: Summary of lighting parameters. The range of individual color components is $(-\infty, +\infty)$.

to the direction specified by $\mathbf{P}_1$. If both $\mathbf{P}_1$ and $\mathbf{P}_2$ have zero w coordinates, then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector obtained by normalizing the direction corresponding to $\mathbf{P}_2 - \mathbf{P}_1$.

If $\mathbf{d}$ is an arbitrary direction, then let $\hat{\mathbf{d}}$ be the unit vector in $\mathbf{d}$'s direction. Let $\|\mathbf{P}_1\mathbf{P}_2\|$ be the distance between $\mathbf{P}_1$ and $\mathbf{P}_2$. Finally, let $\mathbf{V}$ be the point corresponding to the vertex being lit, and $\mathbf{n}$ be the corresponding normal. Let $\mathbf{P}_e$ be the eyepoint ($(0, 0, 0, 1)$ in eye coordinates).

Lighting produces two colors at a vertex: a primary color $\mathbf{c}_{pri}$ and a secondary color $\mathbf{c}_{sec}$. The values of $\mathbf{c}_{pri}$ and $\mathbf{c}_{sec}$ depend on the light model color control, c_{es}. If $c_{es} =$ `SINGLE_COLOR`, then the equations to compute $\mathbf{c}_{pri}$ and $\mathbf{c}_{sec}$ are

$$\begin{aligned}
\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\
&+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\
&+ \sum_{i=0}^{n-1} (att_i)(spot_i) \left[\mathbf{a}_{cm} * \mathbf{a}_{cli}\right. \\
&\qquad + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} \\
&\qquad \left. + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}\right] \\
\mathbf{c}_{sec} &= (0, 0, 0, 1)
\end{aligned}$$

If $c_{es} =$ `SEPARATE_SPECULAR_COLOR`, then

$$\begin{aligned}
\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\
&+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\
&+ \sum_{i=0}^{n-1} (att_i)(spot_i) \left[\mathbf{a}_{cm} * \mathbf{a}_{cli}\right. \\
&\qquad \left. + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli}\right] \\
\mathbf{c}_{sec} &= \sum_{i=0}^{n-1} (att_i)(spot_i)(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}
\end{aligned}$$

where

$$f_i = \begin{cases} 1, & \mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli} \neq 0, \\ 0, & \text{otherwise}, \end{cases} \tag{12.2}$$

$$\mathbf{h}_i = \begin{cases} \overrightarrow{\mathbf{VP}}_{pli} + \overrightarrow{\mathbf{VP}}_e, & v_{bs} = \texttt{TRUE}, \\ \overrightarrow{\mathbf{VP}}_{pli} + \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T, & v_{bs} = \texttt{FALSE}, \end{cases} \tag{12.3}$$

$$att_i = \begin{cases} \dfrac{1}{k_{0i} + k_{1i}\|\mathbf{VP}_{pli}\| + k_{2i}\|\mathbf{VP}_{pli}\|^2}, & \text{if } \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise.} \end{cases} \tag{12.4}$$

$$spot_i = \begin{cases} (\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), \\ 1.0, & c_{rli} = 180.0. \end{cases} \tag{12.5}$$

All computations are carried out in eye coordinates.

The value of A produced by lighting is the alpha value associated with $\mathbf{d}_{cm}$. A is always associated with the primary color $\mathbf{c}_{pri}$; the alpha component of $\mathbf{c}_{sec}$ is always 1.

Results of lighting are undefined if the w_e coordinate (w in eye coordinates) of $\mathbf{V}$ is zero.

Lighting may operate in *two-sided* mode ($t_{bs} = \texttt{TRUE}$), in which a *front* color is computed with one set of material parameters (the *front material*) and a *back* color is computed with a second set of material parameters (the *back material*). This second computation replaces $\mathbf{n}$ with $-\mathbf{n}$. If $t_{bs} = \texttt{FALSE}$, then the back color and front color are both assigned the color computed using the front material with $\mathbf{n}$.

Additionally, vertex and geometry shaders can operate in two-sided color mode. When a vertex or geometry shader is active, front and back colors can be computed by the shader and written to the `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor` and `gl_BackSecondaryColor` outputs. If `VERTEX_PROGRAM_TWO_SIDE` is enabled, the GL chooses between front and back colors, as described below. Otherwise, the front color output is always

selected. Two-sided color mode is enabled and disabled by calling **Enable** or **Disable** with *target* VERTEX_PROGRAM_TWO_SIDE.

The selection between back and front colors depends on the primitive of which the vertex being lit is a part. If the primitive is a point or a line segment, the front color is always selected. If it is a polygon, then the selection is performed based on the sign of the (clipped or unclipped) polygon's area a computed in window coordinates, as described in equation 14.8 of section 14.6.1. If the sign of a (including the possible reversal of this sign as indicated by the last call to **FrontFace**) is positive, the color of each vertex of the polygon becomes the front color computed for that vertex; otherwise the back color is selected.

12.2.2 Lighting Parameter Specification

Lighting parameters are divided into three categories: material parameters, light source parameters, and lighting model parameters (see table 12.1). Sets of lighting parameters are specified with

```
void Material{if}( enum face, enum pname, T param );
void Material{if}v( enum face, enum pname, const
   T *params );
void Light{if}( enum light, enum pname, T param );
void Light{if}v( enum light, enum pname, const
   T *params );
void LightModel{if}( enum pname, T param );
void LightModel{if}v( enum pname, const T *params );
```

pname is a parameter name from table 12.2 indicating which parameter is to be set. In the vector versions of the commands, *params* is a pointer to a group of values to which to set the indicated parameter. The number of values pointed to depends on the parameter being set. In the non-vector versions, *param* is a value to which to set a single-valued parameter.

For **Material**, *face* must be one of FRONT, BACK, or FRONT_AND_BACK, indicating that the property *name* of the front or back material, or both, respectively, should be set. In the case of **Light**, *light* is a symbolic constant of the form LIGHTi, indicating that light i is to have the specified parameter set.

Table 12.2 gives, for each of the three parameter groups, the correspondence between the pre-defined constant names and their names in the lighting equations, along with the number of values that must be specified with each. Color parameters specified with **Material** and **Light** are converted to floating-point values (if specified as integers) as described in equation 2.2.

Parameter	Name	Number of values
Material Parameters (**Material**)		
$\mathbf{a}_{cm}$	AMBIENT	4
$\mathbf{d}_{cm}$	DIFFUSE	4
$\mathbf{a}_{cm}, \mathbf{d}_{cm}$	AMBIENT_AND_DIFFUSE	4
$\mathbf{s}_{cm}$	SPECULAR	4
$\mathbf{e}_{cm}$	EMISSION	4
s_{rm}	SHININESS	1
a_m, d_m, s_m	COLOR_INDEXES	3
Light Source Parameters (**Light**)		
$\mathbf{a}_{cli}$	AMBIENT	4
$\mathbf{d}_{cli}$	DIFFUSE	4
$\mathbf{s}_{cli}$	SPECULAR	4
$\mathbf{P}_{pli}$	POSITION	4
$\mathbf{s}_{dli}$	SPOT_DIRECTION	3
s_{rli}	SPOT_EXPONENT	1
c_{rli}	SPOT_CUTOFF	1
k_0	CONSTANT_ATTENUATION	1
k_1	LINEAR_ATTENUATION	1
k_2	QUADRATIC_ATTENUATION	1
Lighting Model Parameters (**LightModel**)		
$\mathbf{a}_{cs}$	LIGHT_MODEL_AMBIENT	4
v_{bs}	LIGHT_MODEL_LOCAL_VIEWER	1
t_{bs}	LIGHT_MODEL_TWO_SIDE	1
c_{es}	LIGHT_MODEL_COLOR_CONTROL	1

Table 12.2: Correspondence of lighting parameter symbols to names. AMBIENT_-AND_DIFFUSE is used to set $\mathbf{a}_{cm}$ and $\mathbf{d}_{cm}$ to the same value.

Material properties can be changed inside a **Begin / End** pair by calling **Material**. However, when a vertex shader is active such property changes are not guaranteed to update material parameters, defined in table 12.2, until the following **End** command.

The current model-view matrix is applied to the position parameter indicated with **Light** for a particular light source when that position is specified. These transformed values are the values used in the lighting equation.

The spotlight direction is transformed when it is specified using only the upper leftmost 3x3 portion of the model-view matrix. That is, if $\mathbf{M}_u$ is the upper left 3x3 matrix taken from the current model-view matrix M, then the spotlight direction

$$\begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$$

is transformed to

$$\begin{pmatrix} d'_x \\ d'_y \\ d'_z \end{pmatrix} = M_u \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}.$$

Errors

An `INVALID_ENUM` error is generated by **Material*** if *face* is not `FRONT`, `BACK`, or `FRONT_AND_BACK`.

An `INVALID_ENUM` error is generated by **Light*** if *light* is not one of `LIGHT`i, where i is in the range zero to the value of `MAX_LIGHTS` minus one and `LIGHT`i = `LIGHT0` + i.

An `INVALID_ENUM` error is generated if *pname* is not an accepted parameter name for the corresponding commands from table 12.2.

An `INVALID_VALUE` error is generated if a specified lighting parameter value lies outside the allowable range for that parameter (see table 12.1; the symbol "∞" indicates the maximum representable magnitude for the indicated type.)

An `INVALID_ENUM` error is generated by the scalar commands **Material{if}**, **Light{if}**, and **LightModel{if}** if *param* is a non-scalar parameter (e.g. if the number of values for that parameter, as shown in table 12.2, is greater than 1).

An individual light is enabled or disabled by calling **Enable** or **Disable** with *target* `LIGHT`i, where i is in the range zero to the value of `MAX_LIGHTS` minus one. If light i is disabled, the ith term in the lighting equation is effectively removed from the summation.

12.2.3 ColorMaterial

It is possible to attach one or more material properties to the current color, so that they continuously track its component values. This behavior is enabled and disabled by calling **Enable** or **Disable** with *target* `COLOR_MATERIAL`.

The command that controls which of these modes is selected is

`void` **ColorMaterial**(`enum` *face*, `enum` *mode*);

face is `FRONT`, `BACK`, or `FRONT_AND_BACK`, indicating whether the front material, back material, or both are affected by the current color.

mode is one of `EMISSION`, `AMBIENT`, `DIFFUSE`, `SPECULAR`, or `AMBIENT_AND_DIFFUSE` and specifies which material property or properties track the current color. If *mode* is `EMISSION`, `AMBIENT`, `DIFFUSE`, or `SPECULAR`, then the value of $\mathbf{e}_{cm}$, $\mathbf{a}_{cm}$, $\mathbf{d}_{cm}$ or $\mathbf{s}_{cm}$, respectively, will track the current color. If *mode* is `AMBIENT_AND_DIFFUSE`, both $\mathbf{a}_{cm}$ and $\mathbf{d}_{cm}$ track the current color.

The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when **ColorMaterial** is not currently enabled to override that particular value. When `COLOR_MATERIAL` is enabled, the indicated parameter or parameters always track the current color. For instance, calling

ColorMaterial (`FRONT`, `AMBIENT`)

while `COLOR_MATERIAL` is enabled sets the front material $\mathbf{a}_{cm}$ to the value of the current color.

Material properties can be changed inside a **Begin** / **End** pair indirectly by enabling **ColorMaterial** mode and making **Color** calls. However, when a vertex shader is active such property changes are not guaranteed to update material parameters, defined in table 12.2, until the following **End** command.

12.2.4 Lighting Parameter Queries

The command

`void` **GetLight**{**if**}**v**(`enum` *light*, `enum` *pname*, `T` **params*);

returns information about light parameter *pname* for *light* in *params*. `POSITION` and `SPOT_DIRECTION` return values in eye coordinates; these are the coordinates that were computed when the position or direction was specified.

The same errors generated by **Light*** for invalid *light* and *pname* parameters are generated by **GetLight***.

The command

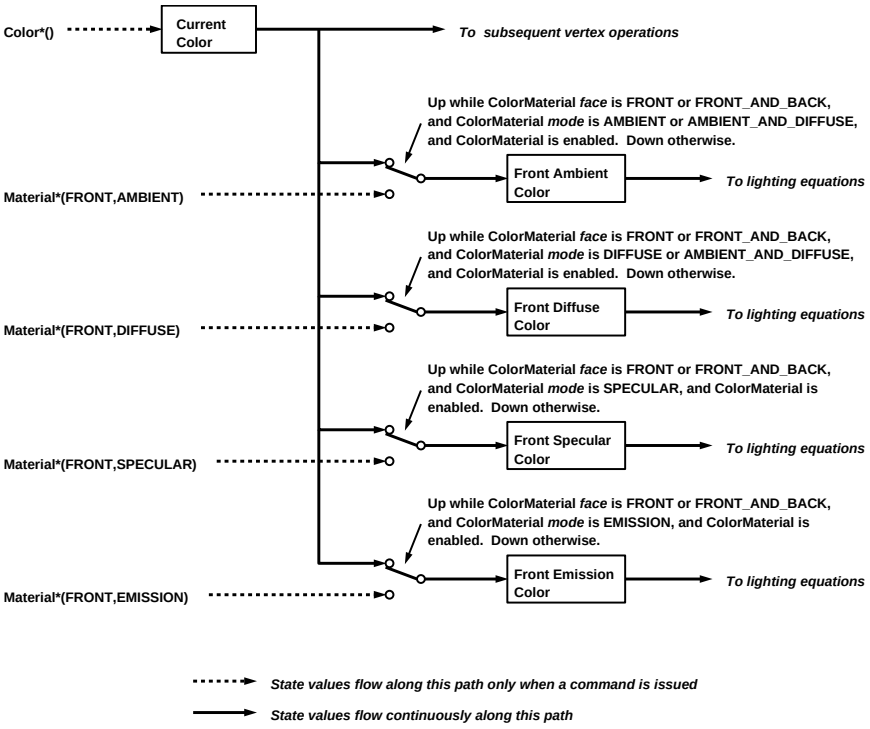


Figure 12.4. **ColorMaterial** operation. Material properties are continuously updated from the current color while **ColorMaterial** is enabled and has the appropriate mode. Only the front material properties are included in this figure. The back material properties are treated identically, except that *face* must be `BACK` or `FRONT_AND_BACK`.

`void` **GetMaterial**{**if**}**v**(`enum` *face*, `enum` *pname*, `T` **params*);

returns information about material property *pname* for *face* in *params*. *face* must be either `FRONT` or `BACK`, indicating the front or back material, respectively.

The same errors generated by **Material*** for invalid *light* and *pname* parameters are generated by **GetMaterial***.

12.2.5 Lighting State

The state required for lighting consists of all of the lighting parameters (front and back material parameters, lighting model parameters, and at least 8 sets of light parameters), a bit indicating whether a back color distinct from the front color should be computed, at least 8 bits to indicate which lights are enabled, a five-valued variable indicating the current **ColorMaterial** mode, a bit indicating whether or not `COLOR_MATERIAL` is enabled, and a single bit to indicate whether lighting is enabled or disabled. In the initial state, all lighting parameters have their default values. Back color evaluation does not take place, **ColorMaterial** is `FRONT_AND_BACK` and `AMBIENT_AND_DIFFUSE`, and both lighting and `COLOR_MATERIAL` are disabled.

12.2.6 Color Index Lighting

A simplified lighting computation applies in color index mode that uses many of the parameters controlling RGBA lighting, but none of the RGBA material parameters. First, the RGBA diffuse and specular intensities of light i ($\mathbf{d}_{cli}$ and $\mathbf{s}_{cli}$, respectively) determine color index diffuse and specular light intensities, d_{li} and s_{li} from

$$d_{li} = (.30)R(\mathbf{d}_{cli}) + (.59)G(\mathbf{d}_{cli}) + (.11)B(\mathbf{d}_{cli})$$

and

$$s_{li} = (.30)R(\mathbf{s}_{cli}) + (.59)G(\mathbf{s}_{cli}) + (.11)B(\mathbf{s}_{cli}).$$

$R(\mathbf{x})$ indicates the R component of the color $\mathbf{x}$ and similarly for $G(\mathbf{x})$ and $B(\mathbf{x})$.

Next, let

$$s = \sum_{i=0}^{n} (att_i)(spot_i)(s_{li})(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}$$

where att_i and $spot_i$ are given by equations 12.4 and 12.5, respectively, and f_i and $\hat{\mathbf{h}}_i$ are given by equations 12.2 and 12.3, respectively. Let $s' = \min(s, 1)$. Finally, let

$$d = \sum_{i=0}^{n} (att_i)(spot_i)(d_{li})(\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli}).$$

Then color index lighting produces a value c, given by

$$c = a_m + d(1 - s')(d_m - a_m) + s'(s_m - a_m).$$

The final color index is

$$c' = \min(c, s_m).$$

The values a_m, d_m and s_m are material properties described in tables 12.1 and 12.2. Any ambient light intensities are incorporated into a_m. As with RGBA lighting, disabled lights cause the corresponding terms from the summations to be omitted. The interpretation of t_{bs} and the calculation of front and back colors is carried out as has already been described for RGBA lighting.

The values a_m, d_m, and s_m are set with **Material** using a *pname* of `COLOR_INDEXES`. Their initial values are 0, 1, and 1, respectively. The additional state consists of three floating-point values. These values have no effect on RGBA lighting.

Chapter 13

Fixed-Function Vertex Post-Processing

After programmable or fixed-function vertex processing, the following fixed-function operations are applied to vertices of the resulting primitives:

- Color clamping or masking (see section 13.1).
- Transform feedback (see section 13.3).
- Primitive queries (see section 13.4).
- Flatshading (see section 13.6).
- Primitive clipping, including client-defined half-spaces (see section 13.7).
- Shader output or associated data clipping (see section 13.7.1).
- Perspective division on clip coordinates (see section 13.8).
- Viewport mapping, including depth range scaling (see section 13.8.1).
- Front face determination for polygon primitives (see section 14.6.1).
- Color, texture coordinate, fog, point-size and generic attribute clipping (see section 13.7.1).
- Final color processing (see section 13.9).

Next, rasterization is performed on primitives as described in chapter 14.

13.1 Clamping or Masking

When the GL is in RGBA mode and vertex color clamping is enabled, all components of both primary and secondary colors are clamped to the range $[0, 1]$ after lighting. If color clamping is disabled, the primary and secondary colors are unmodified. Vertex color clamping is controlled by calling **ClampColor**, as described in section 18.1.1, with a *target* of `CLAMP_VERTEX_COLOR`.

For a color index, the index is first converted to fixed-point with an unspecified number of bits to the right of the binary point; the nearest fixed-point value is selected. Then, the bits to the right of the binary point are left alone while the integer portion is masked (bitwise ANDed) with $2^n - 1$, where n is the number of bits in a color in the color index buffer (buffers are discussed in chapter 9).

The state required for vertex color clamping is a three-valued integer, initially set to `TRUE`.

13.2 The Last Vertex Processing Stage

In the remainder of this chapter, the *last vertex processing stage* is the shader stage which passes outputs on to the operations in this chapter.

The last vertex processing stage is:

- The geometry shader stage, if a geometry shader is active;
- Otherwise, the tessellation evaluation stage, if a tessellation evaluation shader is active;
- Otherwise, the vertex shader stage, if a vertex shader is active.

If none of these stages are active, there is no last vertex processing stage, and the input to fixed-function vertex postprocessing is the result of fixed-function vertex processing.

13.3 Transform Feedback

In transform feedback mode, attributes of the vertices of transformed primitives passed to the transform feedback stage are written out to one or more buffer objects. The vertices are fed back after vertex color clamping, but before flatshading and clipping. The transformed vertices may be optionally discarded after being stored into one or more buffer objects, or they can be passed on down to the clipping stage

for further processing. The set of attributes captured is determined when a program is linked.

Transform feedback captures the vertices of each primitive emitted by the last vertex processing stage.

If separable program objects are in use, the set of attributes captured is taken from the program object active on the last vertex processing stage. The set of attributes to capture in transform feedback mode for any other program active on a previous shader stage is ignored.

13.3.1 Transform Feedback Objects

The set of buffer objects used to capture vertex output variables and related state are stored in a transform feedback object. The set of attributes captured in transform feedback mode is determined using the state of the active program object. The name space for transform feedback objects is the unsigned integers. The name zero designates the default transform feedback object.

The command

```
void GenTransformFeedbacks( sizei n, uint *ids );
```

returns *n* previously unused transform feedback object names in *ids*. These names are marked as used, for the purposes of **GenTransformFeedbacks** only, but they acquire transform feedback state only when they are first bound.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

Transform feedback objects are deleted by calling

```
void DeleteTransformFeedbacks( sizei n, const
   uint *ids );
```

ids contains *n* names of transform feedback objects to be deleted. After a transform feedback object is deleted it has no contents, and its name is again unused. Unused names in *ids* that have been marked as used for the purposes of **GenTransformFeedbacks** are marked as unused again. Unused names in *ids* are silently ignored, as is the value zero. The default transform feedback object cannot be deleted.

Errors

An INVALID_VALUE error is generated if *n* is negative.

An INVALID_OPERATION error is generated if the transform feedback operation for any object named by *ids* is currently active.

The command

```
boolean IsTransformFeedback( uint id );
```

returns TRUE if *id* is the name of a transform feedback object. If *id* is zero, or a non-zero value that is not the name of a transform feedback object, **IsTransformFeedback** returns FALSE. No error is generated if *id* is not a valid transform feedback object name.

A transform feedback object is created by binding a name returned by **GenTransformFeedbacks** with the command

```
void BindTransformFeedback( enum target, uint id );
```

target must be TRANSFORM_FEEDBACK and *id* is the transform feedback object name. The resulting transform feedback object is a new state vector, comprising all the state and with the same initial values listed in table 23.59. Additionally, the new object is bound to the GL state vector and is used for subsequent transform feedback operations[1].

BindTransformFeedback can also be used to bind an existing transform feedback object to the GL state for subsequent use. If the bind is successful, no change is made to the state of the newly bound transform feedback object and any previous binding to *target* is broken.

While a transform feedback buffer object is bound, GL operations on the target to which it is bound affect the bound transform feedback object, and queries of the target to which a transform feedback object is bound return state from the bound object. When buffer objects are bound for transform feedback, they are attached to the currently bound transform feedback object. Buffer objects are used for transform feedback only if they are attached to the currently bound transform feedback object.

In the initial state, a default transform feedback object is bound and treated as a transform feedback object with a name of zero. That object is bound any time **BindTransformFeedback** is called with *id* of zero.

[1]The TRANSFORM_FEEDBACK_BUFFER_BINDING generic buffer binding point has now been moved to context state, rather than per-object state, and should be unaffected by this command. It is possible that some drivers do not yet implement this change.

Errors

An `INVALID_ENUM` error is generated if *target* is not `TRANSFORM_FEEDBACK`.

An `INVALID_OPERATION` error is generated if the transform feedback operation is active on the currently bound transform feedback object, and that operation is not paused (as described below).

An `INVALID_OPERATION` error is generated if *id* is not zero or a name returned from a previous call to **GenTransformFeedbacks**, or if such a name has since been deleted with **DeleteTransformFeedbacks**.

New transform feedback objects may also be created with the command

`void` **CreateTransformFeedbacks**(`sizei` *n*, `uint` **ids*);

CreateTransformFeedbacks returns *n* previously unused transform feedback object names in *ids*, each representing a new state vector, comprising the state and with all the same initial values listed in table 23.59.

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

13.3.2 Transform Feedback Primitive Capture

Transform feedback for the currently bound transform feedback object is started (made *active*) and finished (made *inactive*) with the commands

`void` **BeginTransformFeedback**(`enum` *primitiveMode*);

and

`void` **EndTransformFeedback**(`void`);

respectively. *primitiveMode* must be `TRIANGLES`, `LINES`, or `POINTS`, and specifies the output type of primitives that will be recorded into the buffer objects bound for transform feedback (see below). *primitiveMode* restricts the primitive types that may be rendered while transform feedback is active, as shown in table 13.1.

EndTransformFeedback first performs an implicit **ResumeTransformFeedback** (see below) if transform feedback is paused.

BeginTransformFeedback and **EndTransformFeedback** calls must be paired. Transform feedback is initially inactive.

Transform feedback mode captures the values of output variables written by the last vertex processing stage. It may not be used during fixed-function vertex processing.

Errors

An `INVALID_OPERATION` error is generated by **BeginTransformFeedback** if no vertex or geometry shader is active.

An `INVALID_ENUM` error is generated by **BeginTransformFeedback** if *primitiveMode* is not `TRIANGLES`, `LINES`, or `POINTS`.

An `INVALID_OPERATION` error is generated by **BeginTransformFeedback** if transform feedback is active for the current transform feedback object.

An `INVALID_OPERATION` error is generated by **EndTransformFeedback** if transform feedback is inactive.

Transform feedback operations for the currently bound transform feedback object may be paused and resumed by calling

```
void PauseTransformFeedback( void );
```

and

```
void ResumeTransformFeedback( void );
```

respectively. When transform feedback operations are paused, transform feedback is still considered active and changing most transform feedback state related to the object results in an error. However, a new transform feedback object may be bound while transform feedback is paused.

When transform feedback is active and not paused, all geometric primitives generated must be compatible with the value of *primitiveMode* passed to **BeginTransformFeedback**.

Errors

An `INVALID_OPERATION` error is generated by any command that transfers vertices to the GL if *mode* is not one of the allowed modes in table 13.1. If a tessellation evaluation or geometry shader is active, the type of primitive emitted by that shader is used instead of the *mode* parameter passed to drawing commands for the purposes of this error check. If tessellation evaluation and geometry shaders are both active, the output primitive type of the geometry

Transform Feedback *primitiveMode*	Allowed render primitive (Begin) *mode*s
POINTS	POINTS
LINES	LINES, LINE_LOOP, LINE_STRIP
TRIANGLES	TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN
	QUADS, QUAD_STRIP, POLYGON

Table 13.1: Legal combinations of the transform feedback primitive mode, as passed to **BeginTransformFeedback**, and the current primitive mode.

shader will be used for the purposes of this error. Any primitive type may be used while transform feedback is paused.

Errors

An INVALID_OPERATION error is generated by **PauseTransformFeedback** if the currently bound transform feedback object is not active or is paused.

An INVALID_OPERATION error is generated by **ResumeTransformFeedback** if the currently bound transform feedback object is not active or is not paused.

Regions of buffer objects are bound as targets of the currently bound transform feedback object by calling one of the **BindBuffer*** commands (see sections 6.1 and 6.1.1 with *target* set to TRANSFORM_FEEDBACK_BUFFER. Alternatively, regions of buffer objects may be bound directly to a transform feedback object with the commands

```
void TransformFeedbackBufferRange( uint xfb, uint index,
    uint buffer, intptr offset, sizeiptr size );
void TransformFeedbackBufferBase( uint xfb, uint index,
    uint buffer );
```

xfb must be zero, indicating the default transform feedback object, or the name of an existing transform feedback object. *buffer* must be zero or the name of an existing buffer object.

TransformFeedbackBufferRange and **TransformFeedbackBufferBase** behave similarly to **BindBufferRange** and **BindBufferBase**, respectively, except

that the target of the operation is *xfb*, and they do not affect any binding to the generic TRANSFORM_FEEDBACK_BUFFER target.

Errors

An INVALID_OPERATION error is generated if *xfb* is not zero or the name of an existing transform feedback object.

An INVALID_VALUE error is generated if *buffer* is not zero or the name of an existing buffer object.

An INVALID_VALUE error is generated if *index* is greater than or equal to the number of binding points for transform feedback, as described in section 6.7.1.

An INVALID_VALUE error is generated by **TransformFeedbackBufferRange** if *offset* is negative.

An INVALID_VALUE error is generated by **TransformFeedbackBufferRange** if *size* is less than or equal to zero.

An INVALID_VALUE error is generated by **TransformFeedbackBufferRange** if *offset* or *size* do not satisfy the constraints described for those parameters for transform feedback array bindings, as described in section 6.7.1.

When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active and not paused, the values of the specified output variables of the vertex are appended to the buffer objects bound to the transform feedback binding points. The attributes of the first vertex received after **BeginTransformFeedback** are written at the starting offsets of the bound buffer objects set by **BindBufferRange**, and subsequent vertex attributes are appended to the buffer object. When capturing line and triangle primitives, all attributes of the first vertex are written first, followed by attributes of the subsequent vertices.

When capturing vertices, the stride associated with each transform feedback binding point indicates the number of basic machine units of storage reserved for each vertex in the bound buffer object. For every vertex captured, each output variable with an assigned transform feedback offset will be written to the storage reserved for the vertex at the associated binding point. When writing output variables that are arrays or structures, individual array elements or structure members are written in order. For vector types, individual components are written in order. For matrix types, outputs are written as an array of column vectors. If any component of an output with an assigned transform feedback offset was not written to by its shader, the value recorded for that component is undefined. The results of writing an output variable to a transform feedback buffer are undefined if any component of that variable would be written at an offset not aligned to the size of

the component. When capturing a vertex, any portion of the reserved storage not associated with an output variable with an assigned transform feedback offset will be unmodified.

When transform feedback is paused, no vertices are recorded. When transform feedback is resumed, subsequent vertices are appended to the bound buffer objects immediately following the last vertex written before transform feedback was paused.

When quads and polygons are provided to transform feedback with a primitive mode of `TRIANGLES`, they will be tessellated and recorded as triangles (the order of tessellation within a primitive is undefined). Individual lines or triangles of a strip or fan primitive will be extracted and recorded separately. Incomplete primitives are not recorded.

When using a geometry shader that writes vertices to multiple vertex streams, each vertex emitted may trigger a new primitive in the vertex stream to which it was emitted. If transform feedback is active, the outputs of the primitive are written to a transform feedback binding point if and only if the outputs directed at that binding point belong to the vertex stream in question. All outputs assigned to a given binding point are required to come from a single vertex stream.

If recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position $offset + size - 1$, as set by **BindBufferRange**, then no vertices of that primitive are recorded in any buffer object, and the counter corresponding to the asynchronous query target `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` (see section 13.4) is not incremented. For the purposes of this test, `gl_SkipComponents` variables are counted as recording data to a buffer object.

Any transform feedback binding point used for capturing vertices must have buffer objects bound when **BeginTransformFeedback** is called. A binding point requires a bound buffer object if and only if its associated stride in the program object used for transform feedback primitive capture is non-zero and the transform feedback buffer is associated with output variables, blocks or block members in the program object.

Errors

An `INVALID_OPERATION` error is generated by **BeginTransformFeedback** if any of these binding points does not have a buffer object bound.

An `INVALID_OPERATION` error is generated by **BeginTransformFeedback** if no binding points would be used, either because no program object is

active or because the active program object has specified no output variables to record.

When **BeginTransformFeedback** is called with an active program object containing a vertex, tessellation or geometry shader, the set of output variables captured during transform feedback is taken from the active program object and may not be changed while transform feedback is active. That program object must be active until the **EndTransformFeedback** is called, except while the transform feedback object is paused.

Errors

An `INVALID_OPERATION` error is generated by:

- **UseProgram** if the current transform feedback object is active and not paused;
- **UseProgramStages** if the program pipeline object it refers to is current and the current transform feedback object is active and not paused;
- **BindProgramPipeline** if the current transform feedback object is active and not paused;
- **LinkProgram** or **ProgramBinary** if *program* is the name of a program being used by one or more transform feedback objects, even if the objects are not currently bound or are paused;
- **ResumeTransformFeedback** if the program object being used by the current transform feedback object is not active;
- **ResumeTransformFeedback** if the program pipeline object being used by the current transform feedback object is not bound, if any of its shader stage bindings has changed, or if a single program object is active and overriding it; and
- **BindBufferRange** or **BindBufferBase** if *target* is `TRANSFORM_FEEDBACK_BUFFER` and transform feedback is currently active.

Buffers should not be simultaneously used for both transform feedback and other purposes in the GL. Specifically, if a buffer object is bound to any of the binding points in an active and not paused transform feedback object, only writes to that binding point are supported, and any other writes to or reads from the buffer generate undefined values. This includes writes to another transform feedback

binding point when the same buffer is bound to more than one binding point in the same transform feedback object. Other examples of reads or writes that could generate undefined values include **DrawPixels** and **ReadPixels** to a pixel buffer object binding point and client access to a buffer mapped with **MapBuffer**. Commands that attempt to read or write to an active and unpaused transform feedback buffer will have undefined results. Generating an `INVALID_OPERATION` error is recommended in this case.

However, if a buffer object is written and read sequentially by transform feedback and other mechanisms, it is the responsibility of the GL to ensure that data are accessed consistently, even if the implementation performs the operations in a pipelined manner. For example, **MapBuffer** may need to block pending the completion of a previous transform feedback operation.

13.3.3 Transform Feedback Draw Operations

When transform feedback is active, the values of output variables or transformed vertex attributes are captured into the buffer objects attached to the current transform feedback object. After transform feedback is complete, subsequent rendering operations may use the contents of these buffer objects (see section 6). The number of vertices captured from each vertex stream during transform feedback is stored in the corresponding transform feedback object and may be used in conjunction with the commands

```
void DrawTransformFeedback( enum mode, uint id );
void DrawTransformFeedbackInstanced( enum mode,
  uint id, sizei instancecount );
void DrawTransformFeedbackStream( enum mode, uint id,
  uint stream );
void DrawTransformFeedbackStreamInstanced( enum mode,
  uint id, uint stream, sizei instancecount );
```

to replay the captured vertices.

DrawTransformFeedbackStreamInstanced is equivalent to calling **DrawArraysInstanced** with *mode* as specified, *first* set to zero, *count* set to the number of vertices captured from the vertex stream numbered *stream* the last time transform feedback was active on the transform feedback object named *id*, and *instancecount* as specified.

Calling **DrawTransformFeedbackInstanced** is equivalent to calling **DrawTransformFeedbackStreamInstanced** with *stream* set to zero.

Calling **DrawTransformFeedbackStream** is equivalent to calling **DrawTransformFeedbackStreamInstanced** with *instancecount* set to one.

Finally, calling **DrawTransformFeedback** is equivalent to calling **DrawTransformFeedbackStreamInstanced** with *stream* set to zero and *instancecount* set to one.

Note that the vertex count is from the number of vertices recorded to the selected vertex stream during the transform feedback operation. If no outputs belonging to the selected vertex stream are recorded, the corresponding vertex count will be zero even if complete primitives were emitted to the selected stream.

No error is generated if the transform feedback object named by *id* is active; the vertex count used for the rendering operation is set by the previous **EndTransformFeedback** command.

Errors

An `INVALID_VALUE` error is generated if *stream* is greater than or equal to the value of `MAX_VERTEX_STREAMS`.

An `INVALID_VALUE` error is generated if *id* is not the name of a transform feedback object.

An `INVALID_VALUE` error is generated if *instancecount* is negative.

An `INVALID_OPERATION` error is generated if **EndTransformFeedback** has never been called while the object named by *id* was bound.

13.4 Primitive Queries

Primitive queries use query objects to track the number of primitives in each vertex stream that are generated by the GL and the number of primitives in each vertex stream that are written to buffer objects in transform feedback mode.

When **BeginQueryIndexed** is called with a *target* of `PRIMITIVES_GENERATED`, the primitives generated count maintained by the GL for the vertex stream *index* is set to zero. There is a separate query and counter for each vertex stream. The number of vertex streams is given by the value of the implementation-dependent constant `MAX_VERTEX_STREAMS`. When a generated primitive query for a vertex stream is active, the primitives-generated count is incremented every time a primitive emitted to that stream reaches the transform feedback stage (see section 13.3), whether or not transform feedback is active. This counter counts the number of primitives emitted by a geometry shader, if active, possibly further tessellated into separate primitives during the transform feedback stage, if active.

When **BeginQueryIndexed** is called with a *target* of `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`, the transform feedback primitives written count maintained by the GL for vertex stream *index* is set to zero. There is a

separate query and counter for each vertex stream. When a transform feedback primitives written query for a vertex stream is active, the counter for that vertex stream is incremented every time the vertices of a primitive written to that stream are recorded into one or more buffer objects. If transform feedback is not active or if a primitive to be recorded does not fit in a buffer object, the counter is not incremented.

These two types of queries can be used together to determine if all primitives in a given vertex stream have been written to the bound feedback buffers; if both queries are run simultaneously and the query results are equal, all primitives have been written to the buffer(s). If the number of primitives written is less than the number of primitives generated, one or more buffers overflowed.

13.5 Transform Feedback Overflow Queries

Transform feedback overflow queries use query objects to track whether or not all stream vertices were written to buffer objects in transform feedback mode.

When **BeginQueryIndexed** is called with a *target* of `TRANSFORM_FEEDBACK_OVERFLOW`, the transform feedback overflow state maintained by the GL is set to false. If transform feedback is active and a primitive to be recorded on any stream does not fit in one of the buffer objects used for capture, the overflow state is set to true.

When **BeginQueryIndexed** is called with a *target* of `TRANSFORM_FEEDBACK_STREAM_OVERFLOW`, the transform feedback stream overflow state maintained by the GL for vertex stream *index* is set to false. There is a separate overflow state for each vertex stream. If transform feedback is active and a primitive to be recorded on stream *index* does not fit in one of the buffer objects used for capture, the overflow state for vertex stream *index* is set to true.

13.6 Flatshading

Flatshading an output of the last vertex processing stage means to assign all vertices of the primitive the same value for that output.

For fixed-function vertex processing, *flatshading* a primitive means to assign all vertices of the primitive the same primary and secondary colors (in RGBA mode) or the same color index (in color index mode).

The color and/or output values assigned are those of the *provoking vertex* of the primitive. The provoking vertex is controlled with the command

`void` **ProvokingVertex**(`enum` *provokeMode*);

Primitive type of polygon i	First vertex convention	Last vertex convention
point	i	i
independent line	$2i - 1$	$2i$
line loop	i	$i + 1$, if $i < n$ 1, if $i = n$
line strip	i	$i + 1$
independent triangle	$3i - 2$	$3i$
triangle strip	i	$i + 2$
triangle fan	$i + 1$	$i + 2$
independent quad	$4i - 3$ $4i$	$4i$ [1] $4i$ [2]
quad strip	$2i - 1$ $2i + 2$	$2i + 2$ [1] $2i + 2$ [2]
single polygon ($i = 1$)	1	1
line adjacency	$4i - 2$	$4i - 1$
line strip adjacency	$i + 1$	$i + 2$
triangle adjacency	$6i - 5$	$6i - 1$
triangle strip adjacency	$2i - 1$	$2i + 3$

Table 13.2: Provoking vertex selection. The vertex colors and/or output values used for flatshading the ith primitive generated by drawing commands with the indicated primitive type are derived from the corresponding values of the vertex whose index is shown in the table. Vertices are numbered 1 through n, where n is the number of vertices drawn.
[1] If the value of QUADS_FOLLOW_PROVOKING_VERTEX_CONVENTION is TRUE.
[2] If the value of QUADS_FOLLOW_PROVOKING_VERTEX_CONVENTION is FALSE.

provokeMode must be either FIRST_VERTEX_CONVENTION or LAST_VERTEX_CONVENTION, and controls selection of the vertex whose values are assigned to flatshaded colors and outputs, as shown in table 13.2.

The provoking vertex behavior of quad primitives is implementation-dependent, and may be determined by calling **GetBooleanv** with *pname* QUADS_FOLLOW_PROVOKING_VERTEX_CONVENTION. A return value of TRUE indicates that the provoking vertex mode is respected for quad primitives, while a return value of FALSE indicates that the implementation always behave as though the provoking vertex mode were LAST_VERTEX_CONVENTION.

Flatshading of colors in fixed-function vertex processing, and of the built-in outputs gl_FrontColor, gl_BackColor, gl_FrontSecondaryColor and

gl_BackSecondaryColor when a vertex processing stage is active, is controlled with the command

void **ShadeModel**(enum *mode*);

mode must be SMOOTH or FLAT. If *mode* is SMOOTH, vertex colors are treated individually. If *mode* is FLAT, flatshading is enabled and colors are taken from the provoking vertex of the primitive. The colors selected are those derived from current values, generated by lighting, or generated by the last vertex processing stage, if lighting is disabled, enabled, or a vertex processing stage is active, respectively.

When a vertex processing stage is active, user-defined output variables may be flatshaded by using the flat qualifier when declaring the output, as described in section 4.5("Interpolation Qualifiers") of the OpenGL Shading Language Specification.

The state required for flatshading is one bit for the shade mode, one bit for the implementation-dependent provoking vertex behavior of quad primitives, and one bit for the provoking vertex mode. The initial value of the shade mode is SMOOTH and the initial value of the provoking vertex mode is LAST_VERTEX_CONVENTION.

13.7 Primitive Clipping

The command

void **ClipControl**(enum *origin*, enum *depth*);

controls the clipping volume behavior. *origin* and *depth* specify the clip control origin and depth mode respectively. The interpretation of these parameters is discussed later in this section and in section 13.8.

Errors

An INVALID_ENUM error is generated if *origin* is not LOWER_LEFT or UPPER_LEFT.

An INVALID_ENUM error is generated if *depth* is not NEGATIVE_ONE_TO_ONE or ZERO_TO_ONE.

Primitives are culled against the *cull volume* and then clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by

$$-w_c \leq x_c \leq w_c$$
$$-w_c \leq y_c \leq w_c$$
$$z_{min} \leq z_c \leq w_c.$$

where z_{min} is $-w_c$ when the clip control depth mode is NEGATIVE_ONE_TO_ONE, and 0 when the mode is ZERO_TO_ONE.

This view volume may be further restricted by as many as n client-defined half-spaces. n is an implementation-dependent maximum that must be at least 8, and may be determined by calling **GetIntegerv** with *pname* MAX_COMBINED_CLIP_AND_CULL_DISTANCES.

The cull volume is the intersection of up to the value of MAX_CULL_DISTANCES client-defined half-spaces (if no client-defined cull half-spaces are enabled, culling against the cull volume is skipped). The number of enabled cull half-spaces is determined by the explicit or implicit size of the built-in array gl_CullDistance in the last shader stage before rasterization which has an active program.

A shader may write a single cull distance for each enabled cull half-space to elements of the gl_CullDistance[] array. If the cull distance for any enabled cull half-space is negative for all of the vertices of the primitive under consideration, the primitive is discarded. Otherwise the primitive is clipped against the clip volume as defined below.

The clip volume is the intersection of up to the value of MAX_CLIP_DISTANCES client-defined half-spaces with the view volume (if no client-defined clip half-spaces are enabled, the clip volume is the view volume).

A client-defined half-space may be specified by the coefficients of an equivalent *clip plane* using the command

`void` **ClipPlane**(`enum` *p*, `const double` *eqn[4]*);

The value of the first argument, *p*, is a symbolic constant, CLIP_PLANEi, where i is an integer between 0 and $n-1$, indicating one of n client-defined clip planes. *eqn* is an array of four double-precision floating-point values. These are the coefficients of a plane equation in object coordinates: p_1, p_2, p_3, and p_4 (in that order). The inverse of the current model-view matrix is applied to these coefficients, at the time they are specified, yielding

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 \end{pmatrix} M^{-1}$$

(where M is the current model-view matrix; the resulting plane equation is undefined if M is singular and may be inaccurate if M is poorly-conditioned) to obtain the plane equation coefficients in eye coordinates. All points with eye coordinates

$\begin{pmatrix} x_e & y_e & z_e & w_e \end{pmatrix}^T$ that satisfy

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0$$

lie in the half-space defined by the plane; points that do not satisfy this condition do not lie in the half-space.

When a vertex processing stage is active, the vector $\begin{pmatrix} x_e & y_e & z_e & w_e \end{pmatrix}^T$ is no longer computed. Instead, the value of the `gl_ClipVertex` built-in variable is used in its place. If `gl_ClipVertex` is not written by the last vertex processing stage, its value is undefined, which implies that the results of clipping to any client-defined clip planes are also undefined. The user must ensure that the clip vertex and client-defined clip planes are defined in the same coordinate space.

The last vertex processing stage may, instead of writing to `gl_ClipVertex` write a single clip distance for each enabled clip half-space to elements of the `gl_ClipDistance[]` array. Clip half-space i is then given by the set of points satisfying the inequality

$$c_i(P) \geq 0,$$

where $c_i(P)$ is the value of clip distance i at point P. For point primitives, $c_i(P)$ is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections 14.5 and 14.6.

Client-defined clip half-spaces are enabled or disabled by calling **Enable** or **Disable** with *target* `CLIP_DISTANCE`i, where i is an integer between 0 and $n-1$; specifying a value of i enables or disables the client-defined clip half-space with index i. The constants obey `CLIP_DISTANCE`i = `CLIP_DISTANCE0` + i.

Depth clamping is enabled or disabled by calling **Enable** or **Disable** with *target* `DEPTH_CLAMP`. If depth clamping is enabled, the

$$z_{min} \leq z_c \leq w_c$$

plane equation (see the clip volume definition above) is ignored by view volume clipping (effectively, there is no near or far plane clipping).

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the near and far clip planes, and the client-defined half-spaces; otherwise, it is discarded. In addition, implementations may discard points that lie outside the view volume.

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume, and discards it if it lies entirely outside the volume.

If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $\mathbf{P}$ and the original vertices' coordinates are $\mathbf{P}_1$ and $\mathbf{P}_2$, then t is given by

$$\mathbf{P} = t\mathbf{P}_1 + (1 - t)\mathbf{P}_2.$$

The value of t is used to clip color, secondary color, texture coordinate, fog coordinate, and outputs of the last vertex processing stage as described in section 13.7.1.

If the primitive is a polygon, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Polygon clipping may cause polygon edges to be clipped, but because polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of new vertices into a polygon. Edge flags are associated with these vertices so that edges introduced by clipping are flagged as boundary (edge flag `TRUE`), and so that original edges of the polygon that become cut off at these vertices retain their original flags.

If it happens that a polygon intersects an edge of the clip volume's boundary, then the clipped polygon must include a point on this boundary edge. This point must lie in the intersection of the boundary edge and the convex hull of the vertices of the original polygon. We impose this requirement because the polygon may not be exactly planar.

Primitives rendered with user-defined clip planes must satisfy a complementarity criterion. Suppose a single clip plane with coefficients $\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix}$ (or a number of similarly specified clip planes) is enabled and a series of primitives are drawn. Next, suppose that the original clip plane is respecified with coefficients $\begin{pmatrix} -p'_1 & -p'_2 & -p'_3 & -p'_4 \end{pmatrix}$ (and correspondingly for any other clip planes) and the primitives are drawn again (and the GL is otherwise in the same state). In this case, primitives must not be missing any pixels, nor may any pixels be drawn twice in regions where those primitives are cut by the clip planes.

The state required for clipping is one bit for clip control origin, one bit for clip control depth mode, at least 8 bits indicating which of the client-defined plane equations are enabled, and at least 8 corresponding sets of plane equations (each consisting of four double-precision floating-point coefficients) In the initial state,

the clip control origin is LOWER_LEFT, the depth mode is NEGATIVE_ONE_TO_ONE, all plane equations are disabled, and all client-defined plane equation coefficients are zero.

Implementations are allowed to pass incoming primitives unchanged and to output multiple primitives for an incoming primitive due to implementation dependent reasons as long as the results of rendering otherwise remain unchanged.

13.7.1 Color and Associated Data Clipping

After lighting, clamping or masking and possible flatshading, colors are clipped. Those colors associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the colors assigned to vertices produced by clipping are clipped.

Let the colors assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (section 13.7) for a clipped point $\mathbf{P}$ is used to obtain the color associated with $\mathbf{P}$ as

$$\mathbf{c} = t\mathbf{c}_1 + (1 - t)\mathbf{c}_2.$$

(For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For an RGBA color, it means multiplying each of R, G, B, and A by the scalar. Both primary and secondary colors are treated in the same fashion.)

Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Color clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

Texture and fog coordinates, outputs of the last vertex processing stage (section 11.1.2.1), and point sizes computed on a per vertex basis must also be clipped when a primitive is clipped. The method is exactly analogous to that used for color clipping.

For outputs of the last vertex processing stage specified to be interpolated without perspective correction (using the noperspective qualifier), the value of t used to obtain the output value associated with $\mathbf{P}$ will be adjusted to produce results that vary linearly in screen space.

Implementations need not support interpolation of output values of integer or unsigned integer type, as all such attributes must be flat shaded.

13.7.2 Clip Plane Queries

The command

```
void GetClipPlane( enum plane, double eqn[4] );
```

returns four double-precision values in *eqn*; these are the coefficients of the plane equation of *plane* in eye coordinates (these coordinates are those that were computed when the plane was specified).

Errors

An INVALID_ENUM error is generated if *plane* is not one of CLIP_DISTANCEi, where i is an integer between 0 and $n - 1$.

13.7.3 Primitive Clipping Queries

Primitive clipping queries use query objects to track the number of primitives that are processed by the primitive clipping stage and the number of primitives that are output by the primitive clipping stage and are further processed by the rasterization stage.

When **BeginQuery** is called with a *target* of CLIPPING_INPUT_PRIMITIVES, the clipping input primitives count maintained by the GL is set to zero. When a clipping input primitives query is active, the counter is incremented every time a primitive reaches the primitive clipping stage (see section 13.7).

When **BeginQuery** is called with a *target* of CLIPPING_OUTPUT_PRIMITIVES, the clipping output primitives count maintained by the GL is set to zero. When a clipping output primitives query is active, the counter is incremented every time a primitive passes the primitive clipping stage. The actual number of primitives output by the primitive clipping stage for a particular input primitive is implementation dependent (see section 13.7) but must satisfy the following conditions:

- If at least one vertex of the input primitive lies inside the clipping volume, the counter is incremented by one or more.
- Otherwise, the counter is incremented by zero or more.

If RASTERIZER_DISCARD is enabled, implementations are allowed to discard primitives right after the optional transform feedback state (see section 14.1). As a result, if RASTERIZER_DISCARD is enabled, the clipping input and output primitives count may not be incremented.

13.8 Coordinate Transformations

Clip coordinates for a vertex result from fixed-function transformation of the vertex coordinates, or from shader execution, which yields a vertex coordinate `gl_Position`.

Perspective division on clip coordinates yields *normalized device coordinates*, followed by a *viewport* transformation (see section 13.8.1) to convert these coordinates into *window coordinates*.

If a vertex in clip coordinates is given by

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$$

then the vertex's normalized device coordinates are

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{f \times y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

where f is 1 when the clip control origin is `LOWER_LEFT` and -1 when the origin is `UPPER_LEFT`.

13.8.1 Controlling the Viewport

The viewport transformation is determined by the selected viewport's width and height in pixels, p_x and p_y, respectively, and its center (o_x, o_y) (also in pixels). The vertex's window coordinates, $\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix}$, are given by

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_d + o_x \\ \frac{p_y}{2} y_d + o_y \\ s \times z_d + b \end{pmatrix}.$$

where $s = \frac{f-n}{2}$ and $b = \frac{n+f}{2}$ when the clip control depth mode is `NEGATIVE_ONE_TO_ONE`, or $s = f - n$ and $b = n$ when the mode is `ZERO_TO_ONE`.

Multiple viewports are available and are numbered zero through the value of `MAX_VIEWPORTS` minus one. If a geometry shader is active and writes to `gl_ViewportIndex`, the viewport transformation uses the viewport corresponding

to the value assigned to gl_ViewportIndex taken from an implementation-dependent primitive vertex. If the value of the viewport index is outside the range zero to the value of MAX_VIEWPORTS minus one, the results of the viewport transformation are undefined. If no geometry shader is active, or if the active geometry shader does not write to gl_ViewportIndex, the viewport numbered zero is used by the viewport transformation.

A single vertex may be used in more than one individual primitive, in primitives such as TRIANGLE_STRIP. In this case, the viewport transformation is applied separately for each primitive.

The factor and offset applied to z_d for each viewport encoded by n and f are set using

```
void DepthRangeArrayv( uint first, sizei count, const
   double *v );
void DepthRangeIndexed( uint index, double n,
   double f );
void DepthRange( double n, double f );
void DepthRangef( float n, float f );
```

DepthRangeArrayv is used to specify the depth range for multiple viewports simultaneously. *first* specifies the index of the first viewport to modify and *count* specifies the number of viewports. Viewports whose indices lie outside the range $[first, first + count)$ are not modified. The *v* parameter contains the address of an array of **double** types specifying near (*n*) and far (*f*) for each viewport in that order. Values in *v* are each clamped to the range $[0, 1]$ when specified.

Errors

An INVALID_VALUE error is generated if (*first* + *count*) is greater than the value of MAX_VIEWPORTS.

An INVALID_VALUE error is generated if *count* is negative.

DepthRangeIndexed specifies the depth range for a single viewport and is equivalent (assuming no errors are generated) to:

```
double v[] = { n, f };
DepthRangeArrayv(index, 1, v);
```

DepthRange sets the depth range for all viewports to the same values and is equivalent (assuming no errors are generated) to:

```
for (uint i = 0; i < MAX_VIEWPORTS; i++)
   DepthRangeIndexed(i, n, f);
```

z_w may be represented using either a fixed-point or floating-point representation. However, a floating-point representation must be used if the draw framebuffer has a floating-point depth buffer. If an m-bit fixed-point representation is used, we assume that it represents each value $\frac{k}{2^m-1}$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

Viewport transformation parameters are specified using

```
void ViewportArrayv( uint first, sizei count, const
   float *v );
void ViewportIndexedf( uint index, float x, float y,
   float w, float h );
void ViewportIndexedfv( uint index, const float *v );
void Viewport( int x, int y, sizei w, sizei h );
```

ViewportArrayv specifies parameters for multiple viewports simultaneously. *first* specifies the index of the first viewport to modify and *count* specifies the number of viewports. Viewports whose indices lie outside the range $[first, first + count)$ are not modified. *v* contains the address of an array of floating-point values specifying the left (*x*), bottom (*y*), width (*w*) and height (*h*) of each viewport, in that order. *x* and *y* give the location of the viewport's lower left corner and *w* and *h* give the viewport's width and height, respectively.

Errors

An INVALID_VALUE error is generated if $first + count$ is greater than the value of MAX_VIEWPORTS.

An INVALID_VALUE error is generated if *count* is negative.

ViewportIndexedf and **ViewportIndexedfv** specify parameters for a single viewport and are equivalent (assuming no errors are generated) to:

```
float v[4] = { x, y, w, h };
ViewportArrayv(index, 1, v);
```

and

```
ViewportArrayv(index, 1, v);
```

respectively.

Viewport sets the parameters for all viewports to the same values and is equivalent (assuming no errors are generated) to:

```
for (uint i = 0; i < MAX_VIEWPORTS; i++)
    ViewportIndexedf(i, 1, (float)x, (float)y, (float)w, (float)h);
```

The viewport parameters shown in the above equations are found from these values as

$$o_x = x + \frac{w}{2}$$
$$o_y = y + \frac{h}{2}$$
$$p_x = w$$
$$p_y = h.$$

The location of the viewport's bottom-left corner, given by (x, y), are clamped to be within the implementation-dependent viewport bounds range. The viewport bounds range $[min, max]$ tuple may be determined by calling **GetFloatv** with *pname* `VIEWPORT_BOUNDS_RANGE` (see section 22).

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by calling **GetFloatv** with *pname* `MAX_VIEWPORT_DIMS`. The maximum viewport dimensions must be greater than or equal to the larger of the visible dimensions of the display being rendered to (if a display exists), and the largest renderbuffer image which can be successfully created and attached to a framebuffer object (see chapter 9).

Errors

An `INVALID_VALUE` error is generated if either *w* or *h* is negative.

The state required to implement the viewport transformation is four integers and two clamped floating-point values for each viewport. In the initial state, *w* and *h* for each viewport are set to the width and height, respectively, of the window into which the GL is to do its rendering. If the default framebuffer is bound but no default framebuffer is associated with the GL context (see chapter 9), then *w* and *h* are initially set to zero. o_x, o_y, n, and f are set to $\frac{w}{2}$, $\frac{h}{2}$, 0.0, and 1.0, respectively.

The precision with which the GL interprets the floating-point viewport bounds is implementation-dependent and may be determined by querying the implementation-defined constant `VIEWPORT_SUBPIXEL_BITS`.

13.9 Final Color Processing

In RGBA mode with vertex color clamping disabled, the floating-point RGBA components are not modified.

In RGBA mode with vertex color clamping enabled, each color component may be converted to a signed or unsigned normalized fixed-point value as described in equations 2.3 and 2.4 (depending on the framebuffer format).

GL implementations are not required to convert clamped color components to fixed-point.

Because a number of the form $\frac{k}{2^m-1}$ may not be represented exactly as a limited-precision floating-point quantity, we place a further requirement on the fixed-point conversion of RGBA components. Suppose that lighting is disabled, the color associated with a vertex has not been clipped, and one of **Colorub**, **Colorus**, or **Colorui** was used to specify that color. When these conditions are satisfied, an RGBA component must convert to a value that matches the component as specified in the **Color** command: if m is less than the number of bits b with which the component was specified, then the converted value must equal the most significant m bits of the specified value; otherwise, the most significant b bits of the converted value must equal the specified value.

A color index is converted (by rounding to nearest) to a fixed-point value with at least as many bits as there are in the color index portion of the framebuffer.

Chapter 14

Fixed-Function Primitive Assembly and Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth.

Rasterizing a primitive begins by determining which squares of an integer grid in window coordinates are occupied by the primitive, and assigning a depth value to each such square. This process is described in sections 14.1-14.6 for point, line, and triangle primitives.

A grid square, including its (x, y) window coordinates, z (depth), and *associated data* which may be added by fixed-function fragment processing or fragment shaders, is called a *fragment*. A fragment is located by its lower left corner, which lies on integer grid coordinates. Rasterization operations also refer to a fragment's *center*, which is offset by $(\frac{1}{2}, \frac{1}{2})$ from its lower left corner (and so lies on half-integer coordinates).

Fragments need not actually be square, and rasterization rules are not affected by the aspect ratio of fragments. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other. We assume that fragments are square, since it simplifies antialiasing and texturing.

After rasterization, fragments are processed by the early per-fragment tests described in section 14.9, which may modify or discard fragments.

Surviving fragments are processed by either the fixed-function operations in chapter 16, or by fragment shaders (see chapter 15). These steps determine color values for fragments, and may also modify or replace their assigned depth values.

Figure 14.1 diagrams the rasterization process.

Grid squares need not actually be square in the GL. Rasterization rules are not affected by the actual aspect ratio of the grid squares. Display of non-square grids,

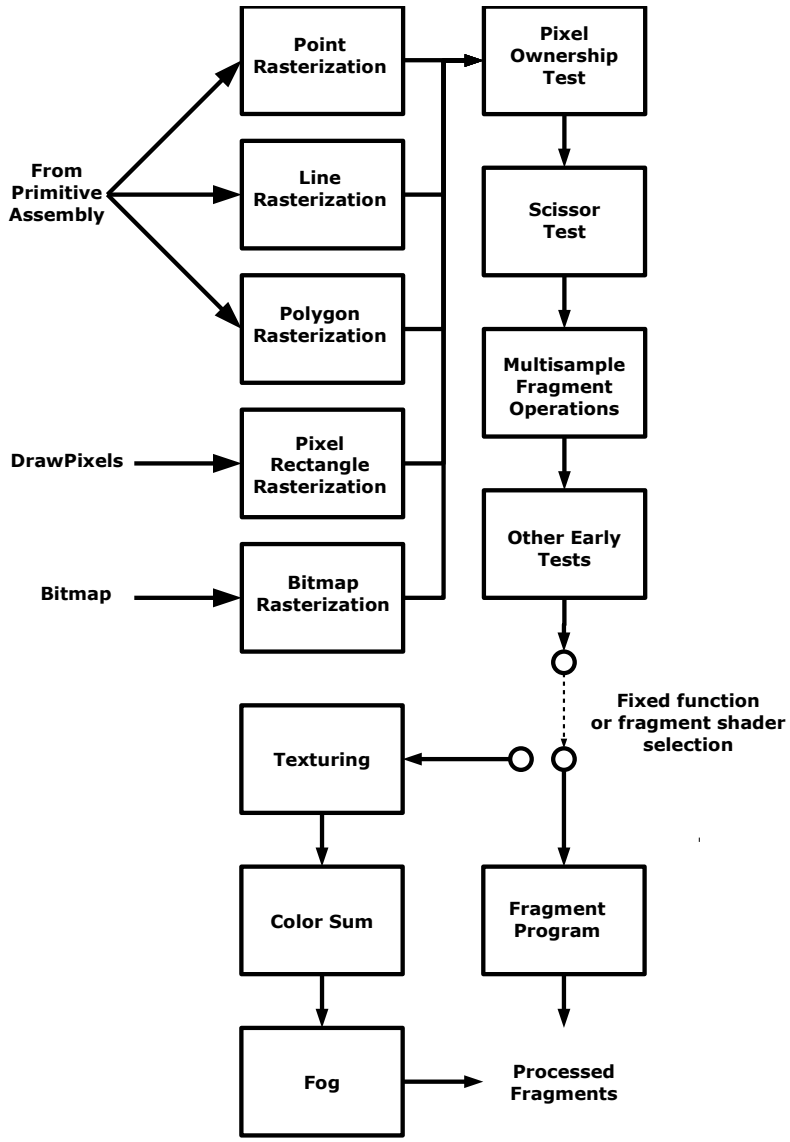


Figure 14.1. Rasterization, early per-fragment tests, and fragment shading. Optional early tests described in section 14.9 are included in the "Other Early Tests" box.

however, will cause rasterized points and line segments to appear fatter in one direction than the other. We assume that fragments are square, since it simplifies antialiasing and texturing.

Several factors affect rasterization. Primitives may be discarded before rasterization. Lines and polygons may be stippled. Points may be given differing diameters and line segments differing widths. A point, line segment, or polygon may be antialiased.

Rasterization only produces fragments corresponding to pixels in the framebuffer. Fragments which would be produced by application of any of the primitive rasterization rules described below but which lie outside the framebuffer are not produced, nor are they processed by any later stage of the GL, including any of the early per-fragment tests described in section 14.9.

14.1 Discarding Primitives Before Rasterization

Primitives sent to vertex stream zero (see section 13.3) are processed further; primitives emitted to any other stream are discarded. When geometry shaders are disabled, all vertices are considered to be emitted to stream zero.

Primitives can be optionally discarded before rasterization by calling **Enable** and **Disable** with *target* `RASTERIZER_DISCARD`. When enabled, primitives are discarded immediately before the rasterization stage, but after the optional transform feedback stage (see section 13.3). When disabled, primitives are passed through to the rasterization stage to be processed normally. When enabled, `RASTERIZER_DISCARD` also causes the **Accum**, **Bitmap**, **CopyPixels**, **DrawPixels**, **Clear**, and **ClearBuffer*** commands to be ignored.

The state required to control primitive discard is a bit indicating whether discard is enabled or disabled. The initial value of primitive discard is `FALSE`.

14.2 Invariance

Consider a primitive p' obtained by translating a primitive p through an offset (x, y) in window coordinates, where x and y are integers. As long as neither p' nor p is clipped, it must be the case that each fragment f' produced from p' is identical to a corresponding fragment f from p except that the center of f' is offset by (x, y) from the center of f.

14.3 Antialiasing

Antialiasing of a point, line, or polygon is effected in one of two ways depending on whether the GL is in RGBA or color index mode.

In RGBA mode, the R, G, and B values of the rasterized fragment are left unaffected, but the A value is multiplied by a floating-point value in the range $[0, 1]$ that describes a fragment's screen pixel coverage. The per-fragment stage of the GL can be set up to use the A value to blend the incoming fragment with the corresponding pixel already present in the framebuffer.

In color index mode, the least significant b bits (to the left of the binary point) of the color index are used for antialiasing; $b = \min(4, m)$, where m is the number of bits in the color index portion of the framebuffer. The antialiasing process sets these b bits based on the fragment's coverage value: the bits are set to zero for no coverage and to all ones for complete coverage.

The details of how antialiased fragment coverage values are computed are difficult to specify in general. The reason is that high-quality antialiasing may take into account perceptual issues as well as characteristics of the monitor on which the contents of the framebuffer are displayed. Such details cannot be addressed within the scope of this document. Further, the coverage value computed for a fragment of some primitive may depend on the primitive's relationship to a number of grid squares neighboring the one corresponding to the fragment, and not just on the fragment's grid square. Another consideration is that accurate calculation of coverage values may be computationally expensive; consequently we allow a given GL implementation to approximate true coverage values by using a fast but not entirely accurate coverage computation.

In light of these considerations, we chose to specify the behavior of exact antialiasing in the prototypical case that each displayed pixel is a perfect square of uniform intensity. The square is called a *fragment square* and has lower left corner (x, y) and upper right corner $(x+1, y+1)$. We recognize that this simple box filter may not produce the most favorable antialiasing results, but it provides a simple, well-defined model.

A GL implementation may use other methods to perform antialiasing, subject to the following conditions:

1. If f_1 and f_2 are two fragments, and the portion of f_1 covered by some primitive is a subset of the corresponding portion of f_2 covered by the primitive, then the coverage computed for f_1 must be less than or equal to that computed for f_2.

2. The coverage computation for a fragment f must be local: it may depend only on f's relationship to the boundary of the primitive being rasterized. It may not depend on f's x and y coordinates.

Another property that is desirable, but not required, is:

3. The sum of the coverage values for all fragments produced by rasterizing a particular primitive must be constant, independent of any rigid motions in window coordinates, as long as none of those fragments lies along window edges.

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 21.5), allowing a user to make an image quality versus speed tradeoff.

14.3.1 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, polygons, bitmaps, and images. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes color, depth, and stencil information, the color (including texture operation), depth, and stencil functions perform equivalently to the single-sample mode.

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. Samples contain separate color values for each fragment color. When the framebuffer includes a multisample buffer, it does not include depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. Color buffers do coexist with the multisample buffer, however.

Multisample antialiasing is most valuable for rendering polygons, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. If only points or lines are being rendered, the "smooth" antialiasing mechanism provided by the base GL may result in a higher quality image. This mechanism is designed to allow multisample and smooth antialiasing techniques to be alternated during the rendering of a single scene.

If the value of `SAMPLE_BUFFERS` (see section 9.2.3.1) is one, the rasterization of all primitives is changed, and is referred to as *multisample rasterization*. Otherwise, primitive rasterization is referred to as *single-sample rasterization*.

During multisample rendering the contents of a pixel fragment are changed in two ways. First, each fragment includes a coverage value with the value of `SAMPLES` bits (see section 9.2.3.1).

The location at which shading is performed for a given sample (the *shading sample location*) is queried with the command

```
void GetMultisamplefv( enum pname, uint index,
  float *val );
```

pname must be SAMPLE_POSITION, and *index* corresponds to the sample for which the location should be returned. The sample location is returned as two floating-point values in *val[0]* and *val[1]*, each between 0 and 1, corresponding to the x and y locations respectively in GL pixel space of that sample. $(0.5, 0.5)$ thus corresponds to the pixel center. If the multisample mode does not have fixed sample locations, the returned values may only reflect the locations of samples within some pixels.

Errors

An INVALID_ENUM error is generated if *pname* is not SAMPLE_POSITION.

An INVALID_VALUE error is generated if *index* is greater than or equal to the value of SAMPLES.

Second, each fragment includes SAMPLES depth values and sets of associated data, instead of the single depth value and set of associated data that is maintained in single-sample rendering mode. An implementation may choose to assign the same associated data to more than one sample. The location for evaluating such associated data can be anywhere within the pixel including the fragment center or any of the sample locations. The different associated data values need not all be evaluated at the same location. Each pixel fragment thus consists of integer x and y grid coordinates, SAMPLES depth values and sets of associated data, and a coverage value with a maximum of SAMPLES bits.

Multisample rasterization is enabled or disabled by calling **Enable** or **Disable** with *target* MULTISAMPLE.

If MULTISAMPLE is disabled, multisample rasterization of all primitives is equivalent to single-sample (fragment-center) rasterization, except that the fragment coverage value is set to full coverage. The color and depth values and the sets of texture coordinates may all be set to the values that would have been assigned by single-sample rasterization, or they may be assigned as described below for multisample rasterization.

If MULTISAMPLE is enabled, multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has sample locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a *sample point*. These sample points do not necessarily correspond to the shading sample locations

returned by **GetMultisamplefv**. Their locations cannot be queried, and may lie inside or outside of the unit square that is considered to bound the pixel. The number of these samples may be different than the value of SAMPLES. Furthermore, the relative locations of sample points may be identical for each pixel in the framebuffer, or they may differ.

If MULTISAMPLE is enabled and the current program object includes a fragment shader with one or more input variables qualified with `sample in`, the data associated with those variables will be assigned independently. The values for each sample must be evaluated at the location of the sample. The data associated with any other variables not qualified with `sample in` need not be evaluated independently for each sample.

If the sample locations differ per pixel, they should be aligned to window, not screen, boundaries. Otherwise rendering results will be window-position specific. The invariance requirement described in section 14.2 is relaxed for all multisample rasterization, because the sample locations may be a function of pixel location.

14.3.1.1 Sample Shading

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. Sample shading is controlled by calling **Enable** or **Disable** with *target* SAMPLE_SHADING.

If MULTISAMPLE or SAMPLE_SHADING is disabled, sample shading has no effect. Otherwise, an implementation must provide a minimum of

$$\max(\lceil mss \times samples \rceil, 1)$$

unique sets of fragment shader inputs for each fragment, where *mss* is the value of MIN_SAMPLE_SHADING_VALUE and *samples* is the number of samples (the value of SAMPLES). These are associated with the samples in an implementation-dependent manner. The value of MIN_SAMPLE_SHADING_VALUE is specified by calling

`void` **MinSampleShading**(`float` *value*);

with *value* set to the desired minimum sample shading fraction. *value* is clamped to $[0, 1]$ when specified. The sample shading fraction may be queried by calling **GetFloatv** with *pname* MIN_SAMPLE_SHADING_VALUE.

When the sample shading fraction is 1.0, a separate set of colors and other associated data are evaluated for each sample, and each set of values is evaluated at the sample location.

14.4 Points

A point is drawn by generating a set of fragments in the shape of a square or circle centered around the vertex of the point. Each vertex has an associated point size that controls the size of that square or circle.

If no vertex, tessellation control, tessellation evaluation, or geometry shader is active, then the rasterization of points is controlled with

`void` **PointSize**(`float` *size*);

size specifies the requested size of a point. The default value is 1.0.

Errors

An `INVALID_VALUE` error is generated if *size* is less than or equal to zero.

The requested point size is multiplied with a distance attenuation factor, clamped to a specified point size range, and further clamped to the implementation-dependent point size range to produce the derived point size:

$$derived_size = clamp\left(size \times \sqrt{\left(\frac{1}{a + b * d + c * d^2}\right)}\right)$$

where d is the eye-coordinate distance from the eye, $(0, 0, 0, 1)$ in eye coordinates, to the vertex, and a, b, and c are distance attenuation function coefficients.

If multisampling is not enabled, the derived size is passed on to rasterization as the point width.

If a vertex processing stage is active and program point size mode is enabled, the derived point size is taken from the (potentially clipped) shader built-in `gl_PointSize` written by the last vertex processing stage and clamped to the implementation-dependent point size range. If the value written to `gl_PointSize` is less than or equal to zero, or if no value was written to `gl_PointSize`, results are undefined. If a vertex, tessellation control, tessellation evaluation, or geometry shader is active and program point size mode is disabled, the derived point size is taken from the point size state as specified by the **PointSize** command. In this case no distance attenuation is performed.

Program point size mode is enabled and disabled by calling **Enable** or **Disable** with *target* `PROGRAM_POINT_SIZE`.

If multisampling is enabled, an implementation may optionally fade the point alpha (see section 17.2) instead of allowing the point width to go below a given

threshold. In this case, the width of the rasterized point is

$$width = \begin{cases} derived_size & derived_size \geq threshold \\ threshold & otherwise \end{cases} \tag{14.1}$$

and the fade factor is computed as follows:

$$fade = \begin{cases} 1 & derived_size \geq threshold \\ \left(\frac{derived_size}{threshold}\right)^2 & otherwise \end{cases} \tag{14.2}$$

The point fade *threshold*, the distance attenuation function coefficients a, b, and c, and the bounds of the first point size range clamp are specified with

```
void PointParameter{if}( enum pname, T param );
void PointParameter{if}v( enum pname, const T *params );
```

If *pname* is POINT_FADE_THRESHOLD_SIZE, then *param* specifies, or *params* points to the point fade *threshold*. If *pname* is POINT_DISTANCE_ATTENUATION, then *params* points to the coefficients a, b, and c. If *pname* is POINT_SIZE_MIN or POINT_SIZE_MAX, then *param* specifies, or *params* points to the lower or upper bound respectively to which the derived point size is clamped. If the lower bound is greater than the upper bound, the point size after clamping is undefined.

Data conversions are performed as specified in section 2.2.1.

The point sprite texture coordinate origin is set with the **PointParameter*** commands where *pname* is POINT_SPRITE_COORD_ORIGIN and *param* is LOWER_LEFT or UPPER_LEFT. The default value is UPPER_LEFT.

Errors

An INVALID_ENUM error is generated if *pname* is not POINT_FADE_THRESHOLD_SIZE, POINT_DISTANCE_ATTENUATION, POINT_SIZE_MIN, POINT_SIZE_MAX, or POINT_SPRITE_COORD_ORIGIN.

An INVALID_VALUE error is generated if negative values are specified for POINT_SIZE_MIN, POINT_SIZE_MAX, or POINT_FADE_THRESHOLD_SIZE.

Point antialiasing is enabled or disabled by calling **Enable** or **Disable** with *target* POINT_SMOOTH. The default state is for point antialiasing to be disabled.

Point sprites are enabled or disabled by calling **Enable** or **Disable** with *target* POINT_SPRITE. The default state is for point sprites to be disabled. When

point sprites are enabled, the state of the point antialiasing enable is ignored. In a deprecated context, point sprites are always enabled.

The point sprite texture coordinate replacement mode is set with one of the **TexEnv*** commands described in section 16.1, where *target* is `POINT_SPRITE` and *pname* is `COORD_REPLACE`. The possible values for *param* are `FALSE` and `TRUE`. The default value for each texture coordinate set is for point sprite texture coordinate replacement to be disabled.

14.4.1 Basic Point Rasterization

In the default state, a point is rasterized by truncating its x_w and y_w coordinates (recall that the subscripts indicate that these are x and y window coordinates) to integers. This (x, y) address, along with data derived from the data associated with the vertex corresponding to the point, is sent as a single fragment to the per-fragment stage of the GL.

The effect of a point width other than 1.0 depends on the state of point antialiasing and point sprites. If antialiasing and point sprites are disabled, the actual width is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased point width. This implementation-dependent value must be no less than the implementation-dependent maximum antialiased point width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value was 1. If the resulting width is odd, then the point

$$(x, y) = (\lfloor x_w \rfloor + \frac{1}{2}, \lfloor y_w \rfloor + \frac{1}{2})$$

is computed from the vertex's x_w and y_w, and a square grid of the odd width centered at (x, y) defines the centers of the rasterized fragments (recall that fragment centers lie at half-integer window coordinate values). If the width is even, then the center point is

$$(x, y) = (\lfloor x_w + \frac{1}{2} \rfloor, \lfloor y_w + \frac{1}{2} \rfloor);$$

the rasterized fragment centers are the half-integer window coordinate values within the square of the even width centered on (x, y). See figure 14.2.

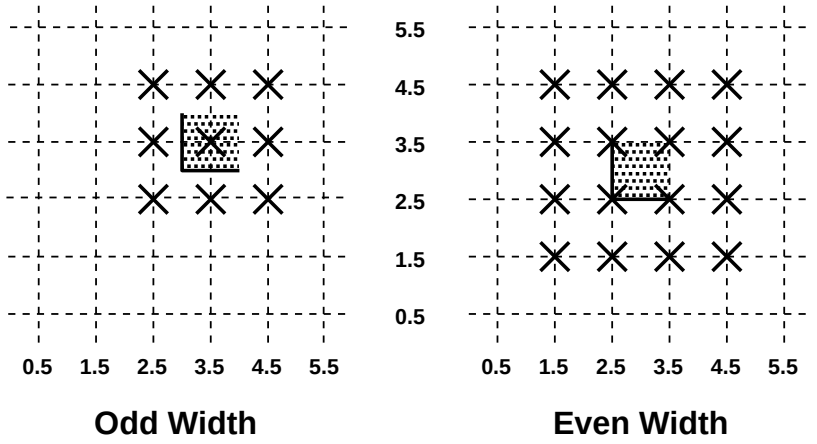


Figure 14.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.

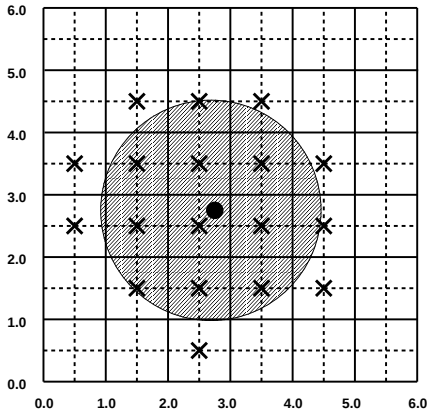


Figure 14.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region that covers the corresponding fragment square. Solid lines lie on integer coordinates.

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point.

If antialiasing is enabled and point sprites are disabled, then point rasterization produces a fragment for each fragment square that intersects the region lying within the circle having diameter equal to the current point width and centered at the point's (x_w, y_w) (figure 14.3). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding fragment square (but see section 14.3). This value is saved and used in the final step of rasterization (section 17.1). The data associated with each fragment are otherwise the data associated with the point being rasterized.

Not all widths need be supported when point antialiasing is on, but the width 1.0 must be provided. If an unsupported width is requested, the nearest supported width is used instead. The range of supported widths and the width of evenly-spaced gradations within that range are implementation-dependent. The range and gradations may be obtained using the query mechanism described in chapter 22. If, for instance, the width range is from 0.1 to 2.0 and the gradation width is 0.1, then the widths $0.1, 0.2, \ldots, 1.9, 2.0$ are supported.

If point sprites are enabled, then point rasterization produces a fragment for each framebuffer pixel whose center lies inside a square centered at the point's (x_w, y_w), with side length equal to the current point size.

All fragments produced in rasterizing a point sprite are assigned the same associated data, which are those of the vertex corresponding to the point. However, the fragment shader built-in `gl_PointCoord` contains point sprite texture coordinates. Additionally, for each texture coordinate set where `COORD_REPLACE` is `TRUE`, these texture coordinates are replaced with point sprite texture coordinates. The s point sprite texture coordinate varies from zero to one across the point horizontally left-to-right. If `POINT_SPRITE_COORD_ORIGIN` is `LOWER_LEFT`, the t coordinate varies from zero to one vertically bottom-to-top. Otherwise if the point sprite texture coordinate origin is `UPPER_LEFT`, the t coordinate varies from zero to one vertically top-to-bottom. The r and q coordinates are replaced with the constants zero and one, respectively. The following formula is used to evaluate the s and t point sprite texture coordinates:

$$s = \frac{1}{2} + \frac{\left(x_f + \frac{1}{2} - x_w\right)}{size} \tag{14.3}$$

$$t = \begin{cases} \frac{1}{2} + \frac{\left(y_f + \frac{1}{2} - y_w\right)}{size}, & \texttt{POINT_SPRITE_COORD_ORIGIN} = \texttt{LOWER_LEFT} \\ \frac{1}{2} - \frac{\left(y_f + \frac{1}{2} - y_w\right)}{size}, & \texttt{POINT_SPRITE_COORD_ORIGIN} = \texttt{UPPER_LEFT} \end{cases} \tag{14.4}$$

where $size$ is the point's size, x_f and y_f are the (integral) window coordinates of the fragment, and x_w and y_w are the exact, unrounded window coordinates of the vertex for the point.

The widths supported for point sprites must be a superset of those supported for antialiased points. There is no requirement that these widths must be equally spaced. If an unsupported width is requested, the nearest supported width is used instead.

14.4.2 Point Rasterization State

The state required to control point rasterization consists of the floating-point point width, two floating-point values specifying the minimum and maximum point size, three floating-point values specifying the distance attenuation coefficients, a bit indicating whether or not antialiasing is enabled, a bit indicating whether or not point sprites are enabled, a bit for the point sprite texture coordinate replacement mode for each texture coordinate set, a bit indicating whether or not vertex program point size mode is enabled, a bit for the point sprite texture coordinate origin, and a floating-point value specifying the point fade threshold size.

14.4.3 Point Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (`POINT_SMOOTH`) is enabled or disabled. Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point's (x_w, y_w). This region is a circle having diameter equal to the current point width if `POINT_SPRITE` is disabled, or a square with side equal to the current point width if `POINT_SPRITE` is enabled. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All data associated with each sample for the fragment are the data associated with the point being rasterized, with the exception of texture coordinates when `POINT_SPRITE` is enabled; these texture coordinates are computed as described in section 14.4.

When `POINT_SPRITE` is disabled, the point size range and number of gradations supported are equivalent to those supported for antialiased points When `POINT_SPRITE` is enabled, the set of point sizes supported is equivalent to those for point sprites without multisample.

14.5 Line Segments

A line segment results from a line strip **Begin / End** object, a line loop, or a series of separate line segments. Line segment rasterization is controlled by several variables. Line width, which may be set by calling

`void` **LineWidth**(`float` *width*);

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is 1.0. Antialiasing may be enabled or disabled by calling **Enable** or **Disable** with *target* `LINE_SMOOTH`.

Errors

An `INVALID_VALUE` error is generated if *width* is less than or equal to zero.

Line segments may be stippled. Stippling is controlled by a GL command that sets a *stipple pattern* (see below).

14.5.1 Basic Line Segment Rasterization

Line segment rasterization begins by characterizing the segment as either *x-major* or *y-major*. x-major line segments have slope in the closed interval $[-1, 1]$; all other line segments are y-major (slope is determined by the segment's endpoints). We shall specify rasterization only for x-major segments except in cases where the modifications for y-major segments are not self-evident.

Ideally, the GL uses a "diamond-exit" rule to determine those fragments that are produced by rasterizing a line segment. For each fragment f with center at window coordinates x_f and y_f, define a diamond-shaped region that is the intersection of four half planes:

$$R_f = \{\, (x, y) \mid \; |x - x_f| + |y - y_f| < \frac{1}{2} \,\}.$$

Essentially, a line segment starting at $\mathbf{p}_a$ and ending at $\mathbf{p}_b$ produces those fragments f for which the segment intersects R_f, except if $\mathbf{p}_b$ is contained in R_f. See figure 14.4.

To avoid difficulties when an endpoint lies on a boundary of R_f we (in principle) perturb the supplied endpoints by a tiny amount. Let $\mathbf{p}_a$ and $\mathbf{p}_b$ have window coordinates (x_a, y_a) and (x_b, y_b), respectively. Obtain the perturbed endpoints $\mathbf{p}'_a$ given by $(x_a, y_a) - (\epsilon, \epsilon^2)$ and $\mathbf{p}'_b$ given by $(x_b, y_b) - (\epsilon, \epsilon^2)$. Rasterizing the line

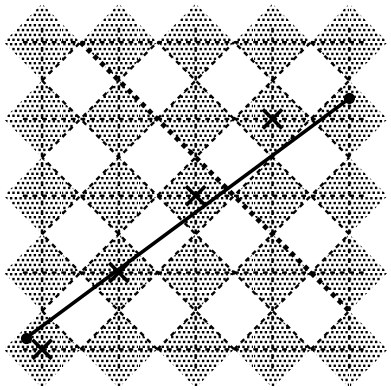


Figure 14.4. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

segment starting at $\mathbf{p}_a$ and ending at $\mathbf{p}_b$ produces those fragments f for which the segment starting at $\mathbf{p}'_a$ and ending on $\mathbf{p}'_b$ intersects R_f, except if $\mathbf{p}'_b$ is contained in R_f. ϵ is chosen to be so small that rasterizing the line segment produces the same fragments when δ is substituted for ϵ for any $0 < \delta \leq \epsilon$.

When $\mathbf{p}_a$ and $\mathbf{p}_b$ lie on fragment centers, this characterization of fragments reduces to Bresenham's algorithm with one modification: lines produced in this description are "half-open," meaning that the final fragment (corresponding to $\mathbf{p}_b$) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham's algorithm).

Because the initial and final conditions of the diamond-exit rule may be difficult to implement, other line segment rasterization algorithms are allowed, subject to the following rules:

1. The coordinates of a fragment produced by the algorithm may not deviate by more than one unit in either x or y window coordinates from a corresponding fragment produced by the diamond-exit rule.

2. The total number of fragments produced by the algorithm may differ from that produced by the diamond-exit rule by no more than one.

3. For an x-major line, no two fragments may be produced that lie in the same window-coordinate column (for a y-major line, no two fragments may appear in the same row).

4. If two line segments share a common endpoint, and both segments are either x-major (both left-to-right or both right-to-left) or y-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments may not produce duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

Next we must specify how the data associated with each rasterized fragment are obtained. Let the window coordinates of a produced fragment center be given by $\mathbf{p}_r = (x_d, y_d)$ and let $\mathbf{p}_a = (x_a, y_a)$ and $\mathbf{p}_b = (x_b, y_b)$. Set

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}. \tag{14.5}$$

(Note that $t = 0$ at $\mathbf{p}_a$ and $t = 1$ at $\mathbf{p}_b$). The value of an associated datum f for the fragment, whether it be primary or secondary R, G, B, or A (in RGBA mode) or a color index (in color index mode), the fog coordinate, an s, t, r, or q texture coordinate, or the clip w coordinate, is found as

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b} \tag{14.6}$$

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments, respectively. However, depth values for lines must be interpolated by

$$z = (1-t)z_a + tz_b \tag{14.7}$$

where z_a and z_b are the depth values of the starting and ending endpoints of the segment, respectively.

When a vertex processing stage is active, the `noperspective` and `flat` keywords used to declare shader outputs affect how they are interpolated. When neither keyword is specified, interpolation is performed as described in equation 14.6. When the `noperspective` keyword is specified, interpolation is performed in the same fashion as for depth values, as described in equation 14.7. When the `flat` keyword is specified, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive (see section 13.6).

14.5.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one using the default line stipple of $FFFF_{16}$. We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

14.5.2.1 Line Stipple

The command

`void` **LineStipple**(`int` *factor*, `ushort` *pattern*);

defines a *line stipple*. *pattern* is an unsigned short integer. The line stipple is taken from the lowest order 16 bits of *pattern*. It determines those fragments that are to be drawn when the line is rasterized. *factor* is a count that is used to modify the effective line stipple by causing each bit in *pattern* to be used *factor* times. *factor* is clamped to the range $[1, 256]$. Line stippling may be enabled or disabled by calling **Enable** or **Disable** with *target* `LINE_STIPPLE`. When disabled, it is as if the line stipple has its default value.

Line stippling masks certain fragments that are produced by rasterization so that they are not sent to the per-fragment stage of the GL. The masking is achieved using three parameters: the 16-bit line stipple p, the line repeat count r, and an integer stipple counter s. Let

$$b = \lfloor s/r \rfloor \bmod 16,$$

Then a fragment is produced if the bth bit of p is 1, and not produced otherwise. The bits of p are numbered with 0 being the least significant and 15 being the most significant. The initial value of s is zero; s is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). s is reset to 0 whenever a **Begin** occurs, and before every line segment in a group of independent segments (as specified when **Begin** is invoked with `LINES`).

If the line segment has been clipped, then the value of s at the beginning of the line segment is indeterminate.

14.5.2.2 Wide Lines

The actual width of non-antialiased lines is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased line width. This implementation-dependent value must be no less than the implementation-dependent maximum antialiased line width,

Figure 14.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value was 1.

Non-antialiased line segments of width other than one are rasterized by offsetting them in the minor direction (for an x-major line, the minor direction is y, and for a y-major line, the minor direction is x) and producing a row or column of fragments in the minor direction (see figure 14.5). Let w be the width rounded to the nearest integer (if $w = 0$, then it is as if $w = 1$). If the line segment has endpoints given by (x_0, y_0) and (x_1, y_1) in window coordinates, the segment with endpoints $(x_0, y_0 - (w - 1)/2)$ and $(x_1, y_1 - (w - 1)/2)$ is rasterized, but instead of a single fragment, a column of fragments of height w (a row of fragments of length w for a y-major segment) is produced at each x (y for y-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates. The whole column is not produced if the stipple bit for the column's x location is zero; otherwise, the whole column is produced.

The preferred method of attribute interpolation for a wide line is to generate the same attribute values for all fragments in the row or column described above, as if the adjusted line were used for interpolation and those values replicated to

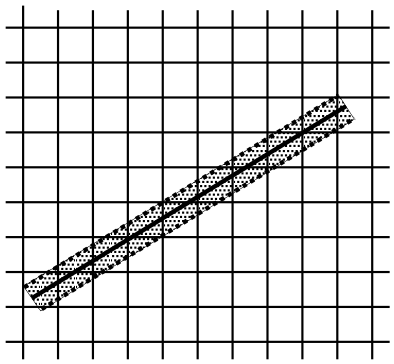


Figure 14.6. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).

the other fragments, except for `gl_FragCoord` which is interpolated as usual. An implementation may instead interpolate each fragment according to the formula in ¡¡primsrast-lines-basic,Basic Line Segment Rasterization¿¿, using the original line segment endpoints.

14.5.2.3 Antialiasing

Rasterized antialiased line segments produce fragments which intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment: one above the segment and one below it. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage values are computed for each fragment by computing the area of the intersection of the rectangle with the fragment square (see figure 14.6; see also section 14.3). Equation 14.6 is used to compute associated data values just as with non-antialiased lines; equation 14.5 is used to find the value of t for each fragment whose square is intersected by the line segment's rectangle. Not all widths need be supported for line segment antialiasing, but width 1.0 antialiased segments must be provided. As with the point width, a GL implementation may be queried for the range and number of gradations of available antialiased line widths.

For purposes of antialiasing, a stippled line is considered to be a sequence of contiguous rectangles centered on the line segment. Each rectangle has width equal

to the current line width and length equal to one pixel (except the last, which may be shorter). These rectangles are numbered from 0 to n, starting with the rectangle incident on the starting endpoint of the segment. Each of these rectangles is either eliminated or produced according to the procedure given under **Line Stipple**, above, where "fragment" is replaced with "rectangle." Each rectangle so produced is rasterized as if it were an antialiased polygon, described below (but culling, non-default settings of **PolygonMode**, and polygon stippling are not applied).

14.5.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width, a bit indicating whether line antialiasing is on or off, a 16-bit line stipple, the line stipple repeat count, and a bit indicating whether stippling is enabled or disabled. In addition, during rasterization an integer stipple counter must be maintained to implement line stippling. The initial value of the line width is 1.0. The initial state of line segment antialiasing is disabled. The initial value of the line stipple is $FFFF_{16}$ (a stipple of all ones). The initial value of the line stipple repeat count is one. The initial state of line stippling is disabled.

14.5.4 Line Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then lines are rasterized using the following algorithm, regardless of whether line antialiasing (`LINE_SMOOTH`) is enabled or disabled. Line rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect the rectangular region that is described in the **Antialiasing** portion of section 14.5.2 (Other Line Segment Features). If line stippling is enabled, the rectangular region is subdivided into adjacent unit-length rectangles, with some rectangles eliminated according to the procedure given in section 14.5.2.1, where "fragment" is replaced by "rectangle".

Coverage bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Each depth value and set of associated data is produced by substituting the corresponding sample location into equation 14.5, then using the result to evaluate equation 14.7. An implementation may choose to assign the associated data to more than one sample by evaluating equation 14.5 at any location within the pixel including the fragment center or any one of the sample locations, then substituting into equation 14.6. The different associated data values need not be evaluated at the same location.

Line width range and number of gradations are equivalent to those supported for antialiased lines.

14.6 Polygons

A polygon results from a triangle arising from a triangle strip, triangle fan, or series of separate triangles; a polygon primitive; or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or **Rect** command. Like points and line segments, polygon rasterization is controlled by several variables. Polygon antialiasing is enabled or disabled by calling **Enable** or **Disable** with *target* `POLYGON_SMOOTH`. The analog to line segment stippling for polygons is polygon stippling, described below.

14.6.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back-facing* or *front-facing*. This determination is made based on the sign of the (clipped or unclipped) polygon's area computed in window coordinates. One way to compute this area is

$$a = \frac{1}{2} f \sum_{i=0}^{n-1} x_w^i y_w^{i \oplus 1} - x_w^{i \oplus 1} y_w^i \tag{14.8}$$

where $f = 1$ when the clip control origin is `LOWER_LEFT` and $f = -1$ when the origin is `UPPER_LEFT`, x_w^i and y_w^i are the x and y window coordinates of the ith vertex of the n-vertex polygon (vertices are numbered starting at zero for purposes of this computation) and $i \oplus 1$ is $(i + 1) \bmod n$. The interpretation of the sign of this value is controlled with

`void` **FrontFace**(`enum` *dir*);

Setting *dir* to `CCW` (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) uses a as computed above. Setting *dir* to `CW` (corresponding to clockwise orientation) indicates that the sign of a should be reversed prior to use. Front face determination requires one bit of state, and is initially set to `CCW`.

Errors

An `INVALID_ENUM` error is generated if *dir* is not `CW` or `CCW`.

If the sign of a (including the possible reversal of this sign as determined by **FrontFace**) is positive, the polygon is front-facing; otherwise, it is back-facing. This determination is used in conjunction with the **CullFace** enable bit and mode

value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

`void` **CullFace**(`enum` *mode*);

mode must be `FRONT`, `BACK` or `FRONT_AND_BACK`. Culling is enabled or disabled by calling **Enable** or **Disable** with *target* `CULL_FACE`. Front-facing polygons are rasterized if either culling is disabled or the **CullFace** mode is `BACK` while back-facing polygons are rasterized only if either culling is disabled or the **CullFace** mode is `FRONT`. The initial setting of the **CullFace** mode is `BACK`. Initially, culling is disabled.

Errors

An `INVALID_ENUM` error is generated if *mode* is not `FRONT`, `BACK`, or `FRONT_AND_BACK`.

The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the x and y window coordinates of the polygon's vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon boundary edge. In such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization.

As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers, a, b, and c, each in the range $[0, 1]$, with $a + b + c = 1$. These coordinates uniquely specify any point p within the triangle or on the triangle's boundary as

$$p = ap_a + bp_b + cp_c,$$

where p_a, p_b, and p_c are the vertices of the triangle. a, b, and c can be found as

$$a = \frac{\mathrm{A}(pp_bp_c)}{\mathrm{A}(p_ap_bp_c)}, \quad b = \frac{\mathrm{A}(pp_ap_c)}{\mathrm{A}(p_ap_bp_c)}, \quad c = \frac{\mathrm{A}(pp_ap_b)}{\mathrm{A}(p_ap_bp_c)},$$

where $\mathrm{A}(lmn)$ denotes the area in window coordinates of the triangle with vertices l, m, and n.

Denote an associated datum at p_a, p_b, or p_c as f_a, f_b, or f_c, respectively. Then the value f of a datum at a fragment produced by rasterizing a triangle is given by

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \tag{14.9}$$

where w_a, w_b and w_c are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. a, b, and c must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment's center. However, depth values for polygons must be interpolated by

$$z = az_a + bz_b + cz_c \tag{14.10}$$

where z_a, z_b, and z_c are the depth values of p_a, p_b, and p_c, respectively.

When a vertex processing stage is active, the `noperspective` and `flat` keywords used to declare shader outputs affect how they are interpolated. When neither keyword is specified, interpolation is performed as described in equation 14.9. When the `noperspective` keyword is specified, interpolation is performed in the same fashion as for depth values, as described in equation 14.10. When the `flat` keyword is specified, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive (see section 13.6).

For a polygon with more than three edges, we require only that a convex combination of the values of the datum at the polygon's vertices can be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

$$f = \sum_{i=1}^{n} a_i f_i$$

where n is the number of vertices in the polygon, f_i is the value of the f at vertex i; for each i $0 \leq a_i \leq 1$ and $\sum_{i=1}^{n} a_i = 1$. The values of the a_i may differ from fragment to fragment, but at vertex i, $a_j = 0, j \neq i$ and $a_i = 1$.

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation 14.9 should be iterated independently and a division performed for each fragment).

14.6.2 Stippling

Polygon stippling works much the same way as line stippling, masking out certain fragments produced by rasterization so that they are not sent to the next stage of the GL. This is the case regardless of the state of polygon antialiasing. Stippling is controlled with

```
void PolygonStipple( const ubyte *pattern );
```

pattern is a pointer to memory into which a 32×32 pattern is packed. The pattern is unpacked from memory according to the procedure given in section 18.1 for **DrawPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were BITMAP, and the *format* were COLOR_INDEX. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones.

If x_w and y_w are the window coordinates of a rasterized polygon fragment, then that fragment is sent to the next stage of the GL if and only if the bit of the pattern $(x_w \bmod 32, y_w \bmod 32)$ is 1.

Polygon stippling may be enabled or disabled by calling **Enable** or **Disable** with *target* POLYGON_STIPPLE. When disabled, it is as if the stipple pattern were all ones.

14.6.2.1 Stipple Query

The commands

```
void GetPolygonStipple( ubyte *pattern );
void GetnPolygonStipple( sizei bufSize, void *pattern );
```

obtain the polygon stipple. The pattern is packed into pixel pack buffer or client memory according to the procedure given in section 18.2 for **ReadPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were BITMAP, and the *format* were COLOR_INDEX.

Errors

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetnPolygonStipple** if the buffer size required to store the requested data is greater than *bufSize*.

14.6.3 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage value is computed at each such fragment, and this value is saved to be applied as described in section 17.1. An associated datum is assigned to a fragment by integrating the datum's value over the region of the intersection of the fragment square with the polygon's interior and dividing this integrated value by the area of the intersection. For a fragment square lying entirely within the polygon, the value of a datum at the fragment's center may be used instead of integrating the value across the fragment.

Polygon stippling operates in the same way whether polygon antialiasing is enabled or not. The polygon point sampling rule defined in section 14.6.1, however, is not enforced for antialiased polygons.

14.6.4 Options Controlling Polygon Rasterization

The interpretation of polygons for rasterization is controlled using

`void` **PolygonMode**(`enum` *face*, `enum` *mode*);

face is one of `FRONT`, `BACK`, or `FRONT_AND_BACK`, indicating that the rasterizing method described by *mode* respectively replaces the rasterizing method for front-facing polygons, back-facing polygons, or both front- and back-facing polygons. *mode* must be one of `POINT`, `LINE`, or `FILL`. Calling **PolygonMode** with `POINT` causes certain vertices of a polygon to be treated, for rasterization purposes, as if they had been drawn with *mode* `POINTS`. The vertices selected for this treatment are those that have been tagged as having a polygon boundary edge beginning on them (see section 10.1.17). `LINE` causes edges that are tagged as boundary to be rasterized as line segments. (The line stipple counter is reset at the beginning of the first rasterized edge of the polygon, but not for subsequent edges.) `FILL` is the default mode of polygon rasterization, corresponding to the description in sections 14.6.1, 14.6.2, and 14.6.3. Note that these modes affect only the final rasterization of polygons: in particular, a polygon's vertices are lit, and the polygon is clipped and possibly culled before these modes are applied.

Polygon antialiasing applies only to the `FILL` state of **PolygonMode**. For `POINT` or `LINE`, point antialiasing or line segment antialiasing, respectively, apply.

14.6.5 Depth Offset

The depth values of all fragments generated by the rasterization of a polygon may be offset by a single value that is computed for that polygon. The function that

determines this value is specified with the commands

```
void PolygonOffsetClamp( float factor, float units, float
    clamp );
void PolygonOffset( float factor, float units );
```

factor scales the maximum depth slope of the polygon, and *units* scales an implementation-dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the polygon offset value, which may then be clamped to a minimum or maximum value specified by *clamp*. The values *factor*, *units*, and *clamp* may each be positive, negative, or zero. Calling the command **PolygonOffset** is equivalent to calling the command **PolygonOffsetClamp** with *clamp* equal to zero.

The maximum depth slope m of a triangle is

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{14.11}$$

where (x_w, y_w, z_w) is a point on the triangle. m may be approximated as

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}. \tag{14.12}$$

If the polygon has more than three vertices, one or more values of m may be used during rasterization. Each may take any value in the range $[min, max]$, where min and max are the smallest and largest values obtained by evaluating equation 14.11 or equation 14.12 for the triangles formed by all three-vertex combinations.

The minimum resolvable difference r is an implementation-dependent parameter that depends on the depth buffer representation. It is the smallest difference in window coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but z_w values that differ by r, will have distinct depth values.

For fixed-point depth buffer representations, r is constant throughout the range of the entire depth buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, e, in the range of z values spanned by the primitive. If n is the number of bits in the floating-point mantissa, the minimum resolvable difference, r, for the given primitive is defined as

$$r = 2^{e-n}.$$

If no depth buffer is present, r is undefined.

The offset value o for a polygon is

$$o = \begin{cases} m \times factor + r \times units, & clamp = 0 \text{ or } NaN \\ \min(m \times factor + r \times units, clamp), & clamp > 0 \\ \max(m \times factor + r \times units, clamp), & clamp < 0 \end{cases} \tag{14.13}$$

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range $[0, 1]$, and o is applied to depth values in the same range.

The state required for polygon offset consists of three boolean state values, POLYGON_OFFSET_POINT, POLYGON_OFFSET_LINE, and POLYGON_OFFSET_FILL, which determine whether polygon offset o is applied during the rasterization of polygons in POINT, LINE, and FILL modes, respectively, and a floating-point state value, POLYGON_OFFSET_CLAMP, specifying the offset clamp.

The initial value of each of the offset enables is FALSE, and the initial value of the offset clamp is 0.0.

If polygon offset point, line, or fill is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon in POINT, LINE, or FILL mode, respectively.

For fixed-point depth buffers, fragment depth values are always limited to the range $[0, 1]$ by clamping after offset addition is performed. Fragment depth values are clamped even when the depth buffer uses a floating-point representation.

14.6.6 Polygon Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (POLYGON_SMOOTH) is enabled or disabled. Polygon rasterization produces a fragment for each framebuffer pixel with one or more sample points that satisfy the point sampling criteria described in section 14.6.1, including the special treatment for sample points that lie on a polygon boundary edge. If a polygon is culled, based on its orientation and the **CullFace** mode, then no fragments are produced during rasterization. Fragments are culled by the polygon stipple just as they are for aliased and antialiased polygons.

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Each associated datum is produced as described in section 14.6.1, but using the corresponding sample location instead of the fragment center. An implementation may choose to assign the same associated

data values to more than one sample by barycentric evaluation using any location within the pixel including the fragment center or one of the sample locations. The color value and the set of texture coordinates need not be evaluated at the same location.

When a vertex processing stage is active, the `noperspective` and `flat` qualifiers affect how shader outputs are interpolated in the same fashion as described for basic polygon rasterization in section 14.6.1.

The rasterization described above applies only to the `FILL` state of **PolygonMode**. For `POINT` and `LINE`, the rasterizations described in sections 14.4.3 (Point Multisample Rasterization) and 14.5.4 (Line Multisample Rasterization) apply.

14.6.7 Polygon Rasterization State

The state required for polygon rasterization consists of a polygon stipple pattern, whether stippling is enabled or disabled, the current state of polygon antialiasing (enabled or disabled), the current values of the **PolygonMode** setting for each of front- and back-facing polygons, whether point, line, and fill mode polygon offsets are enabled or disabled, and the factor and bias values of the polygon offset equation. The initial stipple pattern is all ones; initially stippling is disabled. The initial setting of polygon antialiasing is disabled. The initial state for **PolygonMode** is `FILL` for both front- and back-facing polygons. The initial polygon offset factor and bias values are both 0; initially polygon offset is disabled for all modes.

14.7 Current Raster Position

The *current raster position* is used by commands that directly affect pixels in the framebuffer, including **Bitmap** (see section 14.8) and **DrawPixels** (see section 18.1). These commands generate fragments, but bypass vertex transformation and primitive assembly. The current raster position, however, shares some of the characteristics of a vertex.

The current raster position is set using one of the commands

```
void RasterPos{234}{sifd}( T coords );
void RasterPos{234}{sifd}v( const T *coords );
```

RasterPos4 takes four values indicating x, y, z, and w. **RasterPos3** (or **RasterPos2**) is analogous, but sets only x, y, and z with w implicitly set to 1 (or only x and y with z implicitly set to 0 and w implicitly set to 1).

The coordinates are treated as if they were specified in a **Vertex** command. If a vertex shader is active, this vertex shader is executed using the x, y, z, and w

coordinates as the object coordinates of the vertex. Otherwise, the x, y, z, and w coordinates are transformed by the current model-view and projection matrices. These coordinates, along with current values, are used to generate primary and secondary colors and texture coordinates just as is done for a vertex. The colors and texture coordinates so produced replace the colors and texture coordinates stored in the current raster position's associated data. If a vertex shader is active then the current raster distance is set to the value of the shader built-in output `gl_FogFragCoord`. Otherwise, if the value of the fog source (see section 16.4) is `FOG_COORD`, then the current raster distance is set to the value of the current fog coordinate. Otherwise, the current raster distance is set to the distance from the origin of the eye coordinate system to the vertex as transformed by only the current model-view matrix. This distance may be approximated as discussed in section 16.4.

If depth clamping (see section 13.7) is enabled, then raster position z_w is first clamped to the range $[\min(n, f), \max(n, f)]$, where n and f are the current near and far depth range values (see section 13.8.1).

Since vertex shaders may be executed when the raster position is set, any attributes not written by the shader will result in undefined state in the current raster position. Vertex shaders should write all output variables that would be used when rasterizing pixel primitives using the current raster position.

The transformed coordinates are passed to clipping as if they represented a point. If the "point" is not culled, then the projection to window coordinates is computed (section 13.8) and saved as the current raster position, and the valid bit is set. If the "point" is culled, the current raster position and its associated data become indeterminate and the valid bit is cleared. Figure 14.7 summarizes the behavior of the current raster position.

Alternately, the current raster position may be set by one of the **WindowPos** commands:

`void` **WindowPos**{**23**}{**sifd**}(`T` *coords*);
`void` **WindowPos**{**23**}{**sifd**}**v**(`const T` *coords*);

WindowPos3 takes three values indicating x, y and z, while **WindowPos2** takes two values indicating x and y with z implicitly set to 0. The current raster position, (x_w, y_w, z_w, w_c), is defined by:

$$x_w = x$$

$$y_w = y$$

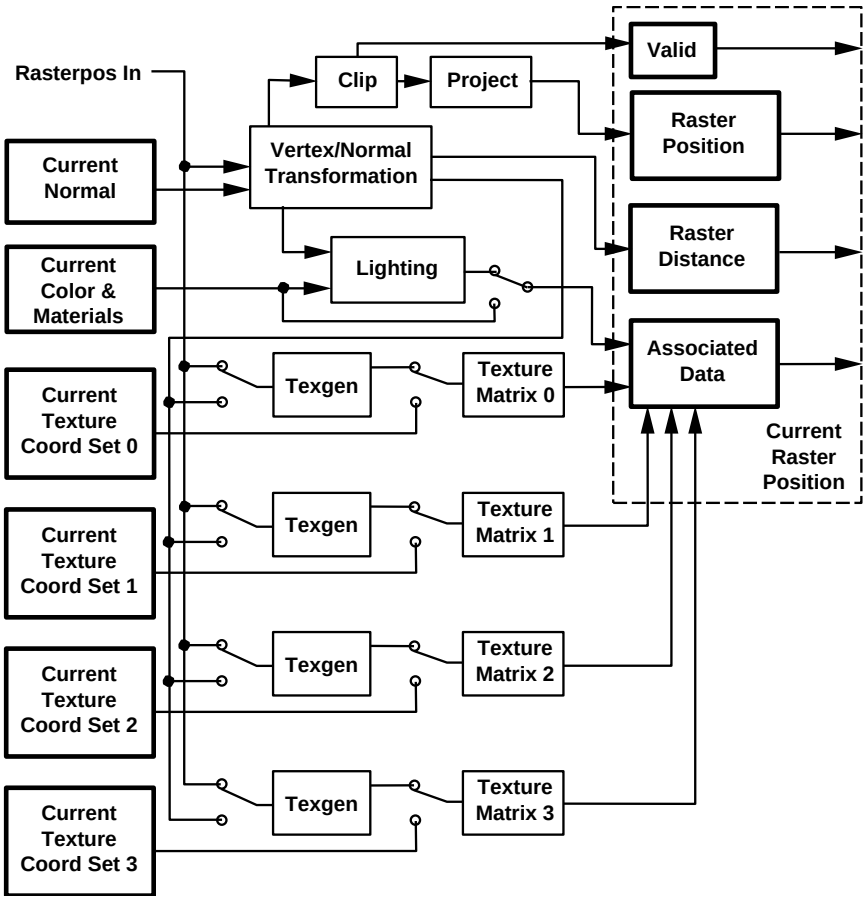


Figure 14.7. The current raster position and how it is set. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

$$z_w = \begin{cases} n, & z \leq 0 \\ f, & z \geq 1 \\ n + z(f - n), & otherwise \end{cases}$$

$$w_c = 1$$

where n and f are the values passed to **DepthRange** (see section 13.8.1).

Lighting, texture coordinate generation and transformation, and clipping are not performed by the **WindowPos** functions. Instead, in RGBA mode, the current raster color and secondary color are obtained from the current color and secondary color, respectively. If vertex color clamping is enabled, the current raster color and secondary color are clamped to $[0, 1]$. In color index mode, the current raster color index is set to the current color index. The current raster texture coordinates are set to the current texture coordinates, and the valid bit is set.

If the value of the fog source is `FOG_COORD_SRC`, then the current raster distance is set to the value of the current fog coordinate. Otherwise, the raster distance is set to 0.

The current raster position requires six single-precision floating-point values for its x_w, y_w, and z_w window coordinates, its w_c clip coordinate, its raster distance (used as the fog coordinate in raster processing), a single valid bit, four floating-point values to store the current RGBA color, four floating-point values to store the current RGBA secondary color, one floating-point value to store the current color index, and 4 floating-point values for texture coordinates for each texture unit. In the initial state, the coordinates and texture coordinates are all $(0, 0, 0, 1)$, the eye coordinate distance is 0, the fog coordinate is 0, the valid bit is set, the associated RGBA color is $(1, 1, 1, 1)$, the associated RGBA secondary color is $(0, 0, 0, 1)$, and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color and secondary color always maintain their initial values.

14.8 Bitmaps

Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. These data are those associated with the *current raster position*.

Bitmaps are sent using

```
void Bitmap( sizei w, sizei h, float xbo, float ybo,
   float xbi, float ybi, const ubyte *data );
```

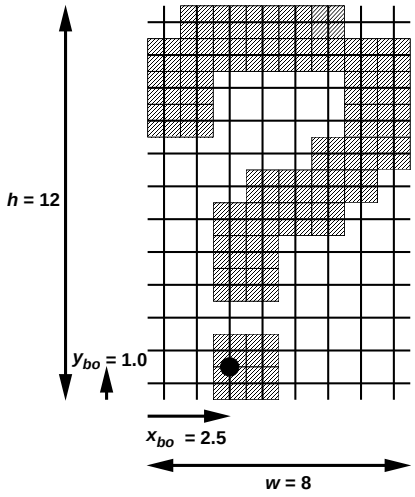


Figure 14.8. A bitmap and its associated parameters. x_{bi} and y_{bi} are not shown.

w and h comprise the integer width and height of the rectangular bitmap, respectively. (x_{bo}, y_{bo}) gives the floating-point x and y values of the bitmap's origin. (x_{bi}, y_{bi}) gives the floating-point x and y increments that are added to the raster position after the bitmap is rasterized. *data* is a pointer to a bitmap.

Like a polygon pattern, a bitmap is unpacked from memory according to the procedure given in section 18.1 for **DrawPixels**; it is as if the *width* and *height* passed to that command were equal to w and h, respectively, the *type* were `BITMAP`, and the *format* were `COLOR_INDEX`. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones. See figure 14.8.

A bitmap sent using **Bitmap** is rasterized as follows. First, if the current raster position is invalid (the valid bit is reset), the bitmap is ignored. Otherwise, a rectangular array of fragments is constructed, with lower left corner at

$$(x_{ll}, y_{ll}) = (\lfloor x_{rp} - x_{bo} \rfloor, \lfloor y_{rp} - y_{bo} \rfloor)$$

and upper right corner at $(x_{ll}+w, y_{ll}+h)$ where w and h are the width and height of the bitmap, respectively. Fragments in the array are produced if the corresponding bit in the bitmap is 1 and not produced otherwise. The associated data for each fragment are those associated with the current raster position. Once the fragments

have been produced, the current raster position is updated:

$$(x_{rp}, y_{rp}) \leftarrow (x_{rp} + x_{bi}, y_{rp} + y_{bi}).$$

The z and w values of the current raster position remain unchanged.

Errors

An INVALID_VALUE error is generated if *w* or *h* is negative.
An INVALID_FRAMEBUFFER_OPERATION error is generated if the object bound to DRAW_FRAMEBUFFER_BINDING is not framebuffer complete (see section 9.4.2).

14.8.0.1 Bitmap Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then bitmaps are rasterized using the following algorithm. If the current raster position is invalid, the bitmap is ignored. Otherwise, a screen-aligned array of pixel-size rectangles is constructed, with its lower left corner at (X_{rp}, Y_{rp}), and its upper right corner at $(X_{rp} + w, Y_{rp} + h)$, where w and h are the width and height of the bitmap. Rectangles in this array are eliminated if the corresponding bit in the bitmap is 0, and are retained otherwise. Bitmap rasterization produces a fragment for each framebuffer pixel with one or more sample points either inside or on the bottom or left edge of a retained rectangle.

Coverage bits that correspond to sample points either inside or on the bottom or left edge of a retained rectangle are 1, other coverage bits are 0. The associated data for each sample are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated exactly as it is in the single-sample rasterization case.

14.9 Early Per-Fragment Tests

Once fragments are produced by rasterization, a number of per-fragment operations are performed prior to fragment shader execution (see section 15). If a fragment is discarded during any of these operations, it will not be processed by any subsequent stage, including fragment shader execution.

Three fragment operations are performed, and a further three are optionally performed on each fragment, in the following order:

- the pixel ownership test (see section 14.9.1);

- the scissor test (see section 14.9.2);
- multisample fragment operations (see section 14.9.3);

If early per-fragment operations are enabled, these tests are also performed:

- the stencil test (see section 17.3.3);
- the depth buffer test (see section 17.3.4); and
- occlusion query sample counting (see section 17.3.5).

14.9.1 Pixel Ownership Test

The first test is to determine if the pixel at location (x_w, y_w) in the framebuffer is currently owned by the GL (more precisely, by this GL context). If it is not, the window system decides the fate of the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured.

If the draw framebuffer is a framebuffer object (see section 17.4.1), the pixel ownership test always passes, since the pixels of framebuffer objects are owned by the GL, not the window system. If the draw framebuffer is the default framebuffer, the window system controls pixel ownership.

14.9.2 Scissor Test

The scissor test determines if (x_w, y_w) lies within the scissor rectangle defined by four values for each viewport. These values are set with

```
void ScissorArrayv( uint first, sizei count, const
    int *v );
void ScissorIndexed( uint index, int left, int bottom,
    sizei width, sizei height );
void ScissorIndexedv( uint index, int *v );
void Scissor( int left, int bottom, sizei width,
    sizei height );
```

ScissorArrayv defines a set of scissor rectangles that are each applied to the corresponding viewport (see section 13.8.1). *first* specifies the index of the first scissor rectangle to modify, and *count* specifies the number of scissor rectangles. *v*

contains the address of an array of integers containing the left, bottom, width and height of the scissor rectangles, in that order.

If $left \leq x_w < left + width$ and $bottom \leq y_w < bottom + height$ for the selected scissor rectangle, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. For points, lines, and polygons, the scissor rectangle for a primitive is selected in the same manner as the viewport (see section 13.8.1). For bitmaps, pixel rectangles and buffer clears (see section 17.4.3), the scissor rectangle numbered zero is used for the scissor test.

Errors

An `INVALID_VALUE` error is generated by **ScissorArrayv** if $first + count$ is greater than the value of `MAX_VIEWPORTS`.

An `INVALID_VALUE` error is generated by **ScissorArrayv** if *count* is negative.

An `INVALID_VALUE` error is generated by **ScissorIndexed** and **Scissor** if *width* or *height* is negative.

The scissor test is enabled or disabled for all viewports using **Enable** or **Disable** with *target* `SCISSOR_TEST`. The test is enabled or disabled for a specific viewport using **Enablei** or **Disablei** with the constant `SCISSOR_TEST` and the index of the selected viewport. When disabled, it is as if the scissor test always passes. The value of the scissor test enable for viewport i can be queried by calling **IsEnabledi** with *target* `SCISSOR_TEST` and *index* i. The value of the scissor test enable for viewport zero may also be queried by calling **IsEnabled** with the same *target*, but no *index* parameter.

Errors

An `INVALID_VALUE` error is generated by **Enablei**, **Disablei** and **IsEnabledi** if *target* is `SCISSOR_TEST` and *index* is greater than or equal to the value of `MAX_VIEWPORTS`.

The state required consists of four integer values per viewport, and a bit indicating whether the test is enabled or disabled for each viewport. In the initial state, $left = bottom = 0$, and *width* and *height* are determined by the size of the window into which the GL is to do its rendering for all viewports. If the default framebuffer is bound but no default framebuffer is associated with the GL context (see chapter 9), then width and height are initially set to zero. Initially, the scissor test is disabled for all viewports.

ScissorIndexed and **ScissorIndexedv** specify the scissor rectangle for a single viewport and are equivalent (assuming no errors are generated) to:

```
int v[] = { left, bottom, width, height };
ScissorArrayv(index, 1, v);
```

and

```
ScissorArrayv(index, 1, v);
```

respectively.

Scissor sets the scissor rectangle for all viewports to the same values and is equivalent (assuming no errors are generated) to:

```
for (uint i = 0; i < MAX_VIEWPORTS; i++) {
    ScissorIndexed(i, left, bottom, width, height);
}
```

Calling **Enable** or **Disable** with *target* SCISSOR_TEST is equivalent, assuming no errors, to:

```
for (uint i = 0; i < MAX_VIEWPORTS; i++) {
    Enablei(SCISSOR_TEST, i);
    /* or */
    Disablei(SCISSOR_TEST, i);
}
```

14.9.3 Multisample Fragment Operations

This step modifies fragment coverage values based on the values of SAMPLE_COVERAGE, SAMPLE_COVERAGE_VALUE, SAMPLE_COVERAGE_INVERT, SAMPLE_MASK, SAMPLE_MASK_VALUE, and an output sample mask optionally written by the fragment shader. If MULTISAMPLE is disabled, or if the value of SAMPLE_BUFFERS is not one, this step is skipped.

All alpha values in this section refer only to the alpha component of the fragment shader output linked to color number zero, index zero (see section 15.2.3) if a fragment shader is in use, or the alpha component of the result of fixed-function fragment shading. If the fragment shader does not write to this output, the alpha value is undefined.

Sample coverage and sample mask operations are enabled or disabled by calling **Enable** and **Disable** with *target*s SAMPLE_COVERAGE or SAMPLE_MASK, respectively.

Next, if SAMPLE_COVERAGE is enabled, the fragment coverage is ANDed with a temporary coverage mask generated from the value of SAMPLE_COVERAGE_VALUE. If the value of SAMPLE_COVERAGE_INVERT is TRUE, this mask is inverted (all bit values are inverted) before it is ANDed with the fragment coverage. Finally, if SAMPLE_MASK is enabled, the fragment coverage is ANDed with the value of SAMPLE_MASK_VALUE. This updated coverage becomes the new fragment coverage value.

No specific algorithm is required for converting the sample coverage value to a temporary coverage mask. It is intended that the number of 1's in this value be proportional to the sample coverage value, with all 1's corresponding to a value of 1.0 and all 0's corresponding to 0.0. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

The values of SAMPLE_COVERAGE_VALUE and SAMPLE_COVERAGE_INVERT are specified by calling

```
void SampleCoverage( float value, boolean invert );
```

with *value* set to the desired coverage value, and *invert* set to TRUE or FALSE. *value* is clamped to $[0, 1]$ before being stored as SAMPLE_COVERAGE_VALUE. These values may be queried as described in table 23.16.

The value of SAMPLE_MASK_VALUE is specified using

```
void SampleMaski( uint maskNumber, bitfield mask );
```

with *mask* set to the desired mask for mask word *maskNumber*. Bit B of mask word M corresponds to sample $32 \times M + B$ as described in section 14.3.1. The sample mask value is queried by calling **GetIntegeri_v** with *target* set to SAMPLE_MASK_VALUE and the index set to *maskNumber*.

Errors

An INVALID_VALUE error is generated if *maskNumber* is greater than or equal to the value of MAX_SAMPLE_MASK_WORDS.

14.9.4 The Early Fragment Test Qualifier

The stencil test, depth buffer test and occlusion query sample counting are performed if and only if early fragment tests are enabled in the active fragment shader

(see section 15.2.4). When early per-fragment operations are enabled, these operations are performed prior to fragment shader execution, and the stencil buffer, depth buffer, and occlusion query sample counts will be updated accordingly; these operations will not be performed again after fragment shader execution.

When there is no active program, the active program has no fragment shader, or the active program was linked with early fragment tests disabled, these operations are performed only after fragment program execution, in the order described in section 17.3.

If early fragment tests are enabled, any depth value computed by the fragment shader has no effect. Additionally, the depth buffer, stencil buffer, and occlusion query sample counts may be updated even for fragments or samples that would be discarded after fragment shader execution due to per-fragment operations such as alpha or alpha-to-coverage tests.

Chapter 15

Programmable Fragment Processing

When the program object currently in use for the fragment stage (see section 7.3) includes a fragment shader, its shader is considered *active* and is used to process fragments resulting from rasterization (see section 14).

If the current fragment stage program object has no fragment shader, or no fragment program object is current for the fragment stage, the fixed-function fragment processing operations described in chapter 16 are used instead.

A fragment shader only applies when the GL is in RGBA mode. Its operation in color index mode is undefined.

Results of rasterization are undefined if any of the selected draw buffers of the draw framebuffer have an integer format and no fragment shader is active.

The processed fragments resulting from fragment shader execution are then further processed and written to the framebuffer as described in chapter 17.

15.1 Fragment Shader Variables

Fragment shaders can access uniforms belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

Fragment shaders also have access to samplers to perform texturing operations, as described in section 7.11.

Fragment shaders can read *input variables* or *inputs* that correspond to the attributes of the fragments produced by rasterization.

The OpenGL Shading Language Specification defines a set of built-in inputs that can be be accessed by a fragment shader. These built-in inputs include data

associated with a fragment that are used for fixed-function fragment processing, such as the fragment's color, secondary color, texture coordinates, fog coordinate, eye z coordinate, and position.

Additionally, the previous active shader stage (if fixed-function vertex processing is not being used) may define one or more output variables (see section 11.1.2.1 and the OpenGL Shading Language Specification). The values of these user-defined outputs are, if not flat shaded, interpolated across the primitive being rendered. The results of these interpolations are available when inputs of the same name are defined in the fragment shader.

User-defined inputs are not saved in the current raster position. When processing fragments generated by the rasterization of a pixel rectangle or bitmap, values of user-defined inputs are undefined. Built-in inputs have well-defined values.

When interpolating input variables, the default screen-space location at which these variables are sampled is defined in previous rasterization sections. The default location may be overriden by interpolation qualifiers. When interpolating variables declared using `centroid in`, the variable is sampled at a location within the pixel covered by the primitive generating the fragment. When interpolating variables declared using `sample in` when `MULTISAMPLE` is enabled, the fragment shader will be invoked separately for each covered sample and the variable will be sampled at the corresponding sample point.

Additionally, built-in fragment shader functions provide further fine-grained control over interpolation. The built-in functions `interpolateAtCentroid` and `interpolateAtSample` will sample variables as though they were declared with the `centroid` or `sample` qualifiers, respectively. The built-in function `interpolateAtOffset` will sample variables at a specified (x, y) offset relative to the center of the pixel. The range and granularity of offsets supported by this function is implementation-dependent. If either component of the specified offset is less than the value of `MIN_FRAGMENT_INTERPOLATION_OFFSET` or greater than the value of `MAX_FRAGMENT_INTERPOLATION_OFFSET`, the position used to interpolate the variable is undefined. Not all values of *offset* may be supported; x and y offsets may be rounded to fixed-point values with the number of fraction bits given by the value of the implementation-dependent constant `FRAGMENT_INTERPOLATION_OFFSET_BITS`.

A fragment shader can also write to output variables. Values written to these outputs are used in the subsequent per-fragment operations. Output variables can be used to write floating-point, integer or unsigned integer values destined for buffers attached to a framebuffer object, or destined for color buffers attached to the default framebuffer. Section 15.2.3 describes how to direct these values to buffers.

15.2 Shader Execution

If there is an active program object present for the fragment stage, the executable code for that program is used to process incoming fragments that are the result of rasterization, instead of the fixed-function method described in chapter 16.

Implementations are allowed to skip the execution of certain fragment shader invocations, and to execute additional fragment shader invocations during programmable fragment processing due to implementation dependent reasons, including the execution of fragment shader invocations when there is not an active program object present for the fragment shader stage, as long as the results of rendering otherwise remain unchanged.

Following shader execution, the fixed-function operations described in chapter 17 are performed.

Special considerations for fragment shader execution are described in the following sections.

15.2.1 Texture Access

Section 11.1.3.1 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value τ in the manner described in sections 8.14 and 8.15, and converts it to a texture base color C_b as shown in table 16.1 (section 16.1), followed by application of the texture swizzle as described in section 16.1 to compute the texture source color C_s and A_s.

The resulting four-component vector (R_s, G_s, B_s, A_s) is returned to the fragment shader. For the purposes of level-of-detail calculations, the derivatives $\frac{du}{dx}$, $\frac{du}{dy}$, $\frac{dv}{dx}$, $\frac{dv}{dy}$, $\frac{dw}{dx}$ and $\frac{dw}{dy}$ may be approximated by a differencing algorithm as described in section 8.13.1(“Derivative Functions”) of the OpenGL Shading Language Specification.

Texture lookups involving textures with depth and/or stencil component data are performed as described in section 11.1.3.5.

15.2.2 Shader Inputs

The OpenGL Shading Language Specification describes the values that are available as inputs to the fragment shader.

The built-in variable `gl_FragCoord` holds the fragment coordinate $\begin{pmatrix} x_f & y_f & z_f & w_f \end{pmatrix}$ for the fragment. Computing the fragment coordinate depends

on the fragment processing pixel-center and origin conventions (discussed below) as follows:

$$
\begin{aligned}
x_f &= \begin{cases} x_w - \frac{1}{2}, & \text{pixel-center convention is integer} \\ x_w, & \text{otherwise} \end{cases} \\
y'_f &= \begin{cases} H - y_w, & \text{origin convention is upper-left} \\ y_w, & \text{otherwise} \end{cases} \\
y_f &= \begin{cases} y'_f - \frac{1}{2}, & \text{pixel-center convention is integer} \\ y'_f & \text{otherwise} \end{cases} \\
z_f &= z_w \\
w_f &= \frac{1}{w_c}
\end{aligned}
\tag{15.1}
$$

where $\begin{pmatrix} x_w & y_w & z_w \end{pmatrix}$ is the fragment's window-space position, w_c is the w component of the fragment's clip-space position (see section 13.8), and H is the window's height in pixels. Note that z_w already has a polygon offset added in, if enabled (see section 14.6.5). z_f must be precisely zero or one in the case where z_w is either zero or one, respectively.

Unless otherwise specified by `layout` qualifiers in the fragment shader (see section 4.4.1.3("Fragment Shader Inputs") of the OpenGL Shading Language Specification), the fragment processing pixel-center convention is half-integer and the fragment processing origin convention is lower left.

The built-in variables `gl_Color` and `gl_SecondaryColor` hold the R, G, B, and A components, respectively, of the fragment color and secondary color. If the primary color or the secondary color components are represented by the GL as fixed-point values, they undergo an implied conversion to floating-point. This conversion must leave the values zero and one invariant. Floating-point color components (resulting from a disabled vertex color clamp) are unmodified.

The built-in variable `gl_FrontFacing` is set to `true` if the fragment is generated from a front-facing primitive, and `false` otherwise. For fragments generated from quadrilateral, polygon, or triangle primitives (including ones resulting from primitives rendered as points or lines), the determination is made by examining the sign of the area computed by equation 14.8 of section 14.6.1 (including the possible reversal of this sign controlled by **FrontFace**). If the sign is positive, fragments generated by the primitive are front-facing; otherwise, they are back-facing. All other fragments are considered front-facing.

If a geometry shader is active, the built-in variable `gl_PrimitiveID` contains the ID value emitted by the geometry shader for the provoking vertex. If no

geometry shader is active, gl_PrimitiveID contains the number of primitives processed by the rasterizer since the last drawing command was called. The first primitive generated by a drawing command is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn.

For QUADS and QUAD_STRIP primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed.

Restarting a primitive using the primitive restart index (see section 10.3) has no effect on the primitive ID counter.

The value of gl_PrimitiveID is undefined for fragments generated by POLYGON primitives or from **DrawPixels** or **Bitmap** commands. Additionally, gl_PrimitiveID is only defined under the same conditions that gl_VertexID is defined, as described under "Shader Inputs" in section 11.1.3.9.

The built-in read-only variable gl_SampleID is filled with the sample number of the sample currently being processed. This variable is in the range zero to gl_NumSamples minus one, where gl_NumSamples is the total number of samples in the framebuffer, or one if rendering to a non-multisample framebuffer. Using this variable in a fragment shader causes the entire shader to be evaluated per-sample. When rendering to a non-multisample buffer, or if multisample rasterization is disabled, gl_SampleID will always be zero. gl_NumSamples is the sample count of the framebuffer regardless of whether multisample rasterization is enabled or not.

The built-in read-only variable gl_SamplePosition contains the position of the current sample within the multi-sample draw buffer. The x and y components of gl_SamplePosition contain the sub-pixel coordinate of the current sample and will have values in the range $[0, 1]$. The sub-pixel coordinates of the center of the pixel are always $(0.5, 0.5)$. Using this variable in a fragment shader causes the entire shader to be evaluated per-sample. When rendering to a non-multisample buffer, or if multisample rasterization is disabled, gl_SamplePosition will always be $(0.5, 0.5)$.

The built-in variable gl_SampleMaskIn is an integer array holding bitfields indicating the set of fragment samples covered by the primitive corresponding to the fragment shader invocation. The number of elements in the array is

$$\left\lceil \frac{s}{32} \right\rceil,$$

where s is the value of the maximum number of color samples supported by the implementation for any renderable internal format. Bit n of element w in the array is set if and only if the sample numbered $32w + n$ is considered covered for this

fragment shader invocation. When rendering to a non-multisample buffer, or if multisample rasterization is disabled, all bits are zero except for bit zero of the first array element. That bit will be one if the pixel is covered and zero otherwise. Bits in the sample mask corresponding to covered samples that will be killed due to SAMPLE_COVERAGE or SAMPLE_MASK will not be set (see section 14.9.3). When per-sample shading is active due to the use of a fragment input qualified by sample or due to the use of the gl_SampleID or gl_SamplePosition variables, only the bit for the current sample is set in gl_SampleMaskIn. When state specifies multiple fragment shader invocations for a given fragment, the bit corresponding to each covered sample will be set in exactly one fragment shader invocation.

Similarly to the limit on geometry shader output components (see section 11.3.4.5), there is a limit on the number of components of built-in and user-defined input variables that can be read by the fragment shader, given by the value of the implementation-dependent constant MAX_FRAGMENT_INPUT_COMPONENTS.

When a program is linked, all components of any input variables read by a fragment shader will count against this limit. A program whose fragment shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for MAX_VERTEX_OUTPUT_COMPONENTS (see section 11.1.2.1).

15.2.3 Shader Outputs

The OpenGL Shading Language Specification describes the values that may be output by a fragment shader. These outputs are split into two categories, user-defined outputs and the built-in outputs gl_FragColor, gl_FragData[n], gl_FragDepth and gl_SampleMask.

If fragment color clamping is enabled and the color buffer has an unsigned normalized fixed-point, signed normalized fixed-point, or floating-point format, the final fragment color, fragment data, or output values written by a fragment shader are clamped to the range $[0, 1]$. Only user-defined outputs declared as a floating-point type are clamped and may be converted. If fragment color clamping is disabled, or the color buffer has an integer format, the final fragment color, fragment data, or output values are not modified. For fixed-point depth buffers, the final fragment depth written by a fragment shader is first clamped to $[0, 1]$ and then converted to fixed-point as if it were a window z value (see section 13.8.1). For floating-point depth buffers, conversion is not performed but clamping is. Note that the depth range computation is not applied here, only the conversion to fixed-point.

The built-in integer array gl_SampleMask can be used to change the sample

coverage for a fragment from within the shader. The number of elements in the array is

$$\left\lceil \frac{s}{32} \right\rceil,$$

where s is the value of the maximum number of color samples supported by the implementation for any renderable internal format. If bit n of element w in the array is set to zero, sample $32w+n$ should be considered uncovered for the purposes of additional sample fragment operations, as described in section 17.3.10, and the corresponding bits in the fragment coverage mask are set to zero. Modifying the sample mask in this way may exclude covered samples from being processed further at a per-fragment granularity. However, setting sample mask bits to one will never enable samples not covered by the original primitive. If the fragment shader is being evaluated at any frequency other than per-fragment, bits of the sample mask not corresponding to the current fragment shader invocation do not affect the fragment coverage mask. If a fragment shader does not statically assign a value to `gl_SampleMask`, the fragment coverage mask is not modified. If a value is not assigned to `gl_SampleMask` due to flow of control, the affected bits of the sample mask are undefined.

Color values written by a fragment shader may be floating-point, signed integer, or unsigned integer. If the color buffer has a signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in equations 2.4 or 2.3, respectively; otherwise no type conversion is applied. If the values written by the fragment shader do not match the format(s) of the corresponding color buffer(s), the result is undefined.

Writing to `gl_FragColor` specifies the fragment color (color number zero) that will be used by subsequent stages of the pipeline. Writing to `gl_FragData[n]` specifies the value of fragment color number n. Any colors, or color components, associated with a fragment that are not written by the fragment shader are undefined.

A fragment shader may not statically assign values to more than one of `gl_FragColor`, `gl_FragData[n]`, and any user-defined output variable. In this case, a compile or link error will result. A shader statically assigns a value to a variable if, after pre-processing, it contains a statement that would write to the variable, whether or not run-time flow of control will cause that statement to be executed.

Writing to `gl_FragDepth` specifies the depth value for the fragment being processed. If the active fragment shader does not statically assign a value to `gl_FragDepth`, then the depth value generated during rasterization is used by subsequent stages of the pipeline. Otherwise, the value assigned to `gl_FragDepth` is used, and is undefined for any fragments where statements assigning a value to

gl_FragDepth are not executed. Thus, if a shader statically assigns a value to gl_FragDepth, then it is responsible for always writing it.

The binding of a user-defined output variable to components of a fragment color number can be specified explicitly in the shader text or SPIR-V shader, or using the command

```
void BindFragDataLocationIndexed( uint program,
    uint colorNumber, uint index, const char * name );
```

specifies that the output variable *name* in *program* should be bound to fragment color *colorNumber* when the program is next linked. *index* may be zero or one to specify that the color will be used as either the first or second color input to the blend equation, respectively, as described in section 17.3.6.

If *name* was bound previously, its assigned binding is replaced with *colorNumber*. *name* must be a null-terminated string.

Errors

An INVALID_VALUE error is generated if *program* is not the name of either a program or shader object.

An INVALID_OPERATION error is generated if *program* is the name of a shader object.

An INVALID_VALUE error is generated if *index* is greater than one, if *colorNumber* is greater than or equal to the value of MAX_DRAW_BUFFERS and *index* is zero, or if *colorNumber* is greater than or equal to the value of MAX_DUAL_SOURCE_DRAW_BUFFERS and *index* is equal to one.

The command

```
void BindFragDataLocation( uint program,
    uint colorNumber, const char * name );
```

is equivalent to

```
BindFragDataLocationIndexed (program, colorNumber, 0, name);
```

BindFragDataLocation has no effect until the program is linked. In particular, it doesn't modify the bindings of outputs in a program that has already been linked.

Errors

An INVALID_OPERATION error is generated if *name* starts with the reserved gl_ prefix.

When a program is linked, each active user-defined fragment shader output variable will have a binding consisting of a fragment color number, a fragment color index, and a component index. Output variables declared with location, component, or index layout qualifiers will use the values specified in the shader text. For SPIR-V shaders, these are specified by the Location, Component, and Index decorations. Output variables without such layout or decoration qualifiers will use the bindings specified by **BindFragDataLocationIndexed** or **BindFragDataLocation**, if any. **BindFragDataLocation*** has no effect on SPIR-V shaders, since the locations must always be fully specified as described in section 15.2.3.1.

Otherwise, the linker will automatically assign a fragment color number, using any color number not already assigned to another active fragment shader output variable. The fragment color index and component index of an output variable binding will default to zero unless values are explicitly specified by a layout qualifer or **BindFragDataLocationIndexed**. The properties of an active fragment shader output variable binding can be queried using the command **GetProgramResourceiv** with a *programInterface* of PROGRAM_OUTPUT and *props* values of LOCATION, LOCATION_INDEX, and LOCATION_COMPONENT.

When a fragment shader terminates, the value of each active user-defined output variable is written to components of the fragment color output to which it is bound. The set of fragment color components written is determined according to the variable's data type and component index binding, using the mappings in table 11.1. For an output variable declared as an array bound to fragment color number i, individual active array elements are written to consecutive fragment color numbers beginning with i, with the components written determined from the array element's data type and the array variable's component index binding.

Output binding assignments will cause **LinkProgram** to fail:

- if the number of active outputs is greater than the value of MAX_DRAW_BUFFERS;
- if the program has an active output assigned to a location greater than or equal to the value of MAX_DUAL_SOURCE_DRAW_BUFFERS and has an active output assigned an index greater than or equal to one;
- if two output variables are bound to the same output number and index with overlapping components selected;

- if two output variables with different component types (signed integer, unsigned integer, or floating-point) are bound to the same output number, even if selected components do not overlap; or
- if the explicit binding assigments do not leave enough space for the linker to automatically assign a location for an output array, which requires multiple contiguous locations.

BindFragDataLocationIndexed may be issued before any shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with `gl_`) to a color number and index, including a name that is never used as an output in any fragment shader object. Assigned bindings for variables that do not exist are ignored.

To determine the set of fragment shader output attribute variables used by a program, applications can query the properties and active resources of the `PROGRAM_OUTPUT` interface of a program including a fragment shader.

Additionally, the commands

```
int GetFragDataLocation( uint program, const
    char *name );
int GetFragDataIndex( uint program, const char *name );
```

are provided to query the location and fragment color index assigned to a fragment shader output variable.

Errors

If *program* has been linked successfully but contains no fragment shader, no error is generated but -1 will be returned.

An `INVALID_OPERATION` error is generated and -1 is returned if *program* has not been linked successfully.

Otherwise, the commands are equivalent to

GetProgramResourceLocation(*program*, `PROGRAM_OUTPUT`, *name*);

and

GetProgramResourceLocationIndex(*program*, `PROGRAM_OUTPUT`, *name*);

respectively.

15.2.3.1 SPIR-V Fragment Output Interface

When a SPIR-V fragment stage is present, the variables listed by `OpEntryPoint` with the `Output` storage class form the fragment output interface. These variables must be decorated with a `Location` They can also be decorated with a `Component` and/or an `Index`.

User-defined fragment shader output variables are matched only by their `Location`, `Component`, and `Index` decorations. If two outputs are placed within the same location, they must have the same underlying type (floating-point or integer). No component aliasing of output variables is allowed. That is, there must not be two output variables which have the same location, component, and index, either explicitly declared or implied. Output values written by a fragment shader must be declared with either `OpTypeFloat` or `OpTypeInt`, and a `Width` of 32. Composites of these types are also permitted.

15.2.4 Early Fragment Tests

An explicit control is provided to allow fragment shaders to enable early fragment tests. If the fragment shader specifies the `early_fragment_tests` `layout` qualifier, the per-fragment tests described in section 14.9 will be performed prior to fragment shader execution. Otherwise, they will be performed after fragment shader execution.

15.3 Fragment Shader Queries

Fragment shader queries use query objects to track the number of fragment shader invocations.

When **BeginQuery** is called with a *target* of `FRAGMENT_SHADER_INVOCATIONS`, the fragment shader invocations count maintained by the GL is set to zero. When a fragment shader invocations query is active, the counter is incremented every time the fragment shader is invoked (see section 15.2).

The result of fragment shader queries may be implementation dependent due to reasons described in section 15.2.

Chapter 16

Fixed-Function Fragment Processing

When programmable fragment processing (see chapter 15) is not being performed, the fixed-function operations described in this chapter are performed instead. Fragments resulting from rasterization have textures applied (see section 16.2, followed by color sum (see section 16.3 and fog application (see section 16.4).

The fragments resulting from fixed-function fragment processing are then further processed and written written to the framebuffer as described in chapter 17.

16.1 Texture Environments and Texture Functions

The command

```
void TexEnv{if}( enum target, enum pname, T param );
void TexEnv{if}v( enum target, enum pname, const
    T *params );
```

sets parameters of the *exture environment* that specifies how texture values are interpreted when texturing a fragment, or sets per-texture-unit filtering parameters.

target must be one of `TEXTURE_FILTER_CONTROL`, `POINT_SPRITE`, or `TEXTURE_ENV`.

Data conversions are performed as specified in section 2.2.1.

pname is a symbolic constant indicating the parameter to be set. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form, *params* is a pointer to an array of parameters: either a single symbolic constant or a value or group of values to which the parameter should be set.

When *target* is TEXTURE_FILTER_CONTROL, *pname* must be TEXTURE_LOD_BIAS. In this case the parameter is a single signed floating-point value, $bias_{texunit}$, that biases the level-of-detail parameter λ as described in section 8.14.

When *target* is POINT_SPRITE, point sprite rasterization behavior is affected as described in section 14.4.

When *target* is TEXTURE_ENV, the possible environment parameters are TEXTURE_ENV_MODE, TEXTURE_ENV_COLOR, COMBINE_RGB, COMBINE_ALPHA, RGB_SCALE, ALPHA_SCALE, SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB, and OPERANDn_ALPHA, where n = 0, 1, or 2. TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, BLEND, ADD, or COMBINE. TEXTURE_ENV_COLOR is set to an RGBA color by providing four single-precision floating-point values. in the range $[0, 1]$ (values outside this range are clamped to it). If integers are provided for TEXTURE_ENV_COLOR, then they are converted to floating-point as described in equation 2.2.

The value of TEXTURE_ENV_MODE specifies a *texture function*. The result of this function depends on the fragment and the texture image value. The precise form of the function depends on the base internal formats of the texture images that were last specified.

C_f and A_f[1] are the primary color components of the incoming fragment. C_c and A_c are the components of the texture environment color. C_p and A_p are the components resulting from the previous texture environment (for texture environment 0, C_p and A_p are identical to C_f and A_f, respectively). C_v and A_v are the primary color components computed by the texture function. Finally, C_s and A_s are the components of the texture source color. They are derived by computing the base color C_b and A_b from the filtered texture values R_t, G_t, B_t, A_t, L_t, and I_t as shown in table 16.1, followed by *swizzling* the components of C_b, controlled by the values of the texture parameters TEXTURE_SWIZZLE_R, TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A. If the value of TEXTURE_SWIZZLE_R is denoted by $swizzle_r$, swizzling computes the first component of C_s according to

```
if (swizzle_r == RED)
   C_s[0] = C_b[0];
else if (swizzle_r == GREEN)
   C_s[0] = C_b[1];
else if (swizzle_r == BLUE)
```

[1] In the remainder of section 16.1, the notation C_x is used to denote each of the three components R_x, G_x, and B_x of a color specified by x. Operations on C_x are performed independently for each color component. The A component of colors is usually operated on in a different fashion, and is therefore denoted separately by A_x.

Texture Base Internal Format	Texture base color C_b	A_b
ALPHA	$(0, 0, 0)$	A_t
LUMINANCE	(L_t, L_t, L_t)	1
LUMINANCE_ALPHA	(L_t, L_t, L_t)	A_t
INTENSITY	(I_t, I_t, I_t)	I_t
RED	$(R_t, 0, 0)$	1
RG	$(R_t, G_t, 0)$	1
RGB	(R_t, G_t, B_t)	1
RGBA	(R_t, G_t, B_t)	A_t

Table 16.1: Correspondence of filtered texture components to texture base components.

```
    Cs[0] = Cb[2];
else if (swizzle_r == ALPHA)
    Cs[0] = Ab;
else if (swizzle_r == ZERO)
    Cs[0] = 0;
else if (swizzle_r == ONE)
    Cs[0] = 1; // float or int depending on texture component type
```

Swizzling of $C_s[1]$, $C_s[2]$, and A_s are similarly controlled by the values of TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A, respectively.

If fragment color clamping is enabled, all of these color values, including the results, are clamped to the range $[0, 1]$. If fragment color clamping is disabled, the values are not clamped. The texture functions are specified in tables 16.2, 16.3, and 16.4.

If the value of TEXTURE_ENV_MODE is COMBINE, the form of the texture function depends on the values of COMBINE_RGB and COMBINE_ALPHA, according to table 16.4. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE and ALPHA_SCALE, respectively. If fragment color clamping is enabled, the arguments and results used in table 16.4 are clamped to $[0, 1]$. Otherwise, the results are unmodified.

The arguments $Arg0$, $Arg1$, and $Arg2$ are determined by the values of SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB and OPERANDn_ALPHA, where $n = 0$, 1, or 2, as shown in tables 16.5 and 16.6. ${C_s}^n$ and ${A_s}^n$ denote the texture source color and alpha from the texture image bound to texture unit n

Texture Base Internal Format	REPLACE Function	MODULATE Function	DECAL Function
ALPHA	$C_v = C_p$ $A_v = A_s$	$C_v = C_p$ $A_v = A_p A_s$	*undefined*
LUMINANCE (or 1)	$C_v = C_s$ $A_v = A_p$	$C_v = C_p C_s$ $A_v = A_p$	*undefined*
LUMINANCE_ALPHA (or 2)	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	*undefined*
INTENSITY	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	*undefined*
RGB, RG, RED, or 3	$C_v = C_s$ $A_v = A_p$	$C_v = C_p C_s$ $A_v = A_p$	$C_v = C_s$ $A_v = A_p$
RGBA or 4	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	$C_v = C_p(1 - A_s) + C_s A_s$ $A_v = A_p$

Table 16.2: Texture functions REPLACE, MODULATE, and DECAL.

Texture Base Internal Format	BLEND Function	ADD Function
ALPHA	$C_v = C_p$ $A_v = A_p A_s$	$C_v = C_p$ $A_v = A_p A_s$
LUMINANCE (or 1)	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
LUMINANCE_ALPHA (or 2)	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p A_s$	$C_v = C_p + C_s$ $A_v = A_p A_s$
INTENSITY	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p(1 - A_s) + A_c A_s$	$C_v = C_p + C_s$ $A_v = A_p + A_s$
RGB, RG, RED, or 3	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
RGBA or 4	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p A_s$	$C_v = C_p + C_s$ $A_v = A_p A_s$

Table 16.3: Texture functions BLEND and ADD.

COMBINE_RGB	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 * Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$
INTERPOLATE	$Arg0 * Arg2 + Arg1 * (1 - Arg2)$
SUBTRACT	$Arg0 - Arg1$
DOT3_RGB	$4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$
DOT3_RGBA	$4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$

COMBINE_ALPHA	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 * Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$
INTERPOLATE	$Arg0 * Arg2 + Arg1 * (1 - Arg2)$
SUBTRACT	$Arg0 - Arg1$

Table 16.4: COMBINE texture functions. The scalar expression computed for the DOT3_RGB and DOT3_RGBA functions is placed into each of the 3 (RGB) or 4 (RGBA) components of the output. The result generated from COMBINE_ALPHA is ignored for DOT3_RGBA.

SRC*n*_RGB	OPERAND*n*_RGB	Argument
TEXTURE	SRC_COLOR	C_s
	ONE_MINUS_SRC_COLOR	$1 - C_s$
	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
TEXTURE*n*	SRC_COLOR	${C_s}^n$
	ONE_MINUS_SRC_COLOR	$1 - {C_s}^n$
	SRC_ALPHA	${A_s}^n$
	ONE_MINUS_SRC_ALPHA	$1 - {A_s}^n$
CONSTANT	SRC_COLOR	C_c
	ONE_MINUS_SRC_COLOR	$1 - C_c$
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	$1 - A_c$
PRIMARY_COLOR	SRC_COLOR	C_f
	ONE_MINUS_SRC_COLOR	$1 - C_f$
	SRC_ALPHA	A_f
	ONE_MINUS_SRC_ALPHA	$1 - A_f$
PREVIOUS	SRC_COLOR	C_p
	ONE_MINUS_SRC_COLOR	$1 - C_p$
	SRC_ALPHA	A_p
	ONE_MINUS_SRC_ALPHA	$1 - A_p$

Table 16.5: Arguments for COMBINE_RGB functions.

SRC*n*_ALPHA	OPERAND*n*_ALPHA	Argument
TEXTURE	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
TEXTURE*n*	SRC_ALPHA	${A_s}^n$
	ONE_MINUS_SRC_ALPHA	$1 - {A_s}^n$
CONSTANT	SRC_ALPHA	A_c
	ONE_MINUS_SRC_ALPHA	$1 - A_c$
PRIMARY_COLOR	SRC_ALPHA	A_f
	ONE_MINUS_SRC_ALPHA	$1 - A_f$
PREVIOUS	SRC_ALPHA	A_p
	ONE_MINUS_SRC_ALPHA	$1 - A_p$

Table 16.6: Arguments for COMBINE_ALPHA functions.

Errors

An INVALID_ENUM error is generated is *target* is not TEXTURE_FILTER_CONTROL, POINT_SPRITE, or TEXTURE_ENV.

An INVALID_ENUM error is generated if *pname* is not one of the valid parameter names described above.

An INVALID_OPERATION error is generated if *pname* is a valid parameter name, but is not a valid parameter for *target*, as described above.

An INVALID_OPERATION error is generated if *param* or *params* are not valid values for *pname*, as described above.

An INVALID_OPERATION error is generated if the active texture unit selected by **ActiveTexture** is greater than or equal to the value of MAX_TEXTURE_UNITS.

The state required for the current texture environment, for each texture unit, consists of a six-valued integer indicating the texture function, an eight-valued integer indicating the RGB combiner function and a six-valued integer indicating the ALPHA combiner function, six four-valued integers indicating the combiner RGB and ALPHA source arguments, three four-valued integers indicating the combiner RGB operands, three two-valued integers indicating the combiner ALPHA operands, and four floating-point environment color values. In the initial state, the texture and combiner functions are each MODULATE, the combiner RGB and ALPHA sources are each TEXTURE, PREVIOUS, and CONSTANT for sources 0, 1, and 2 respectively, the combiner RGB operands for sources zero and one are each SRC_COLOR, the combiner RGB operand for source 2, as well as for the combiner ALPHA operands, are each SRC_ALPHA, and the environment color is $(0, 0, 0, 0)$.

The state required for the texture filtering parameters, for each texture unit, consists of a single floating-point level-of-detail bias. The initial value of the bias is 0.0.

16.1.1 Texture Environment Queries

The command

`void` **GetTexEnv{if}v**(`enum` *target*, `enum` *pname*, `T` **params*);

returns information about *pname* for *target* in *params*. *env* must be either POINT_SPRITE, TEXTURE_ENV, or TEXTURE_FILTER_CONTROL.

Errors

The same errors generated by **TexEnv*** for invalid *target* and *pname* parameters are generated by **GetTexEnv*** with the corresponding parameters.

16.2 Texture Application

Texturing is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with *target* `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_RECTANGLE`, `TEXTURE_3D`, or `TEXTURE_CUBE_MAP` to enable the one-, two-, rectangular, three-dimensional, or cube map texture, respectively. If more than one of these textures is enabled, the first one enabled from the following list is used:

- cube map texture
- three-dimensional texture
- rectangular texture
- two-dimensional texture
- one-dimensional texture

Array and multisample textures are only supported by shaders, and may not be enabled for fixed-function texturing.

If all texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality using the rules given in sections 8.13 through 8.15. This texture value is used along with the incoming fragment in computing the texture function indicated by the currently bound texture environment. The result of this function replaces the incoming fragment's primary R, G, B, and A values. These are the color values passed to subsequent operations. Other data associated with the incoming fragment remain unchanged, except that the texture coordinates may be discarded.

Note that the texture value may contain R, G, B, A, L, I, or D components, but it does not contain an S component. If the texture's base internal format is `DEPTH_STENCIL`, for the purposes of texture application it is as if the base internal format were `DEPTH_COMPONENT`.

Each texture unit is enabled and bound to texture objects independently from the other texture units. Each texture unit follows the precedence rules for one-,

two-, three-dimensional, and cube map textures. Thus texture units can be performing texture mapping of different dimensionalities simultaneously. Each unit has its own enable and binding states.

Each texture unit is paired with an environment function, as shown in figure 16.1. The second texture function is computed using the texture value from the second texture, the fragment resulting from the first texture function computation and the second texture unit's environment function. If there is a third texture, the fragment resulting from the second texture function is combined with the third texture value using the third texture unit's environment function and so on. The texture unit selected by **ActiveTexture** determines which texture unit's environment is modified by **TexEnv** calls.

If the value of TEXTURE_ENV_MODE is COMBINE, the texture function associated with a given texture unit is computed using the values specified by SRC*n*_RGB, SRC*n*_ALPHA, OPERAND*n*_RGB and OPERAND*n*_ALPHA. If TEXTURE*n* is specified as SRC*n*_RGB or SRC*n*_ALPHA, the texture value from texture unit *n* will be used in computing the texture function for this texture unit.

Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit is passed unaltered to the following unit. Individual texture units beyond those specified by MAX_TEXTURE_UNITS are always treated as disabled.

If a texture unit is disabled or has an invalid or incomplete texture (as defined in section 8.17) bound to it, then blending is disabled for that texture unit. If the texture environment for a given enabled texture unit references a disabled texture unit, or an invalid or incomplete texture that is bound to another unit, then the results of texture blending are undefined.

Errors

An INVALID_OPERATION error is generated by **Enable** and **Disable** if the active texture unit selected by **ActiveTexture** is greater than or equal to the value of MAX_TEXTURE_UNITS.

The required state, per texture unit, is four bits indicating whether each of one-, two-, three-dimensional, or cube map texturing is enabled or disabled. In the initial state, all texturing is disabled for all texture units.

16.3 Color Sum

At the beginning of color sum, a fragment has two RGBA colors: a primary color $\mathbf{c}_{pri}$ (which texturing, if enabled, may have modified) and a secondary color $\mathbf{c}_{sec}$.

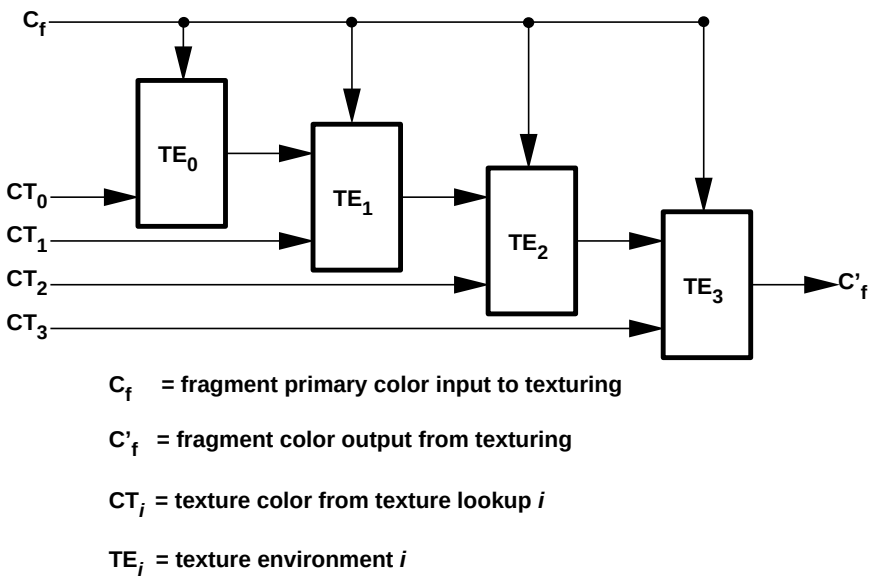


Figure 16.1. Multitexture pipeline. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. The input fragment color is successively combined with each texture according to the state of the corresponding texture environment, and the resulting fragment color passed as input to the next texture unit in the pipeline.

If color sum is enabled, the R, G, and B components of these two colors are summed to produce a single post-texturing RGBA color $\mathbf{c}$. The A component of $\mathbf{c}$ is taken from the A component of $\mathbf{c}_{pri}$; the A component of $\mathbf{c}_{sec}$ is unused. If color sum is disabled, then $\mathbf{c}_{pri}$ is assigned to $\mathbf{c}$. If fragment color clamping is enabled, the components of $\mathbf{c}$ are then clamped to the range $[0, 1]$.

Color sum is enabled or disabled by calling **Enable** or **Disable** with *target* `COLOR_SUM`. If lighting is enabled and no vertex processing stage is active, the color sum stage is always applied, ignoring the value of `COLOR_SUM`.

The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled.

Color sum has no effect in color index mode, or if a fragment shader is active.

16.4 Fog

If enabled, fog blends a fog color with a rasterized fragment's post-texturing color using a blending factor f. Fog is enabled and disabled by calling **Enable** and **Disable** commands with *target* `FOG`.

This factor f is computed according to one of three equations:

$$f = \exp(-d \cdot c), \tag{16.1}$$

$$f = \exp(-(d \cdot c)^2), \text{or} \tag{16.2}$$

$$f = \frac{e - c}{e - s} \tag{16.3}$$

If a vertex or geometry shader is active, or if the fog source, as defined below, is `FOG_COORD`, then c is the interpolated value of the fog coordinate for this fragment. Otherwise, if the fog source is `FRAGMENT_DEPTH`, then c is the eye-coordinate distance from the eye, $(0, 0, 0, 1)$ in eye coordinates, to the fragment center. The equation and the fog source, along with either d or e and s, is specified with

```
void Fog{if}( enum pname, T param );
void Fog{if}v( enum pname, const T *params );
```

If *pname* is `FOG_MODE`, then *param* must be, or *params* must point to an integer that is one of the symbolic constants `EXP`, `EXP2`, or `LINEAR`, in which case equation 16.1, 16.2, or 16.3, respectively, is selected for the fog calculation (if, when 16.3 is selected, $e = s$, results are undefined). If *pname* is `FOG_COORD_SRC`, then *param* must be, or *params* must point to an integer that is one of the symbolic

constants FRAGMENT_DEPTH or FOG_COORD. If *pname* is FOG_DENSITY, FOG_START, or FOG_END, then *param* is or *params* points to a value that is d, s, or e, respectively.

Errors

An INVALID_VALUE error is generated if d is negative.

Data conversions are performed as specified in section 2.2.1.

An implementation may choose to approximate the eye-coordinate distance from the eye to each fragment center by $|z_e|$.

No matter which equation and approximation is used to compute f, the result is clamped to $[0, 1]$ to obtain the final f.

f is used differently depending on whether the GL is in RGBA or color index mode. In RGBA mode, if C_r represents a rasterized fragment's R, G, or B value, then the corresponding value produced by fog is

$$C = fC_r + (1 - f)C_f.$$

(The rasterized fragment's A value is not changed by fog blending.) The R, G, B, and A values of C_f are specified by calling **Fog** with *pname* equal to FOG_COLOR; in this case *params* points to four values comprising C_f. If these are not floating-point values, then they are converted to floating-point as described in equation 2.2. If fragment color clamping is enabled, the components of C_r and C_f and the result C are clamped to the range $[0, 1]$ before the fog blend is performed.

In color index mode, the formula for fog blending is

$$I = i_r + (1 - f)i_f$$

where i_r is the rasterized fragment's color index and i_f is a single-precision floating-point value. $(1 - f)i_f$ is rounded to the nearest fixed-point value with the same number of bits to the right of the binary point as i_r, and the integer portion of I is masked (bitwise ANDed) with $2^n - 1$, where n is the number of bits in a color in the color index buffer (buffers are discussed in chapter 9). The value of i_f is set by calling **Fog** with *pname* set to FOG_INDEX and *param* being or *params* pointing to a single value for the fog index. The integer part of i_f is masked with $2^n - 1$.

The state required for fog consists of a three valued integer to select the fog equation, three floating-point values d, e, and s, an RGBA fog color and a fog color index, a two-valued integer to select the fog coordinate source, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled,

FOG_COORD_SRC is FRAGMENT_DEPTH, FOG_MODE is EXP, $d = 1.0$, $e = 1.0$, and $s = 0.0$; $C_f = (0, 0, 0, 0)$ and $i_f = 0$.

Fog has no effect if a fragment shader is active.

Chapter 17

Writing Fragments and Samples to the Framebuffer

After programmable or fixed-function fragment processing, the following fixed-function operations are applied to the resulting fragments:

- Antialiasing application (see section 17.1).
- Multisample point fade (see section 17.2).
- Per-fragment operations and writing to the framebuffer (see section 17.3).

Writing to the framebuffer is the final set of operations performed as a result of drawing primitives.

Additional commands controlling the framebuffer as a whole are described in section 17.4.

17.1 Antialiasing Application

If antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. In RGBA mode, the value is multiplied by the fragment's alpha value to yield a final alpha value. In color index mode, the value is used to set the low order bits of the color index value as described in section 14.3. The coverage value is applied separately to each fragment color, and only applied if the corresponding color buffer in the framebuffer has a fixed- or floating-point format.

17.2 Multisample Point Fade

If multisampling is enabled and the rasterized fragment results from a point primitive, then the computed fade factor from equation 14.2 is applied to the fragment. In RGBA mode, the fade factor is multiplied by the fragment's alpha value to yield a final alpha value. In color index mode, the fade factor has no effect. The fade factor is applied separately to each fragment color, and only applied if the corresponding color buffer in the framebuffer has a fixed- or floating-point format.

17.3 Per-Fragment Operations

A fragment is produced by rasterization with window coordinates of (x_w, y_w) and depth z, as described in chapter 14. The fragment is then modified by fixed-function or programmable fragment processing, which adds associated data as described in chapter 15. The fragment is then further modified, and possibly discarded by the per-fragment operations described in this chapter. These operations are diagrammed in figure 17.1, in the order in which they are performed. Finally, if the fragment was not discarded, it is used to update the framebuffer at the fragment's window coordinates.

The stencil test, depth test, and occlusion query operations described in sections 17.3.3, 17.3.4, and 17.3.5 may instead be performed prior to fragment processing, as described in section 14.9, if requested by the fragment program.

17.3.1 Alpha To Coverage

This step modifies fragment alpha and coverage values based on the values of `SAMPLE_ALPHA_TO_COVERAGE` and `SAMPLE_ALPHA_TO_ONE`. If `MULTISAMPLE` is disabled, if the value of `SAMPLE_BUFFERS` is not one, or if draw buffer zero is not `NONE` and the buffer it references has an integer format, these operations are skipped.

Alpha to coverage and alpha to one operations are enabled or disabled by calling **Enable** and **Disable** with *target*s `SAMPLE_ALPHA_TO_COVERAGE` or `SAMPLE_ALPHA_TO_ONE`, respectively.

All alpha values in this section refer only to the alpha component of the fragment shader output linked to color number zero, index zero (see section 15.2.3) if a fragment shader is in use, or the alpha component of the result of fixed-function fragment shading.

If `SAMPLE_ALPHA_TO_COVERAGE` is enabled, a temporary coverage value is generated where each bit is determined by the alpha value at the corresponding

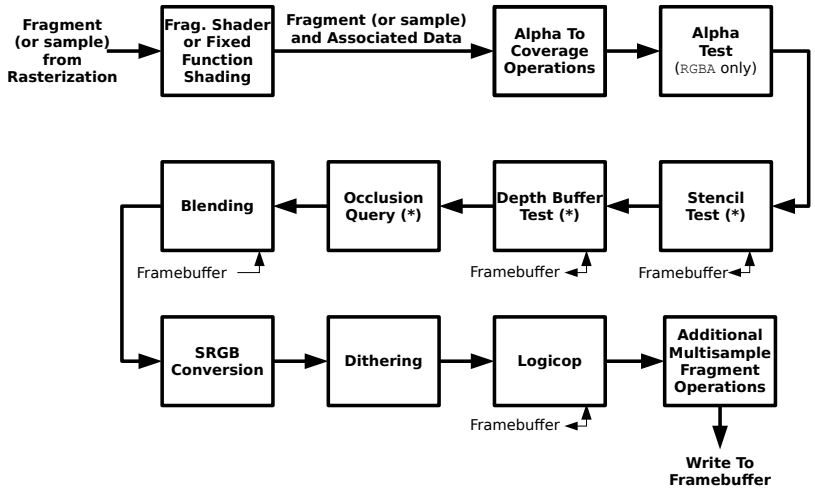


Figure 17.1. Per-fragment operations. The boxes labelled with "(*)" may instead be performed during early per-fragment operations, as described in section 14.9.

sample location (see section 14.3.1). The temporary coverage value is then ANDed with the fragment coverage value to generate a new fragment coverage value.

This temporary coverage is generated in the same manner as for sample coverage (see section 14.9.3), but as a function of the fragment's alpha value, clamped to the range $[0, 1]$. The function need not be identical, but it must have the same properties of proportionality and invariance.

Next, if SAMPLE_ALPHA_TO_ONE is enabled, each alpha value is replaced by the maximum representable alpha value for fixed-point color buffers, or by 1.0 for floating-point buffers. Otherwise, the alpha values are not changed.

17.3.2 Alpha Test

The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value. In color index mode, or if draw buffer zero references a buffer with an integer format, this step is skipped.

The alpha value in this section refers only to the alpha component of the fragment shader output linked to color number zero, index zero (see section 15.2.3) if a fragment shader is in use, or the alpha component of the result of fixed-function fragment shading. If the fragment shader does not write to this output, the alpha value is undefined.

The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using *target* ALPHA_TEST. When disabled, it is as if the comparison always passes. The test is controlled with

```
void AlphaFunc( enum func, float ref );
```

func is a symbolic constant indicating the alpha test function; *ref* is a reference value clamped to the range $[0, 1]$. When performing the alpha test, the GL will convert the reference value to the same representation as the fragment's alpha value (floating-point or fixed-point). For fixed-point, the reference value is converted according to equation 2.3 using the bit-width rule for an A component described in section 2.3.5, If draw buffer zero does not reference a buffer, the reference value is converted as if it were fixed-point with at least 8 bits of A. The fragment's alpha value is rounded to the nearest integer.

The possible constants specifying the test function are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL, meaning pass the fragment never, always, if the fragment's alpha value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the reference value, respectively.

The required state consists of the floating-point reference value, an eight-valued integer indicating the comparison function, and a bit indicating if the comparison is enabled or disabled. The initial state is for the reference value to be 0.0 and the function to be ALWAYS. Initially, the alpha test is disabled.

17.3.3 Stencil Test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location (x_w, y_w) and a reference value. The test is enabled or disabled with the **Enable** and **Disable** commands, using *target* STENCIL_TEST. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed.

The stencil test is controlled with

```
void StencilFunc( enum func, int ref, uint mask );
void StencilFuncSeparate( enum face, enum func, int ref,
    uint mask );
void StencilOp( enum sfail, enum dpfail, enum dppass );
void StencilOpSeparate( enum face, enum sfail, enum dpfail,
    enum dppass );
```

There are two sets of stencil-related state, the front stencil state set and the back stencil state set. Stencil tests and writes use the front set of stencil state when processing fragments rasterized from non-polygon primitives (points, lines, bitmaps, and image rectangles) and front-facing polygon primitives while the back set of stencil state is used when processing fragments rasterized from back-facing polygon primitives. For the purposes of stencil testing, a primitive is still considered a polygon even if the polygon is to be rasterized as points or lines due to the current polygon mode. Whether a polygon is front- or back-facing is determined in the same manner used for two-sided lighting (see section 12.2.1) and

StencilFuncSeparate and **StencilOpSeparate** take a *face* argument which can be FRONT, BACK, or FRONT_AND_BACK and indicates which set of state is affected. **StencilFunc** and **StencilOp** set front and back stencil state to identical values.

StencilFunc and **StencilFuncSeparate** take three arguments that control whether the stencil test passes or fails. *ref* is an integer reference value that is used in the unsigned stencil comparison. Stencil comparison operations and queries of *ref* clamp its value to the range $[0, 2^s - 1]$, where s is the number of bits in the stencil buffer attached to the draw framebuffer. The s least significant bits of *mask* are bitwise ANDed with both the reference and the stored stencil value, and the resulting masked values are those that participate in the comparison controlled by

func. *func* is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL. Accordingly, the stencil test passes never, always, and if the masked reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer.

StencilOp and **StencilOpSeparate** take three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. *sfail* indicates what action is taken if the stencil test fails. The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP, and DECR_WRAP. These correspond to keeping the current value, setting to zero, replacing with the reference value, incrementing with saturation, decrementing with saturation, bitwise inverting it, incrementing without saturation, and decrementing without saturation.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer. Incrementing or decrementing with saturation clamps the stencil value at 0 and the maximum representable value. Incrementing or decrementing without saturation will wrap such that incrementing the maximum representable value results in 0, and decrementing 0 results in the maximum representable value.

The same symbolic values are given to indicate the stencil action if the depth buffer test (see section 17.3.4) fails (*dpfail*), or if it passes (*dppass*).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to **StencilFunc** or **StencilFuncSeparate** and to **StencilOp** or **StencilOpSeparate**, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the front and back stencil reference values are both zero, the front and back stencil comparison functions are both ALWAYS, and the front and back stencil masks are both set to the value $2^s - 1$, where s is greater than or equal to the number of bits in the deepest stencil buffer supported by the GL implementation. Initially, all three front and back stencil operations are KEEP.

If there is no stencil buffer, no stencil modification can occur, and it is as if the stencil tests always pass, regardless of any calls to **StencilFunc**.

17.3.4 Depth Buffer Test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using *target* DEPTH_TEST. When disabled, the depth comparison and subsequent possible updates to the depth buffer value are bypassed and the fragment is passed to the next operation. The stencil value, however, is modified as indicated

below as if the depth buffer test passed. If enabled, the comparison takes place and the depth buffer and stencil value may subsequently be modified.

The comparison is specified with

`void` **DepthFunc**(`enum` *func*);

This command takes a single symbolic constant: one of `NEVER`, `ALWAYS`, `LESS`, `LEQUAL`, `EQUAL`, `GREATER`, `GEQUAL`, `NOTEQUAL`. Accordingly, the depth buffer test passes never, always, if the incoming fragment's z_w value is less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the depth value stored at the location given by the incoming fragment's (x_w, y_w) coordinates.

If depth clamping (see section 13.7) is enabled, before the incoming fragment's z_w is compared z_w is clamped to the range $[\min(n, f), \max(n, f)]$, where n and f are the current near and far depth range values (see section 13.8.1)

If the depth buffer test fails, the incoming fragment is discarded. The stencil value at the fragment's (x_w, y_w) coordinates is updated according to the function currently in effect for depth buffer test failure. Otherwise, the fragment continues to the next operation and the value of the depth buffer at the fragment's (x_w, y_w) location is set to the fragment's z_w value. In this case the stencil value is updated according to the function currently in effect for depth buffer test success.

The necessary state is an eight-valued integer and a single bit indicating whether depth buffering is enabled or disabled. In the initial state the function is `LESS` and the test is disabled.

If there is no depth buffer, it is as if the depth buffer test always passes.

17.3.5 Occlusion Queries

Occlusion queries use query objects to track the number of fragments that are not discarded by earlier stages. An occlusion query can be started and finished by calling **BeginQuery** and **EndQuery**, respectively, with a *target* of `SAMPLES_PASSED`, `ANY_SAMPLES_PASSED`, or `ANY_SAMPLES_PASSED_CONSERVATIVE`.

When an occlusion query is started with *target* `SAMPLES_PASSED`, the samples-passed count maintained by the GL is set to zero. When an occlusion query is active, the samples-passed count is incremented for each fragment still being processed after the depth test. If the value of `SAMPLE_BUFFERS` is zero, then the samples-passed count is incremented by one for each fragment. If the value of `SAMPLE_BUFFERS` is one, then the samples-passed count is incremented by the number of samples whose coverage bit is set. However, implementations, at their discretion, may instead increase the samples-passed count by the value of `SAMPLES` if any sample in the fragment is covered.

When an occlusion query finishes and all fragments generated by commands issued prior to **EndQuery** have been generated, the samples-passed count is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

When an occlusion query is started with the target `ANY_SAMPLES_PASSED`, the samples-boolean state maintained by the GL is set to `FALSE`. While that occlusion query is active, the samples-boolean state is set to `TRUE` for each fragment still being processed after the depth test. When the target is `ANY_SAMPLES_PASSED_CONSERVATIVE`, an implementation may choose to use a less precise version of the test which can additionally set the samples-boolean state to `TRUE` in some other implementation-dependent cases. This may offer better performance on some implementations at the expense of false positives. When the occlusion query finishes, the samples-boolean state of `FALSE` or `TRUE` is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

17.3.6 Blending

Blending combines the incoming *source* fragment's R, G, B, and A values with the *destination* R, G, B, and A values stored in the framebuffer at the fragment's (x_w, y_w) location.

Source and destination values are combined according to the *blend equation*, quadruplets of source and destination weighting factors determined by the *blend functions*, and a constant *blend color* to obtain a new set of R, G, B, and A values, as described below.

If the color buffer is fixed-point, the components of the source and destination values and blend factors are each clamped to $[0, 1]$ or $[-1, 1]$ respectively for an unsigned normalized or signed normalized color buffer prior to evaluating the blend equation. If the color buffer is floating-point, no clamping occurs. The resulting four values are sent to the next operation.

Blending applies only in RGBA mode; and only if the color buffer has a fixed-point or floating-point format. In color index mode, or if the color buffer has an integer format, proceed to the next operation.

Blending is enabled or disabled for an individual draw buffer with the commands

`void` **Enablei**(`enum` *target*, `uint` *index*);
`void` **Disablei**(`enum` *target*, `uint` *index*);

target is the symbolic constant `BLEND` and *index* is an integer i specifying the draw buffer associated with the symbolic constant `DRAW_BUFFER`i. If the color buffer

associated with DRAW_BUFFERi is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK (specifying multiple color buffers), then the state enabled or disabled is applicable for all of the buffers. Blending can be enabled or disabled for all draw buffers using **Enable** or **Disable** with *target* BLEND. If blending is disabled for a particular draw buffer, or if logical operation on color values is enabled (section 17.3.9), proceed to the next operation.

If multiple fragment colors are being written to multiple buffers (see section 17.4.1), blending is computed and applied separately for each fragment color and the corresponding buffer.

Errors

An INVALID_VALUE error is generated by **Enablei**, **Disablei** and **IsEnabledi** if *target* is BLEND and *index* is greater than or equal to the value of MAX_DRAW_BUFFERS.

17.3.6.1 Blend Equation

Blending is controlled by the blend equation. This equation can be simultaneously set to the same value for all draw buffers using the commands

```
void BlendEquation( enum mode );
void BlendEquationSeparate( enum modeRGB,
    enum modeAlpha );
```

or for an individual draw buffer using the indexed commands

```
void BlendEquationi( uint buf, enum mode );
void BlendEquationSeparatei( uint buf, enum modeRGB,
    enum modeAlpha );
```

BlendEquationSeparate and **BlendEquationSeparatei** argument *modeRGB* determines the RGB blend equation while *modeAlpha* determines the alpha blend equation. **BlendEquation** and **BlendEquationi** argument *mode* determines both the RGB and alpha blend equations. *mode*, *modeRGB*, and *modeAlpha* must be one of the blend equation modes in table 17.1. **BlendEquation** and **BlendEquationSeparate** modify the blend equations for all draw buffers. **BlendEquationi** and **BlendEquationSeparatei** modify the blend equations associated with an individual draw buffer. The *buf* argument is an integer i that indicates that the blend equations should be modified for DRAW_BUFFERi.

Errors

An INVALID_VALUE error is generated if *buf* is not in the range zero to the value of MAX_DRAW_BUFFERS minus one.

An INVALID_ENUM error is generated if any of *mode*, *modeRGB*, or *modeAlpha* are not one of the blend equation modes in table 17.1.

Signed or unsigned normalized fixed-point destination (framebuffer) components are represented as described in section 2.3.5. Constant color components, floating-point destination components, and source (fragment) components are taken to be floating-point values. If source components are represented internally by the GL as fixed-point values, they are also interpreted according to section 2.3.5.

Prior to blending, signed and unsigned normalized fixed-point color components undergo an implied conversion to floating-point using equations 2.2 and 2.1, respectively. This conversion must leave the values zero and one invariant. Blending computations are treated as if carried out in floating-point, and will be performed with a precision and dynamic range no lower than that used to represent destination components.

If FRAMEBUFFER_SRGB is enabled and the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the destination buffer is SRGB (see section 9.2.3), the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components in section 8.24.

If FRAMEBUFFER_SRGB is disabled or the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING is not SRGB, no linearization is performed.

The resulting linearized R, G, and B and unmodified A values are recombined as the destination color used in blending computations.

Table 17.1 provides the corresponding per-component blend equations for each mode, whether acting on RGB components for *modeRGB* or the alpha component for *modeAlpha*.

In the table, the s subscript on a color component abbreviation (R, G, B, or A) refers to the source color component for an incoming fragment, the d subscript on a color component abbreviation refers to the destination color component at the corresponding framebuffer location, and the c subscript on a color component abbreviation refers to the constant blend color component. A color component abbreviation without a subscript refers to the new color component resulting from blending. Additionally, S_r, S_g, S_b, and S_a are the red, green, blue, and alpha com-

Mode	RGB Components	Alpha Component
FUNC_ADD	$R = R_s * S_r + R_d * D_r$ $G = G_s * S_g + G_d * D_g$ $B = B_s * S_b + B_d * D_b$	$A = A_s * S_a + A_d * D_a$
FUNC_SUBTRACT	$R = R_s * S_r - R_d * D_r$ $G = G_s * S_g - G_d * D_g$ $B = B_s * S_b - B_d * D_b$	$A = A_s * S_a - A_d * D_a$
FUNC_REVERSE_SUBTRACT	$R = R_d * D_r - R_s * S_r$ $G = G_d * D_g - G_s * S_g$ $B = B_d * D_b - B_s * S_b$	$A = A_d * D_a - A_s * S_a$
MIN	$R = \min(R_s, R_d)$ $G = \min(G_s, G_d)$ $B = \min(B_s, B_d)$	$A = \min(A_s, A_d)$
MAX	$R = \max(R_s, R_d)$ $G = \max(G_s, G_d)$ $B = \max(B_s, B_d)$	$A = \max(A_s, A_d)$

Table 17.1: RGB and alpha blend equations.

ponents of the source weighting factors determined by the source blend function, and D_r, D_g, D_b, and D_a are the red, green, blue, and alpha components of the destination weighting factors determined by the destination blend function. Blend functions are described below.

17.3.6.2 Blend Functions

The weighting factors used by the blend equation are determined by the blend functions. There are four possible sources for weighting factors. These are the constant color (R_c, G_c, B_c, A_c) set with **BlendColor** (see below), the first source color $(R_{s0}, G_{s0}, B_{s0}, A_{s0})$, the second source color $(R_{s1}, G_{s1}, B_{s1}, A_{s1})$, and the destination color (the existing content of the draw buffer). Additionally the special constants ZERO and ONE are available as weighting factors.

Blend functions are simultaneously specified for all draw buffers using the commands

```
void BlendFunc( enum src, enum dst );
void BlendFuncSeparate( enum srcRGB, enum dstRGB,
    enum srcAlpha, enum dstAlpha );
```

or for an individual draw buffer using the indexed commands

```
void BlendFunci( uint buf, enum src, enum dst );
void BlendFuncSeparatei( uint buf, enum srcRGB,
    enum dstRGB, enum srcAlpha, enum dstAlpha );
```

BlendFuncSeparate and **BlendFuncSeparatei** arguments *srcRGB* and *dstRGB* determine the source and destination RGB blend functions, respectively, while *srcAlpha* and *dstAlpha* determine the source and destination alpha blend functions. **BlendFunc** and **BlendFunci** argument *src* determines both RGB and alpha source functions, while *dst* determines both RGB and alpha destination functions. **BlendFuncSeparate** and **BlendFunc** modify the blend functions for all draw buffers. **BlendFuncSeparatei** and **BlendFunci** modify the blend functions associated with an individual draw buffer. The *buf* argument is an integer i that indicates that the blend functions should be modified for DRAW_BUFFERi.

The possible source and destination blend functions and their corresponding computed blend factors are summarized in table 17.2.

Errors

An INVALID_VALUE error is generated if *buf* is not in the range zero to the value of MAX_DRAW_BUFFERS minus one.

An INVALID_ENUM error is generated if any of *src*, *dst*, *srcRGB*, *dstRGB*, *srcAlpha*, or *dstAlpha* are not one of the blend functions in table 17.2.

17.3.6.3 Dual Source Blending and Multiple Draw Buffers

Blend functions that require the second color input, $(R_{s1}, G_{s1}, B_{s1}, A_{s1})$ (SRC1_COLOR, SRC1_ALPHA, ONE_MINUS_SRC1_COLOR, or ONE_MINUS_SRC1_ALPHA) may consume hardware resources that could otherwise be used for rendering to multiple draw buffers. Therefore, the number of draw buffers that can be attached to a framebuffer may be lower when using dual-source blending.

The maximum number of draw buffers that may be attached to a single framebuffer when using dual-source blending functions is implementation-dependent and may be queried by calling **GetIntegerv** with *pname* MAX_DUAL_SOURCE_DRAW_BUFFERS. When using dual-source blending, MAX_DUAL_SOURCE_DRAW_BUFFERS should be used in place of MAX_DRAW_BUFFERS to determine the maximum number of draw buffers that may be attached to a single framebuffer. The value of MAX_DUAL_SOURCE_DRAW_BUFFERS must be at least 1. If the value of MAX_DUAL_SOURCE_DRAW_BUFFERS is 1, then dual-source blending and multiple draw buffers cannot be used simultaneously.

Function	RGB Blend Factors (S_r, S_g, S_b) or (D_r, D_g, D_b)	Alpha Blend Factor S_a or D_a
ZERO	$(0, 0, 0)$	0
ONE	$(1, 1, 1)$	1
SRC_COLOR	(R_{s0}, G_{s0}, B_{s0})	A_{s0}
ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_{s0}, G_{s0}, B_{s0})$	$1 - A_{s0}$
DST_COLOR	(R_d, G_d, B_d)	A_d
ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
SRC_ALPHA	(A_{s0}, A_{s0}, A_{s0})	A_{s0}
ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_{s0}, A_{s0}, A_{s0})$	$1 - A_{s0}$
DST_ALPHA	(A_d, A_d, A_d)	A_d
ONE_MINUS_DST_ALPHA	$(1, 1, 1) - (A_d, A_d, A_d)$	$1 - A_d$
CONSTANT_COLOR	(R_c, G_c, B_c)	A_c
ONE_MINUS_CONSTANT_COLOR	$(1, 1, 1) - (R_c, G_c, B_c)$	$1 - A_c$
CONSTANT_ALPHA	(A_c, A_c, A_c)	A_c
ONE_MINUS_CONSTANT_ALPHA	$(1, 1, 1) - (A_c, A_c, A_c)$	$1 - A_c$
SRC_ALPHA_SATURATE	(f, f, f)[1]	1
SRC1_COLOR	(R_{s1}, G_{s1}, B_{s1})	A_{s1}
ONE_MINUS_SRC1_COLOR	$(1, 1, 1) - (R_{s1}, G_{s1}, B_{s1})$	$1 - A_{s1}$
SRC1_ALPHA	(A_{s1}, A_{s1}, A_{s1})	A_{s1}
ONE_MINUS_SRC1_ALPHA	$(1, 1, 1) - (A_{s1}, A_{s1}, A_{s1})$	$1 - A_{s1}$

Table 17.2: RGB and ALPHA source and destination blending functions and the corresponding blend factors. Addition and subtraction of triplets is performed component-wise.
[1] $f = \min(A_{s0}, 1 - A_d)$.

Errors

An INVALID_OPERATION error is generated by any command that transfers vertices to the GL if either blend function requires the second color input for any draw buffer, and any draw buffers greater than or equal to the value of MAX_DUAL_SOURCE_DRAW_BUFFERS have values other than NONE.

17.3.6.4 Generation of Second Color Source for Blending

There is no way to generate the second source color using the fixed-function fragment pipeline. Rendering using any of the blend functions that consume the second input color (SRC1_COLOR, ONE_MINUS_SRC1_COLOR, SRC1_ALPHA or ONE_MINUS_SRC1_ALPHA) using fixed function will produce undefined results. To produce input for the second source color, a shader must be used.

When using a fragment shader with dual-source blending functions, the color outputs are bound to the first and second inputs of the blender using **BindFragDataLocationIndexed** as described in section 15.2.3. Data written to the first of these outputs becomes the first source color input to the blender (corresponding to SRC_COLOR and SRC_ALPHA). Data written to the second of these outputs generates the second source color input to the blender (corresponding to SRC1_COLOR and SRC1_ALPHA).

If the second color input to the blender is not written in the shader, or if no output is bound to the second input of a blender, the result of the blending operation is not defined.

17.3.6.5 Blend Color

The constant color C_c to be used in blending is specified with the command

```
void BlendColor( float red, float green, float blue,
    float alpha );
```

The constant color can be used in both the source and destination blending functions. If destination framebuffer components use an unsigned normalized fixed-point representation, the constant color components are clamped to the range $[0, 1]$ when computing blend factors.

17.3.6.6 Blending State

The state required for blending, for each draw buffer, is two integers for the RGB and alpha blend equations, four integers indicating the source and destination RGB

and alpha blending functions, and a bit indicating whether blending is enabled or disabled. Additionally, four floating-point values to store the RGBA constant blend color are required.

For all draw buffers, the initial blend equations for RGB and alpha are both FUNC_ADD, and the initial blending functions are ONE for the source RGB and alpha functions and ZERO for the destination RGB and alpha functions. Initially, blending is disabled for all draw buffers. The initial constant blend color is $(\mathrm{R}, \mathrm{G}, \mathrm{B}, \mathrm{A}) = (0, 0, 0, 0)$.

The value of the blend enable for draw buffer i may be queried by calling **IsEnabledi** with *target* BLEND and *index* i, and the values of the blend equations and functions may be queried by calling **GetIntegeri_v** with the corresponding *target* as shown in table 23.27 and *index* i.

The value of the blend enable, or the blend equations and functions for draw buffer zero may also be queried by calling **IsEnabled** or **GetIntegerv** respectively, with the same *target* but no *index* parameter.

Blending occurs once for each color buffer currently enabled for blending and for writing (section 17.4.1) using each buffer's color for C_d. If a color buffer has no A value, then A_d is taken to be 1.

17.3.7 sRGB Conversion

If FRAMEBUFFER_SRGB is enabled and the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the destination buffer is SRGB[1] (see section 9.2.3), the R, G, and B values after blending are converted into the non-linear sRGB color space by computing

$$c_s = \begin{cases} 0.0, & c_l \leq 0 \\ 12.92 c_l, & 0 < c_l < 0.0031308 \\ 1.055 c_l^{0.41666} - 0.055, & 0.0031308 \leq c_l < 1 \\ 1.0, & c_l \geq 1 \end{cases} \tag{17.1}$$

where c_l is the R, G, or B element and c_s is the result (effectively converted into an sRGB color space).

If FRAMEBUFFER_SRGB is disabled or the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING is not SRGB, then

$$c_s = c_l.$$

[1]Note that only unsigned normalized fixed-point color buffers may be SRGB-encoded. Signed normalized fixed-point + SRGB encoding is not defined.

The resulting c_s values for R, G, and B, and the unmodified A form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range $[0, 1]$ and then converted to a fixed-point value using equation 2.3. The resulting four values are sent to the subsequent dithering operation.

17.3.8 Dithering

Dithering selects between two representable color values or indices. A representable value is a value that has an exact representation in the color buffer. In RGBA mode dithering selects, for each color component, either the largest representable color value (for that particular color component) that is less than or equal to the incoming color component value, c, or the smallest representable color value that is greater than or equal to c. The selection may depend on the x_w and y_w coordinates of the pixel, as well as on the exact value of c. If one of the two values does not exist, then the selection defaults to the other value.

In color index mode dithering selects either the largest representable index that is less than or equal to the incoming color value, c, or the smallest representable index that is greater than or equal to c. If one of the two indices does not exist, then the selection defaults to the other value.

Many dithering selection algorithms are possible, but an individual selection must depend only on the incoming color index or component value and the fragment's x and y window coordinates. If dithering is disabled, then one of the two values above is selected, in an implementation-dependent manner that must not depend on the x_w and y_w coordinates of the pixel. A color index is rounded to the nearest representable index value.

Dithering is enabled and disabled by calling **Enable** or **Disable** with *target* `DITHER`. The state required is a single bit. Initially, dithering is enabled.

17.3.9 Logical Operation

Finally, a logical operation is applied between the incoming fragment's color or index values and the color or index values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment's (x_w, y_w) coordinates.

The logical operation on color indices is enabled or disabled by calling **Enable** or **Disable** with *target* `INDEX_LOGIC_OP`. (For compatibility with GL version 1.0, *target* `LOGIC_OP` may also be used.)

The logical operation on color values is enabled or disabled by calling **Enable** or **Disable** with *target* `COLOR_LOGIC_OP`. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of the value of `BLEND`. If

Logicop Mode	Operation
CLEAR	0
AND	$s \wedge d$
AND_REVERSE	$s \wedge \neg d$
COPY	s
AND_INVERTED	$\neg s \wedge d$
NOOP	d
XOR	s xor d
OR	$s \vee d$
NOR	$\neg(s \vee d)$
EQUIV	$\neg(s$ xor $d)$
INVERT	$\neg d$
OR_REVERSE	$s \vee \neg d$
COPY_INVERTED	$\neg s$
OR_INVERTED	$\neg s \vee d$
NAND	$\neg(s \wedge d)$
SET	all 1's

Table 17.3: Logical operation *op* arguments to **LogicOp** and their corresponding operations.

multiple fragment colors are being written to multiple buffers (see section 17.4.1), the logical operation is computed and applied separately for each fragment color and the corresponding buffer.

Logical operation has no effect on a floating-point destination color buffer, or when FRAMEBUFFER_SRGB is enabled and the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the destination buffer is SRGB (see section 9.2.3). However, if logical operation is enabled, blending is still disabled.

The logical operation is selected by

`void` **LogicOp**(`enum` *op*);

op must be one of the logicop modes in table 17.3, which also describes the resulting operation when that mode is selected. s is the value of the incoming fragment and d is the value stored in the framebuffer.

Logical operations are performed independently for each color index buffer that is selected for writing, or for each red, green, blue, and alpha value of each

color buffer that is selected for writing. The required state is an integer indicating the logical operation, and two bits indicating whether the logical operation is enabled or disabled. The initial state is for the logic operation to be given by COPY, and to be disabled.

Errors

An INVALID_VALUE error is generated if *op* is not one of the logicop modes in table 17.3.

17.3.10 Additional Multisample Fragment Operations

If the **DrawBuffer** mode (see section 17.4.1) is NONE, no change is made to any multisample or color buffer. Otherwise, fragment processing is as described below.

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, the alpha test, stencil test, depth test, blending, dithering, and logical operations are performed for each pixel sample, rather than just once for each fragment. Failure of the alpha, stencil, or depth test results in termination of the processing of that sample, rather than discarding of the fragment. All operations are performed on the color, depth, and stencil values stored in the multisample renderbuffer attachments if a draw framebuffer object is bound, or otherwise in the multisample buffer of the default framebuffer. The contents of the color buffers are not modified at this point.

Stencil, depth, blending, dithering, and logical operations are performed for a pixel sample only if that sample's fragment coverage bit is a value of 1. If the corresponding coverage bit is 0, no operations are performed for that sample.

If MULTISAMPLE is disabled, and the value of SAMPLE_BUFFERS is one, the fragment may be treated exactly as described above, with optimization possible because the fragment coverage must be set to full coverage. Further optimization is allowed, however. An implementation may choose to identify a centermost sample, and to perform alpha, stencil, and depth tests on only that sample. Regardless of the outcome of the stencil test, all multisample buffer stencil sample values are set to the appropriate new stencil value. If the depth test passes, all multisample buffer depth sample values are set to the depth of the fragment's centermost sample's depth value, and all multisample buffer color sample values are set to the color value of the incoming fragment. Otherwise, no change is made to any multisample buffer color or depth value.

If a draw framebuffer object is not bound, after all operations have been completed on the multisample buffer, the sample values for each color in the multisample buffer are combined to produce a single color value, and that value is written

into the corresponding color buffers selected by **DrawBuffer** or **DrawBuffers**. An implementation may defer the writing of the color buffers until a later time, but the state of the framebuffer must behave as if the color buffers were updated as each fragment was processed. The method of combination is not specified. If the framebuffer contains sRGB values, then it is recommended that the an average of sample values is computed in a linearized space, as for blending (see section 17.3.6). Otherwise, a simple average computed independently for each color component is recommended.

17.4 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

17.4.1 Selecting Buffers for Writing

The first such operation is controlling the color buffers into which each of the fragment color values is written. This is accomplished with either ***DrawBuffer** or ***DrawBuffers** commands described below.

The set of buffers of a framebuffer object to which fragment color zero is written is controlled with the commands

```
void DrawBuffer( enum buf );
void NamedFramebufferDrawBuffer( uint framebuffer,
    enum buf );
```

For **DrawBuffer**, the framebuffer object is that bound to the DRAW_FRAMEBUFFER binding. For **NamedFramebufferDrawBuffer**, *framebuffer* is zero or the name of a framebuffer object. If *framebuffer* is zero, then the default draw framebuffer is affected.

If the default framebuffer is affected (see section 9), *buf* must be one of the values listed in table 17.4. In this case, *buf* is a symbolic constant specifying zero, one, two, or four buffers for writing. These constants refer to the four potentially visible buffers (front left, front right, back left, and back right), and to the auxiliary buffers.

Arguments other than AUXi that omit reference to LEFT or RIGHT refer to both left and right buffers. Arguments other than AUXi that omit reference to FRONT or BACK refer to both front and back buffers. AUXi enables drawing only to auxiliary buffer i. Each AUXi adheres to AUXi = AUX0 + i, and i must be in the range zero to the value of AUX_BUFFERS minus one.

If a framebuffer object is affected, *buf* must be one of the values listed in table 17.5, which summarizes the constants and the buffers they indicate. In this case, *buf* specifies a single color buffer for writing. Specifying COLOR_ATTACHMENTi enables drawing only to the image attached to the framebuffer at that attachment point.

Errors

An INVALID_OPERATION error is generated by **NamedFramebufferDrawBuffer** if *framebuffer* is not zero or the name of an existing framebuffer object.

An INVALID_ENUM error is generated if *buf* is not one of the values in tables 17.5 or 17.4.

An INVALID_OPERATION error is generated if the default framebuffer is affected and *buf* is a value (other than NONE) that does not indicate one of the color buffers allocated to the default framebuffer.

An INVALID_OPERATION error is generated if a framebuffer object is affected and *buf* is one of the constants from table 17.4 (other than NONE), or COLOR_ATTACHMENTm and m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

DrawBuffer will set the draw buffer for fragment colors other than zero to NONE.

The set of buffers of a framebuffer object to which all fragment colors are written is controlled with the commands

The commands

```
void DrawBuffers( sizei n, const enum *bufs );
void NamedFramebufferDrawBuffers( uint framebuffer,
   sizei n, const enum *bufs );
```

For **DrawBuffers**, the framebuffer object is that bound to the DRAW_FRAMEBUFFER binding. For **NamedFramebufferDrawBuffers**, *framebuffer* is the name of the framebuffer object. If *framebuffer* is zero, then the default framebuffer is affected.

n specifies the number of buffers in *bufs*. *bufs* is a pointer to an array of values specifying the buffer to which each fragment color is written.

Each buffer listed in *bufs* must be one of the values from tables 17.5 or 17.6. Further, acceptable values for the constants in *bufs* depend on whether the default framebuffer or a framebuffer object is affected. For more information about framebuffer objects, see section 9.

Symbolic Constant	Front Left	Front Right	Back Left	Back Right	Aux i
NONE					
FRONT_LEFT	•				
FRONT_RIGHT		•			
BACK_LEFT			•		
BACK_RIGHT				•	
FRONT	•	•			
BACK			•	•	
LEFT	•		•		
RIGHT		•		•	
FRONT_AND_BACK	•	•	•	•	
AUXi					•

Table 17.4: Arguments to **DrawBuffer** when the context is bound to a default framebuffer, and the buffers they indicate. The same arguments are valid for **ReadBuffer**, but only a single buffer is selected as discussed in section 18.2.1.

Symbolic Constant	Meaning
NONE	No buffer
COLOR_ATTACHMENTi (see caption)	Output fragment color to image attached at color attachment point i

Table 17.5: Arguments to **DrawBuffer**(s) and **ReadBuffer** when the context is bound to a framebuffer object, and the buffers they indicate. i in COLOR_ATTACHMENTi may range from zero to the value of MAX_COLOR_ATTACHMENTS minus one.

Symbolic Constant	Front Left	Front Right	Back Left	Back Right	Aux i
NONE					
FRONT_LEFT	•				
FRONT_RIGHT		•			
BACK_LEFT			•		
BACK_RIGHT				•	
AUXi					•

Table 17.6: Arguments to **DrawBuffers** when the default framebuffer is affected, and the buffers they indicate.

If the default framebuffer is affected, then each of the constants must be one of the values listed in table 17.6 or the special value BACK. When BACK is used, n must be 1 and color values are written into the left buffer for single-buffered contexts, or into the back left buffer for double-buffered contexts.

If a framebuffer object is affected, then each of the constants must be one of the values listed in table 17.5.

In both cases, the draw buffers being defined correspond in order to the respective fragment colors. The draw buffer for fragment colors beyond n is set to NONE.

The maximum number of draw buffers is implementation-dependent. The number of draw buffers supported may be queried by calling **GetIntegerv** with *pname* MAX_DRAW_BUFFERS.

Except for NONE, a buffer may not appear more than once in the array pointed to by *bufs*.

If fixed-function fragment shading is being performed, **DrawBuffers** specifies a set of draw buffers into which the fragment color is written.

If a fragment shader writes to gl_FragColor, **DrawBuffers** specifies a set of draw buffers into which the single fragment color defined by gl_FragColor is written. If a fragment shader writes to gl_FragData or a user-defined output variable, **DrawBuffers** specifies a set of draw buffers into which each of the multiple output colors defined by these variables are separately written. If a fragment shader writes to none of gl_FragColor, gl_FragData, nor any user-defined output variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color. If some, but not all elements of gl_FragData or of theser-defined output variables are written, the values of fragment colors corresponding to unwritten elements oriables are similarly un-

defined.

The order of writes to gl_FragData and to user-defined output variables is undefined. If the same image is attached to multiple attachment points of a framebuffer object and different values are written to outputs bound to those attachments, the resulting value in the attached image is undefined. Similarly undefined behavior results during any other per-fragment operations where a multiply-attached image may be written to by more than one output, such as during blending.

Errors

An INVALID_OPERATION error is generated by **NamedFramebufferDrawBuffers** if *framebuffer* is not zero or the name of an existing framebuffer object.

An INVALID_VALUE error is generated if *n* is negative, or greater than the value of MAX_DRAW_BUFFERS.

An INVALID_ENUM error is generated if any value in *bufs* is not one of the values in tables 17.5 or 17.6.

An INVALID_OPERATION error is generated if a buffer other than NONE is specified more than once in the array pointed to by *bufs*.

An INVALID_ENUM error is generated if any value in *bufs* is FRONT, LEFT, RIGHT, or FRONT_AND_BACK. This restriction applies to both the default framebuffer and framebuffer objects, and exists because these constants may themselves refer to multiple buffers, as shown in table 17.4.

An INVALID_OPERATION error is generated for the default framebuffer if any value in *bufs* is BACK, and *n* is not one.

An INVALID_OPERATION error is generated if the default framebuffer is affected and any value in *bufs* is a constant (other than NONE or BACK) that does not indicate one of the color buffers allocated to the default framebuffer.

An INVALID_OPERATION error is generated if the GL is bound to a draw framebuffer object and the ith argument is a value other than COLOR_ATTACHMENTi or NONE.

Indicating a buffer or buffers using **DrawBuffer** or **DrawBuffers** causes subsequent pixel color value writes to affect the indicated buffers. If a framebuffer object is affected and a draw buffer selects an attachment that has no image attached, then that fragment color is not written.

Specifying NONE as the draw buffer for a fragment color will inhibit that fragment color from being written.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only

front buffers, while double buffered contexts include both front and back buffers. The type of context is selected at GL initialization.

The state required to handle color buffer selection for each framebuffer is an integer for each supported fragment color. For the default framebuffer, in the initial state the draw buffer for fragment color zero is BACK if there is a back buffer; FRONT if there is no back buffer; and NONE if no default framebuffer is associated with the context. For framebuffer objects, in the initial state the draw buffer for fragment color zero is COLOR_ATTACHMENT0. For both the default framebuffer and framebuffer objects, the initial state of draw buffers for fragment colors other then zero is NONE.

The draw buffer of the currently bound draw framebuffer selected for fragment color i may be queried by calling **GetIntegerv** with *pname* set to DRAW_BUFFERi. DRAW_BUFFER is equivalent to DRAW_BUFFER0.

17.4.2 Fine Control of Buffer Updates

Writing of bits to each of the logical buffers after all per-fragment operations have been performed may be *masked*. The commands

```
void IndexMask( uint mask );
void ColorMask( boolean r, boolean g, boolean b,
  boolean a );
void ColorMaski( uint buf, boolean r, boolean g,
  boolean b, boolean a );
```

control writes to the active draw buffers.

The least significant n bits of *mask*, where n is the number of bits in a color index buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the color index buffer (or buffers) is written; where a 0 appears, the bit is not written. This mask applies only in color index mode.

In RGBA mode, **ColorMask** and **ColorMaski** are used to mask the writing of R, G, B and A values to the draw buffer or buffers. **ColorMaski** sets the mask for a particular draw buffer. The mask for DRAW_BUFFERi is modified by passing i as the parameter *buf*. *r*, *g*, *b*, and *a* indicate whether R, G, B, or A values, respectively, are written or not (a value of TRUE means that the corresponding value is written). The mask specified by *r*, *g*, *b*, and *a* is applied to the color buffer associated with DRAW_BUFFERi. For any i where the value of DRAW_BUFFERi is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK, specifying multiple color buffers, the mask is applied to all of the buffers.

ColorMask sets the mask for all draw buffers to the same values as specified by *r*, *g*, *b*, and *a*.

Errors

An INVALID_VALUE error is generated by **ColorMaski** if *buf* is greater than the value of MAX_DRAW_BUFFERS minus one.

In the initial state, all bits (in color index mode) and all color values (in RGBA mode) are enabled for writing for all draw buffers.

The value of the color writemask for draw buffer i may be queried by calling **GetBooleani_v** with *target* COLOR_WRITEMASK and *index* i. The value of the color writemask for draw buffer zero may also be queried by calling **GetBooleanv** with *pname* COLOR_WRITEMASK.

The depth buffer can be enabled or disabled for writing z_w values using

```
void DepthMask( boolean mask );
```

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The commands

```
void StencilMask( uint mask );
void StencilMaskSeparate( enum face, uint mask );
```

control the writing of particular bits into the stencil planes.

The least significant s bits of *mask*, where s is the number of bits in the stencil buffer, specify an integer mask. Where a 1 appears in this mask, the corresponding bit in the stencil buffer is written; where a 0 appears, the bit is not written. The *face* parameter of **StencilMaskSeparate** can be FRONT, BACK, or FRONT_AND_BACK and indicates whether the front or back stencil mask state is affected. **StencilMask** sets both front and back stencil mask state to identical values.

Fragments generated by front-facing primitives use the front mask and fragments generated by back-facing primitives use the back mask (see section 17.3.3). The clear operation always uses the front stencil write mask when clearing the stencil buffer.

The state required for the various masking operations is an integer for color indices, two integers for the front and back stencil values, and a bit for depth values. A set of four bits is also required indicating which color components of an RGBA value should be written. In the initial state, the integer masks are all ones, as are the bits controlling depth value and RGBA component writing.

17.4.2.1 Fine Control of Multisample Buffer Updates

When the value of SAMPLE_BUFFERS is one, **ColorMask**, **DepthMask**, and **StencilMask** or **StencilMaskSeparate** control the modification of values in the multisample buffer. The color mask has no effect on modifications to the color buffers. If the color mask is entirely disabled, the color sample values must still be combined (as described above) and the result used to replace the color values of the buffers enabled by **DrawBuffer**.

17.4.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

void **Clear**(bitfield *buf*);

is zero or the bitwise OR of one or more values indicating which buffers are to be cleared. The values are ACCUM_BUFFER_BIT, COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, and STENCIL_BUFFER_BIT, indicating the accumulation buffer, the buffers currently enabled for color writing, the depth buffer, and the stencil buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If *buf* is zero, no buffers are cleared.

Errors

An INVALID_VALUE error is generated if *buf* contains any bits other than ACCUM_BUFFER_BIT, COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, or STENCIL_BUFFER_BIT.

void **ClearColor**(float *r*, float *g*, float *b*, float *a*);

sets the clear value for fixed-point and floating-point color buffers in RGBA mode. The specified components are stored as floating-point values.

The command

void **ClearIndex**(float *index*);

sets the clear color index. *index* is converted to a fixed-point value with unspecified precision to the left of the binary point; the integer part of this value is then masked with $2^m - 1$, where m is the number of bits in a color index value stored in the framebuffer.

The command

```
void ClearDepth( double d );
void ClearDepthf( float d );
```

sets the depth value used when clearing the depth buffer. d is clamped to the range $[0, 1]$ when specified. When clearing a fixed-point depth buffer, d is converted to fixed-point according to the rules for a window z value given in section 13.8.1. No conversion is applied when clearing a floating-point depth buffer.

The command

```
void ClearStencil( int s );
```

takes a single integer argument that is the value to which to clear the stencil buffer. s is masked to the number of bitplanes in the stencil buffer.

The command

```
void ClearAccum( float r, float g, float b, float a );
```

takes four floating-point arguments that are the values, in order, to which to set the R, G, B, and A values of the accumulation buffer (see the next section). These values are clamped to the range $[-1, 1]$ when they are specified.

When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, sRGB conversion (see section 17.3.7), and dithering. The masking operations described in section 17.4.2 are also applied. If a buffer is not present, then a **Clear** directed at that buffer has no effect.

Unsigned normalized fixed-point and signed normalized fixed-point RGBA color buffers are cleared to color values derived by clamping each component of the clear color to the range $[0, 1]$ or $[-1, 1]$ respectively, then converting the (possibly sRGB converted and/or dithered) color to fixed-point using equations 2.3 or 2.4, respectively. The result of clearing integer color buffers is undefined.

The state required for clearing is a clear value for each of the color buffer, the accumulation buffer, the depth buffer, and the stencil buffer. Initially, the RGBA color clear value is $(0.0, 0.0, 0.0, 0.0)$, the accumulation buffer clear value is $(0.0, 0.0, 0.0, 0.0)$, the clear color index is 0, the depth buffer clear value is 1.0, and the stencil buffer clear index is 0.

17.4.3.1 Clearing Individual Buffers

Individual buffers of a framebuffer object may be cleared with the commands

```
void ClearBuffer{if ui}v( enum buffer, int drawbuffer,
  const T *value );
```

```
void ClearNamedFramebuffer{if ui}v( uint framebuffer,
    enum buffer, int drawbuffer, const T *value );
```

For **ClearBuffer***, the framebuffer object is the bound draw framebuffer object. For **ClearNamedFramebuffer***, *framebuffer* is the name of the framebuffer object. If *framebuffer* is zero, the default draw framebuffer is affected.

buffer and *drawbuffer* identify a buffer to clear, and *value* specifies the value or values to clear it to. The ***fv**, ***iv**, and ***uiv** forms of these commands should be used to clear fixed- and floating-point, signed integer, and unsigned integer color buffers respectively.

If *buffer* is `COLOR`, a particular draw buffer `DRAW_BUFFER`i is specified by passing i as the parameter *drawbuffer*, and *value* points to a four-element vector specifying the R, G, B, and A color to clear that draw buffer to. If the value of `DRAW_BUFFER`i is `NONE`, the command has no effect. Otherwise, the value of `DRAW_BUFFER`i is one of the possible values in tables 17.4 and 17.5 identifying one or more color buffers, each of which is cleared to the same value. Clamping and conversion for fixed-point color buffers are performed in the same fashion as for **ClearColor**.

If *buffer* is `DEPTH`, *drawbuffer* must be zero, and *value* points to the single depth value to clear the depth buffer to. Clamping and type conversion for fixed-point depth buffers are performed in the same fashion as for **ClearDepth**. Only the ***fv** forms of these commands should be used to clear depth buffers; other forms do not accept a *buffer* of `DEPTH`.

If *buffer* is `STENCIL`, *drawbuffer* must be zero, and *value* points to the single stencil value to clear the stencil buffer to. Masking is performed in the same fashion as for **ClearStencil**. Only the ***iv** forms of these commands should be used to clear stencil buffers; other forms do not accept a *buffer* of `STENCIL`.

Both depth and stencil buffers of a framebuffer object may be cleared with the commands

```
void ClearBufferfi( enum buffer, int drawbuffer,
    float depth, int stencil );
void ClearNamedFramebufferfi( uint framebuffer,
    enum buffer, int drawbuffer, float depth, int stencil );
```

For **ClearBufferfi**, the framebuffer object is the bound draw framebuffer object. For **ClearNamedFramebufferfi**, *framebuffer* is the name of the framebuffer object. If *framebuffer* is zero, the default draw framebuffer is affected.

buffer must be `DEPTH_STENCIL` and *drawbuffer* must be zero. *depth* and *stencil* are the values to clear the depth and stencil buffers to, respectively. Clamping

and type conversion of *depth* for fixed-point depth buffers is performed in the same fashion as for **ClearDepth**. Masking of *stencil* for stencil buffers is performed in the same fashion as for **ClearStencil**. These commands are equivalent to clearing the depth and stencil buffers separately, but may be faster when a buffer of internal format DEPTH_STENCIL is being cleared. The same per-fragment and masking operations defined for **Clear** are applied.

For all forms of **ClearBuffer*** and **ClearNamedFramebuffer***, the result of these commands is undefined if no conversion between the type of the specified *value* and the type of the buffer being cleared is defined (for example, if **ClearBufferiv** is called for a fixed- or floating-point buffer, or if **ClearBufferfv** is called for a signed or unsigned integer buffer). This is not an error.

Errors

An INVALID_OPERATION error is generated by **ClearNamedFramebuffer*** if *framebuffer* is not zero or the name of an existing framebuffer object.

An INVALID_ENUM error is generated by **ClearBufferiv** and **ClearNamedFramebufferiv** if *buffer* is not COLOR or STENCIL.

An INVALID_ENUM error is generated by **ClearBufferuiv** and **ClearNamedFramebufferuiv** if *buffer* is not COLOR.

An INVALID_ENUM error is generated by **ClearBufferfv** and **ClearNamedFramebufferfv** if *buffer* is not COLOR or DEPTH.

An INVALID_ENUM error is generated by **ClearBufferfi** and **ClearNamedFramebufferfi** if *buffer* is not DEPTH_STENCIL.

An INVALID_VALUE error is generated if *buffer* is COLOR and *drawbuffer* is negative, or greater than the value of MAX_DRAW_BUFFERS minus one; or if *buffer* is DEPTH, STENCIL, or DEPTH_STENCIL and *drawbuffer* is not zero.

An INVALID_OPERATION error is generated if *buffer* is COLOR and the GL is in color index mode.

17.4.3.2 Clearing the Multisample Buffer

The color samples of the multisample buffer are cleared when one or more color buffers are cleared, as specified by the **Clear** mask bit COLOR_BUFFER_BIT and the **DrawBuffer** mode. If the **DrawBuffer** mode is NONE, the color samples of the multisample buffer cannot be cleared using **Clear**.

If the **Clear** mask bits DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT are set, then the corresponding depth or stencil samples, respectively, are cleared.

The **Clear*Buffer*** commands also clear color, depth, or stencil samples of

multisample buffers corresponding to the specified buffer.

Masking and scissoring affect clearing the multisample buffer in the same way as they affect clearing the corresponding color, depth, and stencil buffers.

17.4.4 Invalidating Framebuffer Contents

To signal that the GL need not preserve all contents of a framebuffer object (invalidating portions of every pixel or a subregion of pixels), use the commands

```
void InvalidateSubFramebuffer( enum target,
    sizei numAttachments, const enum *attachments, int x,
    int y, sizei width, sizei height );
void InvalidateNamedFramebufferSubData( uint framebuffer,
    sizei numAttachments, const enum *attachments, int x,
    int y, sizei width, sizei height );
```

For **InvalidateSubFramebuffer**, the framebuffer object is that bound to *target*, which must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. For **InvalidateNamedFramebufferSubData**, *framebuffer* is the name of the framebuffer object. If *framebuffer* is zero, the default draw framebuffer is affected.

numAttachments indicates how many attachments are supplied in the *attachments* list. If an attachment is specified that does not exist in the framebuffer object, it is ignored. *x* and *y* are the origin (with lower left-hand corner at $(0, 0)$) and *width* and *height* are the width and height, respectively, of the pixel rectangle to be invalidated. Any of these pixels lying outside of the window allocated to the current GL context (for the default framebuffer), or outside of the attachments of the framebuffer object, are ignored.

If the framebuffer object is not complete, these commands may be ignored.

Errors

An INVALID_ENUM error is generated by **InvalidateSubFramebuffer** if *target* is not FRAMEBUFFER, DRAW_FRAMEBUFFER, or READ_FRAMEBUFFER.

An INVALID_OPERATION error is generated by **InvalidateNamedFramebufferData** if *framebuffer* is not zero or the name of an existing framebuffer object.

An INVALID_VALUE error is generated if *numAttachments*, *width*, or *height* is negative.

An INVALID_ENUM error is generated if a framebuffer object is affected,

and any element of *attachments* is not one of the values in table 9.2.

An INVALID_OPERATION error is generated if *attachments* contains COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS.

An INVALID_ENUM error is generated if the default framebuffer is affected, and any elements of *attachments* are not one of:

- ACCUM, AUXi, FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, and BACK_RIGHT, identifying that specific buffer
- COLOR, which is treated as BACK_LEFT for a double-buffered context and FRONT_LEFT for a single-buffered context
- DEPTH, identifying the depth buffer
- STENCIL, identifying the stencil buffer.

The commands

```
void InvalidateFramebuffer( enum target,
    sizei numAttachments, const enum *attachments );
void InvalidateNamedFramebufferData( uint framebuffer,
    sizei numAttachments, const enum *attachments );
```

are equivalent to

```
InvalidateSubFramebuffer(target, numAttachments, attachments,
    0, 0, vw, vh);
```

and

```
InvalidateNamedFramebufferSubData(framebuffer, numAttachments,
    attachments, 0, 0, vw, vh);
```

respectively, where vw and vh are equal to the maximum viewport width and height, respectively, obtained by querying MAX_VIEWPORT_DIMS.

17.4.5 The Accumulation Buffer

Each portion of a pixel in the accumulation buffer consists of four values: one for each of R, G, B, and A. The accumulation buffer is controlled exclusively through the use of

```
void Accum( enum op, float value );
```

(except for clearing it). *op* is one of ACCUM, LOAD, RETURN, MULT, and ADD, indicating an accumulation buffer operation, and *value* is a floating-point value to be used in that operation.

When the scissor test is enabled (section 14.9.2), then only those pixels within the current scissor box are updated by any **Accum** operation; otherwise, all pixels in the window are updated. The accumulation buffer operations apply identically to every affected pixel, so we describe the effect of each operation on an individual pixel. Accumulation buffer values are taken to be signed values in the range $[-1, 1]$.

Using ACCUM obtains R, G, B, and A components from the buffer currently selected for reading (section 18.2). If the color buffer is fixed-point, each component is considered as an unsigned normalized value in the range $[0, 1]$ and is converted to floating-point using equation 2.1. If FRAMEBUFFER_SRGB is enabled and the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the buffer selected for reading is SRGB (see section 9.2.3), then the floating-point R, G, and B components are considered to be encoded for the sRGB color space and are linearized using equation 8.18. Each component is then multiplied by *value*. The result of this multiplication is then added to the corresponding color component currently in the accumulation buffer, replacing that component.

The LOAD operation has the same effect as ACCUM, but the computed values replace the corresponding accumulation buffer components rather than being added to them.

The RETURN operation takes each color value from the accumulation buffer, multiplies each of the R, G, B, and A components by *value*. If fragment color clamping is enabled, the results are then clamped to the range $[0, 1]$. The resulting color value is placed in the buffers currently enabled for color writing as if it were a fragment produced from rasterization, except that the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test (section 14.9.2), sRGB conversion (see section 17.3.7), and dithering (section 17.3.8). Color masking (section 17.4.2) is also applied.

The MULT operation multiplies each R, G, B, and A in the accumulation buffer by *value* and then returns the scaled color components to their corresponding accumulation buffer locations. ADD is the same as MULT except that *value* is added to each of the color components.

The color components operated on by **Accum** must be clamped only if the operation is RETURN. In this case, a value sent to the enabled color buffers is first clamped to $[0, 1]$. Otherwise, results are undefined if the result of an operation on

a color component is out of the range $[-1, 1]$.

Errors

An INVALID_OPERATION error is generated if there is no accumulation buffer; if the DRAW_FRAMEBUFFER and READ_FRAMEBUFFER bindings (see section 9.4.2) do not refer to the same object; or if the GL is in color index mode,

No state (beyond the accumulation buffer itself) is required for accumulation buffering.

Chapter 18

Drawing, Reading, and Copying Pixels

Pixels may be written to the framebuffer using **DrawPixels** and read from the framebuffer using **ReadPixels**. **CopyPixels** and **BlitFramebuffer** can be used to copy a block of pixels from one portion of the framebuffer to another.

18.1 Drawing Pixels

Pixel rectangles may be drawn using the command

```
void DrawPixels( sizei width, sizei height, enum format,
    enum type, const void *data );
```

Results of rasterization are undefined if any of the selected draw buffers of the draw framebuffer have an integer format and no fragment shader is active.

Errors

An INVALID_VALUE error is generated if *width* or *height* is negative.

An INVALID_OPERATION error is generated if the GL is in color index mode and *format* is not one of COLOR_INDEX, STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL.

An INVALID_OPERATION error is generated if *format* contains integer components, as shown in table 8.8.

An INVALID_FRAMEBUFFER_OPERATION error is generated if the object bound to DRAW_FRAMEBUFFER_BINDING is not framebuffer complete (see

section 9.4.2).

Calling **DrawPixels** with a *type* of BITMAP is a special case in which the data are a series of GL ubyte values. Each ubyte value specifies 8 1-bit elements with its 8 least-significant bits. The 8 single-bit elements are ordered from most significant to least significant if the value of UNPACK_LSB_FIRST is FALSE; otherwise, the ordering is from least significant to most significant. The values of bits other than the 8 least significant in each ubyte are not significant.

The first element of the first row is the first bit (as defined above) of the ubyte pointed to by the pointer passed to **DrawPixels**. The first element of the second row is the first bit (again as defined above) of the ubyte at location $p + k$, where k is computed as

$$k = a \left\lceil \frac{l}{8a} \right\rceil \tag{18.1}$$

There is a mechanism for selecting a sub-rectangle of elements from a BITMAP image as well. Before obtaining the first element from memory, the pointer supplied to **DrawPixels** is effectively advanced by UNPACK_SKIP_ROWS $* k$ ubytes. Then UNPACK_SKIP_PIXELS 1-bit elements are ignored, and the subsequent *width* 1-bit elements are obtained, without advancing the ubyte pointer, after which the pointer is advanced by k ubytes. *height* sets of *width* elements are obtained this way.

Once pixels are transferred, **DrawPixels** performs final conversion on pixel values (see section 18.1.1, then converts them to fragments (see section 18.1.2 which are processed in the same fashion as fragments generated by rasterization (see chapters 15 and 16).

18.1.1 Final Conversion

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by $2^n - 1$, where n is the number of bits in an index buffer.

For integer RGBA components, no conversion is performed. For floating-point RGBA components, if fragment color clamping is enabled, each element is clamped to $[0, 1]$, and may be converted to fixed-point according to equation 2.3. If fragment color clamping is disabled, RGBA components are unmodified. Fragment color clamping is controlled by calling

`void` **ClampColor**(`enum` *target*, `enum` *clamp*);

with *target* set to CLAMP_FRAGMENT_COLOR. If *clamp* is TRUE, fragment color clamping is enabled; if *clamp* is FALSE, fragment color clamping is disabled. If

clamp is FIXED_ONLY, fragment color clamping is enabled if all enabled color buffers have fixed-point components.

If fragment color clamping is enabled, querying of the texture border color, texture environment color, fog color, alpha test reference value, blend color, and RGBA clear color will clamp the corresponding state values to $[0, 1]$ before returning them. This behavior provides compatibility with previous versions of the GL that clamped these values when specified.

For a depth component, an element is processed according to the depth buffer's representation. For fixed-point depth buffers, the element is first clamped to the range $[0, 1]$ and then converted to fixed-point as if it were a window z value (see section 13.8.1). Conversion is not necessary when the depth buffer uses a floating-point representation, but clamping is.

Stencil indices are masked by $2^n - 1$, where n is the number of bits in the stencil buffer.

The state required for fragment color clamping is a three-valued integer. The initial value of fragment color clamping is FIXED_ONLY.

18.1.2 Conversion to Fragments

The conversion of a group to fragments is controlled with

`void` **PixelZoom**(`float` z_x, `float` z_y);

Let (x_{rp}, y_{rp}) be the current raster position (section 14.7). (If the current raster position is invalid, then **DrawPixels** is ignored; pixel transfer operations do not update the histogram or minmax tables, and no fragments are generated. However, the histogram and minmax tables are updated even if the corresponding fragments are later rejected by the pixel ownership (section 14.9.1) or scissor (section 14.9.2) tests.) If a particular group (index or components) is the nth in a row and belongs to the mth row, consider the region in window coordinates bounded by the rectangle with corners

$$(x_{rp} + z_x n, y_{rp} + z_y m) \quad \text{and} \quad (x_{rp} + z_x(n+1), y_{rp} + z_y(m+1))$$

(either z_x or z_y may be negative). A fragment representing group (n, m) is produced for each framebuffer pixel inside, or on the bottom or left boundary, of this rectangle.

A fragment arising from a group consisting of color data takes on the color index or color components of the group and the current raster position's associated depth value, while a fragment arising from a depth component takes that component's depth value and the current raster position's associated color index or color

components. In both cases, the fog coordinate is taken from the current raster position's associated raster distance, the secondary color is taken from the current raster position's associated secondary color, and texture coordinates are taken from the current raster position's associated texture coordinates. Groups arising from **DrawPixels** with a *format* of DEPTH_STENCIL or STENCIL_INDEX are treated specially and are described in section 18.1.4.

18.1.3 Pixel Rectangle Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then pixel rectangles are rasterized using the following algorithm. Let (X_{rp}, Y_{rp}) be the current raster position. (If the current raster position is invalid, then **DrawPixels** is ignored.) If a particular group (index or components) is the nth in a row and belongs to the mth row, consider the region in window coordinates bounded by the rectangle with corners

$$(X_{rp} + Z_x * n, Y_{rp} + Z_y * m)$$

and

$$(X_{rp} + Z_x * (n + 1), Y_{rp} + Z_y * (m + 1))$$

where Z_x and Z_y are the pixel zoom factors specified by **PixelZoom**, and may each be either positive or negative. A fragment representing group (n, m) is produced for each framebuffer pixel with one or more sample points that lie inside, or on the bottom or left boundary, of this rectangle. Each fragment so produced takes its associated data from the group and from the current raster position, in a manner consistent with the discussion in section 18.1.2. All depth and color sample values are assigned the same value, taken either from their group (for depth and color component groups) or from the current raster position (if they are not). All sample values are assigned the same fog coordinate and the same set of texture coordinates, taken from the current raster position.

A single pixel rectangle will generate multiple, perhaps very many fragments for the same framebuffer pixel, depending on the pixel zoom factors.

18.1.4 Writing to the Stencil or Depth/Stencil Buffers

If the *format* argument to **DrawPixels** is STENCIL_INDEX or DEPTH_STENCIL, all operations described for **DrawPixels** take place, but window (x, y) coordinates, each with the corresponding stencil index, or depth value and stencil index, are produced in lieu of fragments. Each coordinate-data pair is sent directly to the per-fragment operations, bypassing the fragment processing operations in chapters 15

and 16. Each pair is then treated as a fragment for purposes of the pixel ownership and scissor tests; all other per-fragment operations are bypassed. Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current front stencil mask (set with **StencilMask** or **StencilMaskSeparate**). If a depth component is present, and the setting of **DepthMask** is not FALSE, it is also written to the framebuffer; the setting of **DepthFunc** is ignored.

Errors

An INVALID_OPERATION error is generated if the *format* argument is STENCIL_INDEX and there is no stencil buffer, or if *format* is DEPTH_STENCIL and there is not both a depth buffer and a stencil buffer.

18.2 Reading Pixels

The method for reading pixels from the framebuffer and placing them in pixel pack buffer or client memory is diagrammed in figure 18.1. We describe the stages of the pixel reading process in the order in which they occur.

18.2.1 Selecting Buffers for Reading

When reading pixels from a color buffer of a framebuffer object, the buffer selected for reading is termed the *read buffer*, and is controlled with the commands

```
void ReadBuffer( enum src );
void NamedFramebufferReadBuffer( uint framebuffer,
    enum src );
```

For **ReadBuffer**, the target framebuffer object is that bound to READ_FRAMEBUFFER. For **NamedFramebufferReadBuffer**, *framebuffer* is zero or the name of the target framebuffer object. If *framebuffer* is zero, then the default read framebuffer is affected.

If the default framebuffer is affected (see section 9), *src* must be one of the values listed in table 17.4, including NONE. FRONT_AND_BACK, FRONT, and LEFT refer to the front left buffer, BACK refers to the back left buffer, and RIGHT refers to the front right buffer. Other constants correspond directly to the buffers that they name. The initial value of the read buffer for the default framebuffer is FRONT if there is no back buffer; BACK if there is a back buffer; and NONE if no default framebuffer is associated with the context.

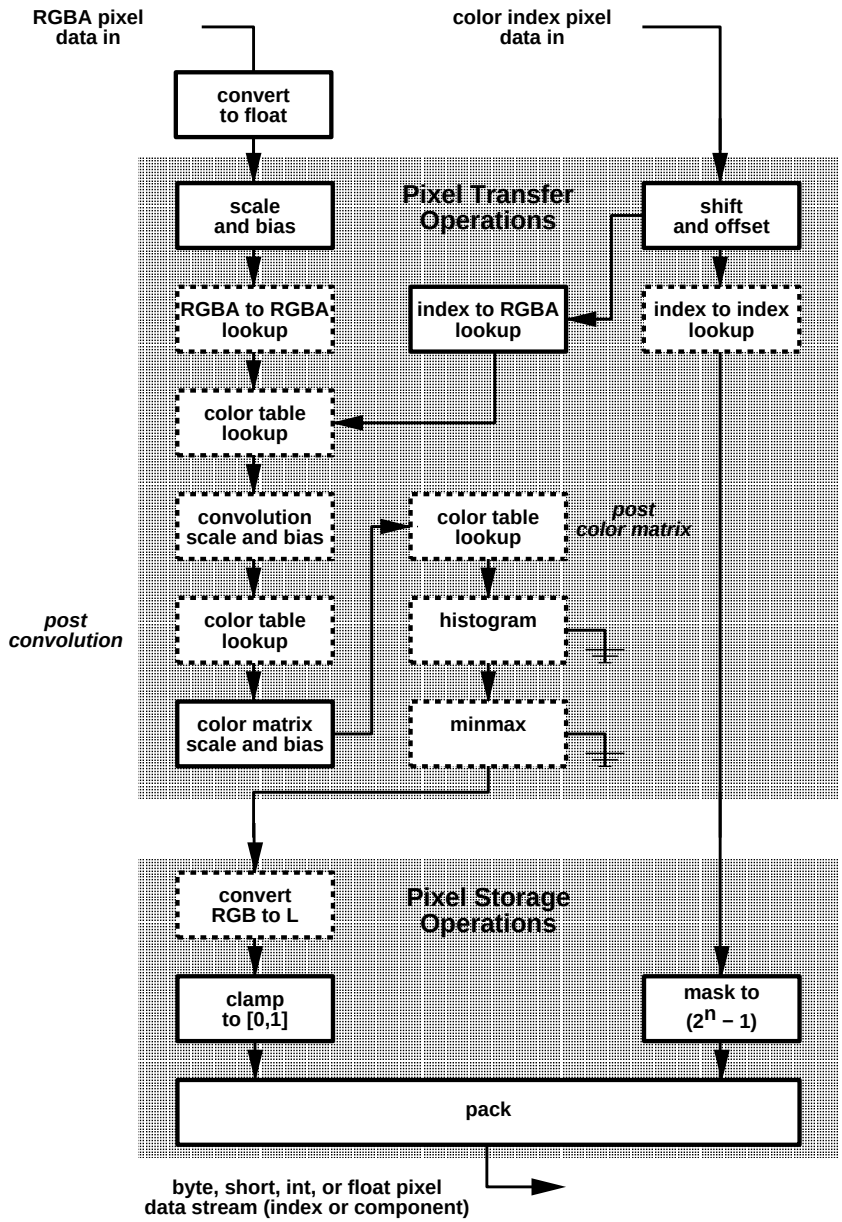


Figure 18.1. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled, except in the case of "convert RGB to L", which is only applied when reading color data in luminosity formats. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

If a framebuffer object is affected, *src* must be one of the values listed in table 17.5, including NONE. Specifying COLOR_ATTACHMENT*i* enables reading from the image attached to the framebuffer at that attachment point. The initial value of the read buffer for framebuffer objects is COLOR_ATTACHMENT0.

The read buffer of the currently bound read framebuffer may be queried by calling **GetIntegerv** with *pname* set to READ_BUFFER.

Errors

An INVALID_OPERATION error is generated by **NamedFramebufferReadBuffer** if *framebuffer* is not zero or the name of an existing framebuffer object.

An INVALID_ENUM error is generated if *src* is not one of the values in tables 17.4 or 17.5.

An INVALID_OPERATION error is generated if the default framebuffer is affected and *src* is a value (other than NONE) that does not indicate any of the color buffers allocated to the default framebuffer.

An INVALID_OPERATION error is generated if a framebuffer object is affected, and *src* is one of the constants from table 17.4 (other than NONE, or COLOR_ATTACHMENTm where m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS).

18.2.2 ReadPixels

Initially, zero is bound for the PIXEL_PACK_BUFFER, indicating that image read and query commands such as **ReadPixels** return pixel results into client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel pack buffer, then the pointer parameter is treated as an offset into the designated buffer object.

Pixels are read with the commands

```
void ReadPixels( int x, int y, sizei width, sizei height,
    enum format, enum type, void *data );
void ReadnPixels( int x, int y, sizei width,
    sizei height, enum format, enum type, sizei bufSize,
    void *data );
```

The arguments after *x* and *y* to **ReadPixels** are described in section 8.4.4. The pixel storage modes that apply to **ReadPixels** and other commands that query images (see section 8.11) are summarized in table 18.1.

Parameter Name	Type	Initial Value	Valid Range
PACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
PACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
PACK_ROW_LENGTH	integer	0	$[0, \infty)$
PACK_SKIP_ROWS	integer	0	$[0, \infty)$
PACK_SKIP_PIXELS	integer	0	$[0, \infty)$
PACK_ALIGNMENT	integer	4	1,2,4,8
PACK_IMAGE_HEIGHT	integer	0	$[0, \infty)$
PACK_SKIP_IMAGES	integer	0	$[0, \infty)$
PACK_COMPRESSED_BLOCK_WIDTH	integer	0	$[0, \infty)$
PACK_COMPRESSED_BLOCK_HEIGHT	integer	0	$[0, \infty)$
PACK_COMPRESSED_BLOCK_DEPTH	integer	0	$[0, \infty)$
PACK_COMPRESSED_BLOCK_SIZE	integer	0	$[0, \infty)$

Table 18.1: **PixelStore** parameters pertaining to **ReadPixels**, **GetColorTable**, **GetConvolutionFilter**, **GetSeparableFilter**, **GetHistogram**, **GetMinmax**, **GetPolygonStipple**, **GetCompressedTexImage** and **GetTexImage**.

Errors

An INVALID_OPERATION error is generated if the value of READ_FRAMEBUFFER_BINDING (see section 9) is non-zero, the read framebuffer is framebuffer complete, and the effective value of SAMPLE_BUFFERS for the read framebuffer is one.

An INVALID_VALUE error is generated if *width*, *height*, or *bufSize* is negative.

An INVALID_OPERATION error is generated by **ReadnPixels** if the buffer size required to store the requested data is greater than *bufSize*.

Preferred values for *format* and *type* may be determined by calling **GetIntegerv** with *pname*s IMPLEMENTATION_COLOR_READ_FORMAT and IMPLEMENTATION_COLOR_READ_TYPE, respectively. The preferred format may vary depending on the format of the selected read buffer of the currently bound read framebuffer.

Errors

An INVALID_OPERATION error is generated by **GetIntegerv** if *pname* is

IMPLEMENTATION_COLOR_READ_FORMAT or IMPLEMENTATION_COLOR_READ_TYPE and any of:

- the read framebuffer is not framebuffer complete.
- the read framebuffer is a framebuffer object, and the selected read buffer (see section 18.2.1) has no image attached.
- the selected read buffer is NONE.

18.2.3 Obtaining Pixels from the Framebuffer

If the *format* is DEPTH_COMPONENT, then values are obtained from the depth buffer.

If there is a multisample buffer (the value of SAMPLE_BUFFERS is one), then values are obtained from the depth samples in this buffer. It is recommended that the depth value of the centermost sample be used, though implementations may choose any function of the depth sample values at each pixel.

If the *format* is DEPTH_STENCIL, then values are taken from both the depth buffer and the stencil buffer. *type* must be UNSIGNED_INT_24_8 or FLOAT_32_UNSIGNED_INT_24_8_REV.

If there is a multisample buffer, then values are obtained from the depth and stencil samples in this buffer. It is recommended that the depth and stencil values of the centermost sample be used, though implementations may choose any function of the depth and stencil sample values at each pixel.

If the *format* is STENCIL_INDEX, then values are taken from the stencil buffer.

If there is a multisample buffer, then values are obtained from the stencil samples in this buffer. It is recommended that the stencil value of the centermost sample be used, though implementations may choose any function of the stencil sample values at each pixel.

For all other formats, values are obtained from the color buffer selected by the read buffer.

Errors

An INVALID_ENUM error is generated if *format* is DEPTH_STENCIL and *type* is not UNSIGNED_INT_24_8 or FLOAT_32_UNSIGNED_INT_24_8_REV.

An INVALID_OPERATION error is generated if *format* is DEPTH_COMPONENT and there is no depth buffer; if *format* is STENCIL_INDEX and there is no stencil buffer; or if *format* is DEPTH_STENCIL and either there is

no depth buffer, or there is no stencil buffer.

An INVALID_FRAMEBUFFER_OPERATION error is generated if the object bound to READ_FRAMEBUFFER_BINDING is not *framebuffer complete* (as defined in section 9.4.2).

An INVALID_OPERATION error is generated if *format* selects a color buffer while the read buffer is NONE, or if the GL is using a framebuffer object (the value of READ_FRAMEBUFFER_BINDING is non-zero) and the read buffer selects an attachment that has no image attached.

ReadPixels obtains values from the selected buffer from each pixel with lower left hand corner at $(x+i, y+j)$ for $0 \leq i < width$ and $0 \leq j < height$; this pixel is said to be the ith pixel in the jth row. If any of these pixels lies outside of the window allocated to the current GL context, or outside of the image attached to the currently bound read framebuffer object, then the values obtained for those pixels are undefined. When READ_FRAMEBUFFER_BINDING is zero, values are also undefined for individual pixels that are not owned by the current context. Otherwise, **ReadPixels** obtains values from the selected buffer, regardless of how those values were placed there.

If the GL is in RGBA mode, and *format* is one of LUMINANCE, LUMINANCE_ALPHA, RED, GREEN, BLUE, ALPHA, RG, RGB, RGBA, BGR, or BGRA, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location. If *format* is COLOR_INDEX and the GL is in RGBA mode then an INVALID_OPERATION error is generated. If the GL is in color index mode, and *format* is not DEPTH_COMPONENT, DEPTH_STENCIL, or STENCIL_INDEX, then the color index is obtained at each pixel location.

An INVALID_OPERATION error is generated if *format* is an integer format and the color buffer is not an integer format, or if the color buffer is an integer format and *format* is not an integer format.

When READ_FRAMEBUFFER_BINDING is non-zero, the red, green, blue, and alpha values are obtained by first reading the internal component values of the corresponding value in the image attached to the selected logical buffer. Internal components are converted to an RGBA color by taking each R, G, B, and A component present according to the base internal format of the buffer (as shown in table 8.18). If G, B, or A values are not present in the internal format, they are taken to be zero, zero, and one respectively.

18.2.4 Conversion of RGBA values

This step applies only if the GL is in RGBA mode, and then only if *format* is not STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL. The R, G, B, and A

values form a group of elements.

For a signed or unsigned normalized fixed-point color buffer, each element is converted to floating-point using equations 2.2 or 2.1, respectively. For an integer or floating-point color buffer, the elements are unmodified.

18.2.5 Conversion of Depth values

This step applies only if *format* is DEPTH_COMPONENT or DEPTH_STENCIL and the depth buffer uses a fixed-point representation. An element is taken to be a fixed-point value in $[0, 1]$ with m bits, where m is the number of bits in the depth buffer (see section 13.8.1). No conversion is necessary if the depth buffer uses a floating-point representation.

18.2.6 Pixel Transfer Operations

This step is actually the sequence of steps that was described separately in section 8.4.5. After the processing described in that section is completed, groups are processed as described in the following sections.

18.2.7 Conversion to L

This step applies only to RGBA component groups. If the *format* is either LUMINANCE or LUMINANCE_ALPHA, a value L is computed as

$$L = R + G + B$$

where R, G, and B are the values of the R, G, and B components. The single computed L component replaces the R, G, and B components in the group.

18.2.8 Final Conversion

Read color clamping is controlled by calling **ClampColor** (see section 18.1.1) with *target* set to CLAMP_READ_COLOR. If *clamp* is TRUE, read color clamping is enabled; if *clamp* is FALSE, read color clamping is disabled. If *clamp* is FIXED_ONLY, read color clamping is enabled if the selected read color buffer has fixed-point components.

For an integer RGBA color, each component is clamped to the representable range of *type*.

For a floating-point RGBA color, if *type* is FLOAT or HALF_FLOAT, each component is clamped to $[0, 1]$ if read color clamping is enabled. Then the appropriate conversion formula from table 18.2 is applied to the component.

If *type* is UNSIGNED_INT_10F_11F_11F_REV and *format* is RGB, each component is clamped to $[0, 1]$ if read color clamping is enabled. The returned data are then packed into a series of uint values. The red, green, and blue components are converted to unsigned 11-bit floating-point, unsigned 11-bit floating-point, and unsigned 10-bit floating-point as described in sections 2.3.4.3 and 2.3.4.4. The resulting red 11 bits, green 11 bits, and blue 10 bits are then packed as the 1st, 2nd, and 3rd components of the UNSIGNED_INT_10F_11F_11F_REV format as shown in table 8.13.

If *type* is UNSIGNED_INT_5_9_9_9_REV and *format* is RGB, each component is clamped to $[0, 1]$ if read color clamping is enabled. The returned data are then packed into a series of uint values. The red, green, and blue components are converted to red_s, $green_s$, $blue_s$, and exp_s integers as described in section 8.5.2 when *internalformat* is RGB9_E5. red_s, $green_s$, $blue_s$, and exp_s are then packed as the 1st, 2nd, 3rd, and 4th components of the UNSIGNED_INT_5_9_9_9_REV format as shown in table 8.13.

For other *type*s, and for a floating-point or unsigned normalized fixed-point color buffer, each component is clamped to $[0, 1]$ whether or not read color clamping is enabled. For a signed normalized fixed-point color buffer, each component is clamped to $[0, 1]$ if read color clamping is enabled, or if *type* represents unsigned integer components; otherwise *type* represents signed integer components, and each component is clamped to $[-1, 1]$. Following clamping, the appropriate conversion formula from table 18.2 is applied to the component[1].

For an index, if the *type* is not FLOAT or HALF_FLOAT, final conversion consists of masking the index with the value given in table 18.3. If the *type* is FLOAT or HALF_FLOAT, then the integer index is converted to a GL float or half data value.

18.2.9 Placement in Pixel Pack Buffer or Client Memory

If a pixel pack buffer is bound (as indicated by a non-zero value of PIXEL_PACK_BUFFER_BINDING), *data* is an offset into the pixel pack buffer and the pixels are packed into the buffer relative to this offset; otherwise, *data* is a pointer to a block of client memory and the pixels are packed into the client memory relative to the pointer.

An INVALID_OPERATION error is generated if a pixel pack buffer object is bound and packing the pixel data according to the pixel pack storage state would access memory beyond the size of the pixel pack buffer's memory size.

[1] OpenGL 4.2 changes the behavior of **ReadPixels** to allow readbacks from a signed normalized color buffer to a signed integer type without loss of information.

type Parameter	GL Data Type	Component Conversion Formula
UNSIGNED_BYTE	ubyte	Equation 2.3, $b = 8$
BYTE	byte	Equation 2.4, $b = 8$
UNSIGNED_SHORT	ushort	Equation 2.3, $b = 16$
SHORT	short	Equation 2.4, $b = 16$
UNSIGNED_INT	uint	Equation 2.3, $b = 32$
INT	int	Equation 2.4, $b = 32$
HALF_FLOAT	half	$c = f$
FLOAT	float	$c = f$
UNSIGNED_BYTE_3_3_2	ubyte	Equation 2.3, $b =$ bitfield width
UNSIGNED_BYTE_2_3_3_REV	ubyte	Equation 2.3, $b =$ bitfield width
UNSIGNED_SHORT_5_6_5	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_SHORT_5_6_5_REV	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_SHORT_4_4_4_4	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_SHORT_4_4_4_4_REV	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_SHORT_5_5_5_1	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_SHORT_1_5_5_5_REV	ushort	Equation 2.3, $b =$ bitfield width
UNSIGNED_INT_8_8_8_8	uint	Equation 2.3, $b =$ bitfield width
UNSIGNED_INT_8_8_8_8_REV	uint	Equation 2.3, $b =$ bitfield width
UNSIGNED_INT_10_10_10_2	uint	Equation 2.3, $b =$ bitfield width
UNSIGNED_INT_2_10_10_10_REV	uint	Equation 2.3, $b =$ bitfield width
UNSIGNED_INT_24_8	uint	Equation 2.3, $b =$ bitfield width
UNSIGNED_INT_10F_11F_11F_REV	uint	Special
UNSIGNED_INT_5_9_9_9_REV	uint	Special
FLOAT_32_UNSIGNED_INT_24_8_REV	float	$c = f$ (depth only)

Table 18.2: Reversed component conversions, used when component data are being returned to client memory. Color, normal, and depth components are converted from the internal floating-point representation (f) to a datum of the specified GL data type (c). All arithmetic is done in the internal floating-point format. These conversions apply to component data returned by GL query commands and to components of pixel data returned to client memory. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges (see table 2.2).

type Parameter	Index Mask
UNSIGNED_BYTE	$2^8 - 1$
BITMAP	1
BYTE	$2^7 - 1$
UNSIGNED_SHORT	$2^{16} - 1$
SHORT	$2^{15} - 1$
UNSIGNED_INT	$2^{32} - 1$
INT	$2^{31} - 1$
UNSIGNED_INT_24_8	$2^8 - 1$
FLOAT_32_UNSIGNED_INT_24_8_REV	$2^8 - 1$

Table 18.3: Index masks used by **ReadPixels**. Floating-point data are not masked.

An INVALID_OPERATION error is generated if a pixel pack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 8.7 for the *type* parameter.

Groups of elements are placed in memory just as they are taken from memory when transferring pixel rectangles to the GL. That is, the ith group of the jth row (corresponding to the ith pixel in the jth row) is placed in memory just where the ith group of the jth row would be taken from when transferring pixels. See **Unpacking** under section 8.4.4.1. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_. If the *format* is LUMINANCE, ALPHA, RED, GREEN, or BLUE, only the corresponding single element is written. Likewise if the *format* is LUMINANCE_ALPHA, RG, RGB, or BGR, only the corresponding two or three elements are written. Otherwise all the elements of each group are written.

18.3 Copying Pixels

Several commands copy pixel data between regions of the framebuffer (see sections 18.3.1 and 18.3.2), or between regions of textures and renderbuffers (see section 18.3.3). For all such commands, if the source and destination are identical or are different views of the same underlying texture image, and if the source and destination regions overlap in that framebuffer, renderbuffer, or texture image, pixel values resulting from the copy operation are undefined.

18.3.1 Copying Pixel Rectangles

The command

```
void CopyPixels( int x, int y, sizei width, sizei height,
    enum type );
```

transfers a rectangle of pixel values from one region of the read framebuffer to another in the draw framebuffer. Pixel copying is diagrammed in figure 18.2. *type* must be one of COLOR, STENCIL, DEPTH, or DEPTH_STENCIL, indicating that the values to be transferred are colors, stencil values, depth values, or depth/stencil values, respectively. The first four arguments have the same interpretation as the corresponding arguments to **ReadPixels**.

Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section 8.4.5, just as if **ReadPixels** were called with the corresponding arguments.

If the *type* is STENCIL or DEPTH, then it is as if the *format* for **ReadPixels** were STENCIL_INDEX or DEPTH_COMPONENT, respectively. If the *type* is DEPTH_STENCIL, then it is as if the *format* for **ReadPixels** were specified as described in table 18.4. If the *type* is COLOR, then if the GL is in RGBA mode, it is as if the *format* were RGBA, while if the GL is in color index mode, it is as if the *format* were COLOR_INDEX.

The groups of elements so obtained are then written to the framebuffer just as if **DrawPixels** had been given $width$ and $height$, beginning with final conversion of elements. The effective *format* is the same as that already described.

Undefined pixel values result from overlapping copies, as described in the introduction to section 18.3.

Errors

An INVALID_VALUE error is generated if *width* or *height* is negative.

The behavior of several GL operations is specified as if the arguments were passed to **CopyPixels**. These operations include **CopyTexImage***, **CopyTexSubImage***, **CopyColorTable**, **CopyColorSubTable**, and **CopyConvolutionFilter***. The following errors apply to all such commands as well as to **CopyPixels**:

An INVALID_FRAMEBUFFER_OPERATION error is generated if the object bound to READ_FRAMEBUFFER_BINDING (see section 9) is not framebuffer complete (as defined in section 9.4.2).

An INVALID_OPERATION error is generated if the object bound to

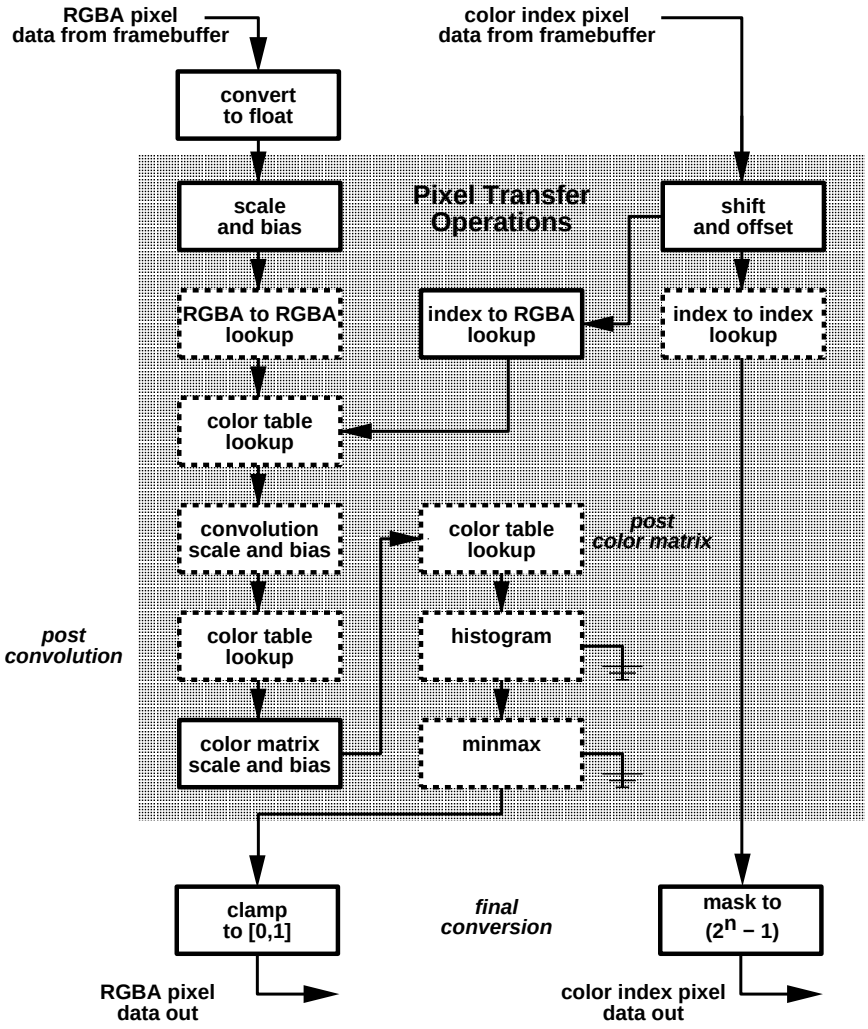


Figure 18.2. Operation of **CopyPixels**. Operations in dashed boxes may be enabled or disabled. Index-to-RGBA lookup is currently never performed. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

DEPTH_BITS	STENCIL_BITS	*format*
zero	zero	DEPTH_STENCIL
zero	non-zero	DEPTH_COMPONENT
non-zero	zero	STENCIL_INDEX
non-zero	non-zero	DEPTH_STENCIL

Table 18.4: Effective **ReadPixels** format for DEPTH_STENCIL **CopyPixels** operation.

READ_FRAMEBUFFER_BINDING is framebuffer complete and its effective value of SAMPLE_BUFFERS is one.

An INVALID_FRAMEBUFFER_OPERATION error is generated if the object bound to DRAW_FRAMEBUFFER_BINDING (see section 9) is not framebuffer complete.

An INVALID_OPERATION error is generated if the read buffer contains integer or unsigned integer components.

18.3.2 Blitting Pixel Rectangles

To transfer a rectangle of pixel values from one region of a source framebuffer to another region of a destination framebuffer, use the commands

```
void BlitFramebuffer( int srcX0, int srcY0, int srcX1,
    int srcY1, int dstX0, int dstY0, int dstX1, int dstY1,
    bitfield mask, enum filter );
void BlitNamedFramebuffer( uint readFramebuffer,
    uint drawFramebuffer, int srcX0, int srcY0, int srcX1,
    int srcY1, int dstX0, int dstY0, int dstX1, int dstY1,
    bitfield mask, enum filter );
```

For **BlitFramebuffer**, the source and destination framebuffers are those bound to READ_FRAMEBUFFER and DRAW_FRAMEBUFFER respectively. For **BlitNamedFramebuffer**, *readFramebuffer* and *drawFramebuffer* are the names of the source and destination framebuffer objects respectively.

If no framebuffer is bound to READ_FRAMEBUFFER or DRAW_FRAMEBUFFER (for **BlitFramebuffer**), or if *readFramebuffer* or *drawFramebuffer* is zero (for **BlitNamedFramebuffer**), then the default read or draw framebuffer is used as the corresponding source or destination framebuffer, respectively.

There are some important distinctions from **CopyPixels**, as described below.

mask is zero or the bitwise OR of one or more values indicating which buffers are to be copied. The values are `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, and `STENCIL_BUFFER_BIT`, which are described in section 17.4.3. The pixels corresponding to these buffers are copied from the source rectangle bounded by the locations $(srcX0, srcY0)$ and $(srcX1, srcY1)$ to the destination rectangle bounded by the locations $(dstX0, dstY0)$ and $(dstX1, dstY1)$.

Pixels have half-integer center coordinates. Only pixels whose centers lie within the destination rectangle are written by **BlitFramebuffer**. Linear filter sampling (see below) may result in pixels outside the source rectangle being read.

If *mask* is zero, no buffers are copied.

When the color buffer is transferred, values are taken from the read buffer of the read framebuffer and written to each of the draw buffers of the draw framebuffer, just as with **CopyPixels**.

The actual region taken from the read framebuffer is limited to the intersection of the source buffers being transferred, which may include the color buffer selected by the read buffer, the depth buffer, and/or the stencil buffer depending on *mask*. The actual region written to the draw framebuffer is limited to the intersection of the destination buffers being written, which may include multiple draw buffers, the depth buffer, and/or the stencil buffer depending on *mask*. Whether or not the source or destination regions are altered due to these limits, the scaling and offset applied to pixels being transferred is performed as though no such limits were present.

If the source and destination rectangle dimensions do not match, the source image is stretched to fit the destination rectangle. *filter* must be `LINEAR` or `NEAREST`, and specifies the method of interpolation to be applied if the image is stretched. `LINEAR` filtering is allowed only for the color buffer. If the source and destination dimensions are identical, no filtering is applied. If either the source or destination rectangle specifies a negative width or height ($X1 < X0$ or $Y1 < Y0$), the image is reversed in the corresponding direction. If both the source and destination rectangles specify a negative width or height for the same direction, no reversal is performed. If a linear filter is selected and the rules of `LINEAR` sampling would require sampling outside the bounds of a source buffer, it is as though `CLAMP_TO_EDGE` texture sampling were being performed. If a linear filter is selected and sampling would be required outside the bounds of the specified source region, but within the bounds of a source buffer, the implementation may choose to clamp while sampling or not.

If the source and destination buffers are identical, and the source and destination rectangles overlap, the result of the blit operation is undefined as described in the introduction to section 18.3.

When values are taken from the read buffer, if `FRAMEBUFFER_SRGB` is enabled

and the value of FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING for the framebuffer attachment corresponding to the read buffer is SRGB (see section 9.2.3), the red, green, and blue components are converted from the non-linear sRGB color space according to equation 8.18.

When values are written to the draw buffers, blit operations bypass most of the fragment pipeline. The only fragment operations which affect a blit are the pixel ownership test, the scissor test, and sRGB conversion (see section 17.3.7). Color, depth, and stencil masks (see section 17.4.2) are ignored.

If the read framebuffer is layered (see section 9.8), pixel values are read from layer zero. If the draw framebuffer is layered, pixel values are written to layer zero. If both read and draw framebuffers are layered, the blit operation is still performed only on layer zero.

If a buffer is specified in *mask* and does not exist in both the read and draw framebuffers, the corresponding bit is silently ignored.

If the color formats of the read and draw buffers do not match, and *mask* includes COLOR_BUFFER_BIT, pixel groups are converted to match the destination format as in **CopyPixels**. However, no pixel transfer operations are applied, and colors are clamped only if all draw color buffers have fixed-point components, as if CLAMP_FRAGMENT_COLOR were set to FIXED_ONLY. Format conversion is not supported for all data types, as described below.

If the read framebuffer is multisampled (its effective value of SAMPLE_BUFFERS is one) and the draw framebuffer is not (its value of SAMPLE_BUFFERS is zero), the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination. *filter* is ignored. If the source formats are integer types or stencil values, a single sample's value is selected for each pixel. If the source formats are floating-point or normalized types, the sample values for each pixel are resolved in an implementation-dependent manner. If the source formats are depth values, sample values are resolved in an implementation-dependent manner where the result will be between the minimum and maximum depth values in the pixel.

If the read framebuffer is not multisampled and the draw framebuffer is multisampled, the value of the source sample is replicated in each of the destination samples.

If both the read and draw framebuffers are multisampled, and their effective values of SAMPLES are identical, the samples are copied without modification (other than possible format conversion) from the read framebuffer to the draw framebuffer. Note that the samples in the draw buffer are not guaranteed to be at the same sample location as the read buffer, so rendering using this newly created buffer can potentially have geometry cracks or incorrect antialiasing. This may occur if the sizes of the framebuffers do not match or if the source and destination

rectangles are not defined with the same $(X0, Y0)$ and $(X1, Y1)$ bounds.

Errors

An `INVALID_OPERATION` error is generated by **BlitNamedFramebuffer** if *readFramebuffer* or *drawFramebuffer* is not zero or the name of an existing framebuffer object.

An `INVALID_VALUE` error is generated if *mask* contains any bits other than `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, or `STENCIL_BUFFER_BIT`.

An `INVALID_ENUM` error is generated if *filter* is not `LINEAR` or `NEAREST`.

An `INVALID_OPERATION` error is generated if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and filter is not `NEAREST`.

An `INVALID_FRAMEBUFFER_OPERATION` error is generated if either the read framebuffer or the draw framebuffer is not framebuffer complete (section 9.4.2).

An `INVALID_OPERATION` error is generated if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and the source and destination depth and stencil buffer formats do not match.

An `INVALID_OPERATION` error is generated if *filter* is `LINEAR` and the read buffer contains integer data.

An `INVALID_OPERATION` error is generated if either the read or draw framebuffer is multisampled, and the dimensions of the source and destination rectangles provided to **BlitFramebuffer** are not identical.

An `INVALID_OPERATION` error is generated if both the read and draw framebuffers are multisampled, and their effective values of `SAMPLES` are not identical.

An `INVALID_OPERATION` error is generated if format conversions are not supported, which occurs under any of the following conditions:

- The read buffer contains fixed-point or floating-point values and any draw buffer contains neither fixed-point nor floating-point values.
- The read buffer contains unsigned integer values and any draw buffer does not contain unsigned integer values.
- The read buffer contains signed integer values and any draw buffer does not contain signed integer values.

18.3.3 Copying Between Images

The command

```
void CopyImageSubData( uint srcName, enum srcTarget,
    int srcLevel, int srcX, int srcY, int srcZ,
    uint dstName, enum dstTarget, int dstLevel, int dstX,
    int dstY, int dstZ, sizei srcWidth, sizei srcHeight,
    sizei srcDepth );
```

may be used to copy a region of texel data between two image objects. An image object may be either a texture or a renderbuffer.

CopyImageSubData does not perform general-purpose conversions such as scaling, resizing, blending, color-space, or format conversions. It should be considered to operate in a manner similar to a CPU memcpy. **CopyImageSubData** can copy between images with different internal formats, provided the formats are compatible.

CopyImageSubData also allows copying between certain types of compressed and uncompressed internal formats as described in table 18.5. This copy does not perform on-the-fly compression or decompression. When copying from an uncompressed internal format to a compressed internal format, each texel of uncompressed data becomes a single block of compressed data. When copying from a compressed internal format to an uncompressed internal format, a block of compressed data becomes a single texel of uncompressed data. The texel size of the uncompressed format must be the same size as the block size of the compressed formats. Thus it is permitted to copy between a 128-bit uncompressed format and a compressed format which uses 8-bit 4×4 blocks, or between a 64-bit uncompressed format and a compressed format which uses 4-bit 4×4 blocks.

The source object is identified by *srcName* and *srcTarget*. Similarly the destination object is identified by *dstName* and *dstTarget*. The interpretation of the name depends on the value of the corresponding target parameter. If the target parameter is `RENDERBUFFER`, the name is interpreted as the name of a renderbuffer object. If the target parameter is a texture target, the name is interpreted as a texture object. All non-proxy texture targets are accepted, with the exception of `TEXTURE_BUFFER` and the cubemap face selectors described in table 8.27.

srcLevel and *dstLevel* identify the source and destination level-of-detail. For textures, this must be a valid level-of-detail in the texture object. For renderbuffers, this value must be zero.

srcX, *srcY*, and *srcZ* specify the lower left texel coordinates of a *srcWidth*-wide by *srcHeight*-high by *srcDepth*-deep rectangular subregion of the source texture image. Negative values of *srcX*, *srcY*, and *srcZ* correspond to the coordinates of

border texels, addressed as in figure 8.3. Similarly, *dstX*, *dstY* and *dstZ* specify the coordinates of a subregion of the destination texture image. The source and destination subregions must be contained entirely within the specified level of the corresponding image objects. The dimensions are always specified in texels, even for compressed texture formats. But it should be noted that if only one of the source and destination textures is compressed then the number of texels touched in the compressed image will be a factor of the block size larger than in the uncompressed image.

Slices of a one-dimensional array, two-dimensional array, cube map array, or three dimensional texture, or faces of a cube map texture are all compatible provided they share a compatible internal format, and multiple slices or faces may be copied between these objects with a single call by specifying the starting slice with *srcZ* and *dstZ*, and the number of slices to be copied with *srcDepth*. Cubemap textures always have six faces which are selected by a zero-based face index, according to the order specified in table 8.27.

For the purposes of **CopyImageSubData**, two internal formats are considered compatible if any of the following conditions are met:

- the formats are the same If the formats are the same but are a base internal format, the the implementation's effective internal format (see the end of section 8.5) for each image must be the same.
- the formats are considered compatible according to the compatibility rules used for texture views as defined in section 8.18. In particular, if both internal formats are listed in the same entry of table 8.30, they are considered compatible
- one format is compressed and the other is uncompressed and table 18.5 lists the two formats in the same row.

Undefined pixel values result from overlapping copies, as described in the introduction to section 18.3.

Errors

An `INVALID_OPERATION` error is generated if the texel size of the uncompressed image is not equal to the block size of the compressed image.

An `INVALID_ENUM` error is generated if either target is not `RENDERBUFFER` or a valid non-proxy texture target; is `TEXTURE_BUFFER` or one of the cubemap face selectors described in table 8.27; or if the target does not match the type of the object.

Texel / Block Size	Uncompressed internal format	Compressed internal format
128-bit	RGBA32UI, RGBA32I, RGBA32F	COMPRESSED_RG_RGTC2, COMPRESSED_SIGNED_RG_RGTC2, COMPRESSED_RGBA_BPTC_UNORM, COMPRESSED_SRGB_ALPHA_BPTC_UNORM, COMPRESSED_RGB_BPTC_SIGNED_FLOAT, COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT
64-bit	RGBA16F, RG32F, RGBA16UI, RG32UI, RGBA16I, RG32I, RGBA16, RGBA16_SNORM	COMPRESSED_RED_RGTC1, COMPRESSED_SIGNED_RED_RGTC1

Table 18.5: Compatible internal formats for copying between compressed and uncompressed internal formats with **CopyImageSubData**. Formats in the same row can be copied between each other.

An INVALID_OPERATION error is generated if either object is a texture and the texture is not complete (as defined in section 8.17, but ignoring format-based completeness rules), or if the source and destination internal formats are not compatible, or if the number of samples do not match.

An INVALID_VALUE error is generated if either name does not correspond to a valid renderbuffer or texture object according to the corresponding target parameter.

An INVALID_VALUE error is generated if *srcLevel* and *dstLevel* are not valid levels for the corresponding images.

An INVALID_VALUE error is generated if *srcWidth*, *srcHeight*, or *srcDepth* is negative.

An INVALID_VALUE error is generated if the dimensions of either subregion exceeds the boundaries of the corresponding image object, or if the image format is compressed and the dimensions of the subregion fail to meet the alignment constraints of the format.

An INVALID_OPERATION error is generated if the formats are not compatible.

18.4 Pixel Draw and Read State

The state required for pixel operations consists of the parameters that are set with **PixelStore**, **PixelTransfer**, and **PixelMap**. This state has been summarized in tables 8.1 and 18.1, 8.2, and 8.3, respectively Additional state includes the current raster position (section 14.7), and a three-valued integer controlling clamping during final conversion. The initial value of read color clamping is `FIXED_ONLY`. State set with **PixelStore** is GL client state.

Chapter 19

Compute Shaders

In addition to graphics-oriented shading operations such as vertex, tessellation, geometry, and fragment shading, generic computation may be performed by the GL through the use of compute shaders. The compute pipeline is a form of single-stage machine that runs generic shaders. Compute shaders are created as described in section 7.1 using a *type* parameter of `COMPUTE_SHADER`. They are attached to and used in program objects as described in section 7.3.

Compute workloads are formed from groups of work items called *workgroups* and processed by the executable code for a compute program. A workgroup is a collection of shader invocations that execute the same code, potentially in parallel. An invocation within a workgroup may share data with other members of the same workgroup through shared variables (see section 4.3.8("Shared Variables") of the OpenGL Shading Language Specification) and issue memory and control barriers to synchronize with other members of the same workgroup. One or more workgroups is launched by calling:

```
void DispatchCompute( uint num_groups_x,
    uint num_groups_y, uint num_groups_z );
```

Each workgroup is processed by the active program object for the compute shader stage. The active program for the compute shader stage will be determined in the same manner as the active program for other pipeline stages, as described in section 7.3. While the individual shader invocations within a workgroup are executed as a unit, workgroups are executed completely independently and in unspecified order.

num_groups_x, *num_groups_y* and *num_groups_z* specify the number of workgroups that will be dispatched in the X, Y and Z dimensions, respectively. The built-in vector variable `gl_NumWorkGroups` will be initialized with the contents

of the *num_groups_x*, *num_groups_y* and *num_groups_z* parameters. The maximum number of workgroups that may be dispatched at one time may be determined by calling **GetIntegeri_v** with *target* set to `MAX_COMPUTE_WORK_GROUP_COUNT` and *index* set to zero, one, or two, representing the X, Y, and Z dimensions respectively. If the workgroup count in any dimension is zero, no workgroups are dispatched.

The workgroup size in each dimension is specified at compile time using an input `layout` qualifier in one or more of the compute shaders attached to the program (see section 4.4.1.4("Compute Shader Inputs") of the OpenGL Shading Language Specification). After the program has been linked, the workgroup size of the program may be queried by calling **GetProgramiv** with *pname* `COMPUTE_WORK_GROUP_SIZE`. This will return an array of three integers containing the workgroup size of the compute program as specified by its input `layout` qualifier(s).

The maximum workgroup size may be determined by calling **GetIntegeri_v** with *target* set to `MAX_COMPUTE_WORK_GROUP_SIZE` and *index* set to 0, 1, or 2 to retrieve the maximum workgroup size in the X, Y and Z dimension, respectively. Furthermore, the maximum number of invocations in a single workgroup (i.e., the product of the three dimensions) may be determined by calling **GetIntegerv** with *pname* set to `MAX_COMPUTE_WORK_GROUP_INVOCATIONS`.

Errors

An `INVALID_OPERATION` error is generated if there is no active program for the compute shader stage.

An `INVALID_VALUE` error is generated if any of *num_groups_x*, *num_groups_y* and *num_groups_z* are greater than the maximum workgroup count for the corresponding dimension.

The command

`void` **DispatchComputeIndirect**(`intptr` *indirect*);

is equivalent to calling **DispatchCompute** with *num_groups_x*, *num_groups_y* and *num_groups_z* initialized with the three `uint` values contained in the buffer currently bound to the `DISPATCH_INDIRECT_BUFFER` binding at an offset, in basic machine units, specified in *indirect*. If any of *num_groups_x*, *num_groups_y* or *num_groups_z* is greater than the value of `MAX_COMPUTE_WORK_GROUP_COUNT` for the corresponding dimension then the results are undefined.

Errors

An `INVALID_OPERATION` error is generated if there is no active program

for the compute shader stage.

An INVALID_VALUE error is generated if *indirect* is negative or is not a multiple of the size, in basic machine units, of uint.

An INVALID_OPERATION error is generated if the command would source data beyond the end of the buffer object.

An INVALID_OPERATION error is generated if zero is bound to the DISPATCH_INDIRECT_BUFFER binding.

19.1 Compute Shader Variables

Compute shaders can access uniform variables belonging to the current program object. Limits on uniform storage and methods for manipulating uniforms are described in section 7.6.

There is a limit to the total size of all variables declared as shared in a single program object. This limit, expressed in units of basic machine units, may be queried as the value of MAX_COMPUTE_SHARED_MEMORY_SIZE. The total size required for a given program object is implementation-dependent. However, this total size may not exceed the largest block size that would be obtained if all active variables declared as shared were instead declared as arbitrarily ordered members of a shader storage block qualified with std430.

19.2 Compute Shader Queries

Compute shader queries use query objects to track the number of compute shader invocations.

When **BeginQuery** is called with a *target* of COMPUTE_SHADER_INVOCATIONS, the compute shader invocations count maintained by the GL is set to zero. When a compute shader invocations query is active, the counter is incremented every time the compute shader is invoked (see chapter 19).

Implementations are allowed to skip the execution of certain compute shader invocations, and to execute additional compute shader invocations due to implementation dependent reasons as long as the results of rendering otherwise remain unchanged.

Chapter 20

Debug Output

Application developers can obtain details about errors, undefined behavior, implementation-dependent performance warnings, or other useful hints from the GL in the form of *debug output*.

Debug output is communicated through a stream of *debug message*s that are generated as GL commands are executed. The application can choose to receive these messages either through a callback routine, or by querying for them from a message log.

Controls are provided for disabling messages that the application does not care about, and for inserting application-generated messages into the stream.

Different levels of debug output may be provided, depending on how the context was created. If the context is not a *debug context*[1] (e.g. if it was created without the `CONTEXT_FLAG_DEBUG_BIT` set in the `CONTEXT_FLAGS` state, as described in section 22.2), then the GL may optionally not generate any debug messages, but the commands described in this chapter will otherwise operate without error.

Debug output functionality is enabled or disabled by calling **Enable** or **Disable** with *target* `DEBUG_OUTPUT`. If the context is a *debug context* (if it was created with the `CONTEXT_FLAG_DEBUG_BIT` set in `CONTEXT_FLAGS`) then the initial value of `DEBUG_OUTPUT` is `TRUE`; otherwise the initial value is `FALSE`.

In a debug context, if `DEBUG_OUTPUT` is disabled the GL will not generate any debug output logs or callbacks. Enabling `DEBUG_OUTPUT` again will enable full debug output functionality.

In a non-debug context, if `DEBUG_OUTPUT` is later enabled, the level of debug output logging is defined by the GL implementation, which may have zero debug

[1] Debug contexts are specified at context creation time, using window system binding APIs such as those specified in the `GLX_ARB_create_context` and `WGL_ARB_create_context` extensions for GLX and WGL, respectively.

Debug Output Message Source	Messages Generated by
DEBUG_SOURCE_API	The GL
DEBUG_SOURCE_SHADER_COMPILER	The GLSL shader compiler or compilers for other extension-provided languages
DEBUG_SOURCE_WINDOW_SYSTEM	The window system, such as WGL or GLX
DEBUG_SOURCE_THIRD_PARTY	External debuggers or third-party middleware libraries
DEBUG_SOURCE_APPLICATION	The application
DEBUG_SOURCE_OTHER	Sources that do not fit to any of the ones listed above

Table 20.1: Sources of debug output messages. Each message must originate from a source listed in this table.

output.

Full debug output support is guaranteed only in a debug context.

20.1 Debug Messages

A debug message is uniquely identified by the source that generated it, a type within that source, and an unsigned integer ID identifying the message within that type. The message source is one of the symbolic constants listed in table 20.1. The message type is one of the symbolic constants listed in table 20.2.

Each message source and type pair contains its own namespace of messages with every message being associated with an ID. The assignment of IDs to messages within a namespace is implementation-dependent. There can potentially be overlap between the namespaces of two different pairs of source and type, so messages can only be uniquely distinguished from each other by the full combination of source, type and ID.

Each message is also assigned a severity level that roughly describes its importance across all sources and types along a single global axis. The severity of a message is one of the symbolic constants defined in table 20.3. Because messages can be disabled by their severity, the global volume of debug output can be limited.

Every message also has a null-terminated string representation that is used to describe the message. The contents of the string can change slightly between different instances of the same message (e.g. which parameter value caused a specific GL error to occur). The format of a message string is left as implementation-

Debug Output Message Type	Informs about
DEBUG_TYPE_ERROR	Events that generated an error
DEBUG_TYPE_DEPRECATED_BEHAVIOR	Behavior that has been marked for deprecation
DEBUG_TYPE_UNDEFINED_BEHAVIOR	Behavior that is undefined according to the specification
DEBUG_TYPE_PERFORMANCE	Implementation-dependent performance warnings
DEBUG_TYPE_PORTABILITY	Use of extensions or shaders in a way that is highly vendor-specific
DEBUG_TYPE_MARKER	Annotation of the command stream
DEBUG_TYPE_PUSH_GROUP	Entering a debug group
DEBUG_TYPE_POP_GROUP	Leaving a debug group
DEBUG_TYPE_OTHER	Types of events that do not fit any of the ones listed above

Table 20.2: Types of debug output messages. Each message is associated with one of these types that describes the nature of the message.

Severity Level Token	Suggested examples of messages
DEBUG_SEVERITY_HIGH	Any GL error; dangerous undefined behavior; any shader compiler and linker errors;
DEBUG_SEVERITY_MEDIUM	Severe performance warnings; GLSL or other shader compiler and linker warnings; use of currently deprecated behavior
DEBUG_SEVERITY_LOW	Performance warnings from redundant state changes; trivial undefined behavior
DEBUG_SEVERITY_NOTIFICATION	Any message which is not an error or performance concern

Table 20.3: Severity levels of messages. Each debug output message is associated with one of these severity levels.

dependent, although it should at least represent a concise description of the event that caused the message to be generated. Messages with different IDs should also have sufficiently distinguishable string representations to warrant their separation.

The lengths of all messages, including their null terminators, is guaranteed to be less than or equal to the value of the implementation-dependent constant MAX_DEBUG_MESSAGE_LENGTH.

Messages can be either enabled or disabled. Messages that are disabled will not be generated. All messages are initially enabled unless their assigned severity is DEBUG_SEVERITY_LOW. The enabled state of messages can be changed using the command **DebugMessageControl**.

20.2 Debug Message Callback

Applications can provide a callback function for receiving debug messages using the command

```
void DebugMessageCallback( DEBUGPROC callback, const
  void *userParam );
```

with *callback* storing the address of the callback function. *callback* must be a function whose prototype is of the form

```
void callback( enum source, enum type, uint id,
  enum severity, sizei length, const char *message,
  const void *userParam );
```

Additionally, *callback* must be declared with the same platform-dependent calling convention used in the definition of the type DEBUGPROC. Anything else will result in undefined behavior.

Only one debug callback can be specified for the current context, and further calls overwrite the previous callback. Specifying NULL as the value of *callback* clears the current callback and disables message output through callbacks. Applications can provide user-specified data through the pointer *userParam*. The context will store this pointer and will include it as one of the parameters in each call to the callback function.

If the application has specified a callback function for receiving debug output, the implementation will call that function whenever any enabled message is generated. The source, type, ID, and severity of the message are specified by the DEBUGPROC parameters *source*, *type*, *id*, and *severity*, respectively. The string representation of the message is stored in *message* and its length (excluding the

null-terminator) is stored in *length*. The parameter *userParam* is the user-specified parameter that was given when calling **DebugMessageCallback**.

Applications that specify a callback function must be aware of certain special conditions when executing code inside a callback when it is called by the GL, regardless of the debug source.

The memory for *message* is owned and managed by the GL, and should only be considered valid for the duration of the function call.

The behavior of calling any GL or window system function from within the callback function is undefined and may lead to program termination.

Care must also be taken in securing debug callbacks for use with asynchronous debug output by multi-threaded GL implementations. Section 20.8 describes this in further detail.

If `DEBUG_OUTPUT` is disabled, then the GL will not call the *callback* function.

20.3 Debug Message Log

If `DEBUG_CALLBACK_FUNCTION` is `NULL`, then debug messages are instead stored in an internal message log up to some maximum number of messages as defined by the value of `MAX_DEBUG_LOGGED_MESSAGES`.

Each context stores its own message log and will only store messages generated by commands operating in that context. If the message log fills up, then any subsequently generated messages will not be placed in the log until the message log is cleared, and will instead be discarded.

Applications can query the number of messages currently in the log by obtaining the value of `DEBUG_LOGGED_MESSAGES`, and the string length (including its null terminator) of the oldest message in the log through the value of `DEBUG_NEXT_LOGGED_MESSAGE_LENGTH`.

To fetch message data stored in the log, the command **GetDebugMessageLog** can be used.

If `DEBUG_CALLBACK_FUNCTION` is not `NULL`, no generated messages will be stored in the log but will instead be passed to the debug callback routine as described in section 20.2.

If `DEBUG_OUTPUT` is disabled, then no messages are added to the message log.

20.4 Controlling Debug Messages

Applications can control the volume of debug output in the active debug group (see section 20.6) by disabling specific groups of messages with the command

```
void DebugMessageControl( enum source, enum type,
    enum severity, sizei count, const uint *ids,
    boolean enabled );
```

If *enabled* is TRUE, the referenced subset of messages will be enabled. If FALSE, then those messages will be disabled.

This command can reference different subsets of messages by first considering the set of all messages, and filtering out messages based on the following ways:

- If *source*, *type*, or *severity* is DONT_CARE, then messages from all sources, of all types, or of all severities are referenced respectively.
- When values other than DONT_CARE are specified, all messages whose source, type, or severity match the specified *source*, *type*, or *severity* respectively will be referenced.
- If *count* is greater than zero, then *ids* is an array of *count* message IDs for the specified combination of *source* and *type*. In this case, *source* and *type* must not be DONT_CARE, and *severity* must be DONT_CARE,

  Unrecognized message IDs in *ids* are ignored. If *count* is zero, the value if *ids* is ignored.

Although messages are grouped into an implicit hierarchy by their sources and types, there is no explicit per-source, per-type or per-severity enabled state. Instead, the enabled state is stored individually for each message. There is no difference between disabling all messages from one source in a single call, and individually disabling all messages from that source using their types and IDs.

If DEBUG_OUTPUT is disabled, then it is as if messages of every *source*, *type*, or *severity* are disabled.

Errors

An INVALID_ENUM error is generated if any of *source*, *type*, and *severity* is neither DONT_CARE nor one of the symbols from, respectively, tables 20.1, 20.2, and 20.3.

An INVALID_VALUE error is generated if *count* is negative,

An INVALID_OPERATION error is generated if *count* is greater than zero and either *source* or *type* is DONT_CARE, or *severity* is not DONT_CARE.

20.5 Externally Generated Messages

To support applications and third-party libraries generating their own messages, such as ones containing timestamp information or signals about specific render system events, the following function can be called

```
void DebugMessageInsert( enum source, enum type, uint id,
    enum severity, int length, const char *buf );
```

The value of *id* specifies the ID for the message and *severity* indicates its severity level as defined by the caller. The string *buf* contains the string representation of the message. The parameter *length* contains the number of characters in *buf*. If *length* is negative, it is implied that *buf* contains a null terminated string.

Errors

If DEBUG_OUTPUT is disabled, then calls to **DebugMessageInsert** are discarded, but do not generate an error.

An INVALID_ENUM error is generated if *type* is not one of the values from table 20.2, or if *source* is not DEBUG_SOURCE_APPLICATION or DEBUG_SOURCE_THIRD_PARTY.

An INVALID_ENUM error is generated if *severity* is not one of the severity levels listed in table 20.3.

An INVALID_VALUE error is generated if the number of characters in *buf*, excluding the null terminator when *length* is negative, is not less than the value of MAX_DEBUG_MESSAGE_LENGTH.

20.6 Debug Groups

*Debug group*s provide a method for annotating a command stream with discrete groups of commands using a descriptive text. Debug output messages, either generated by the implementation or inserted by the application with **DebugMessageInsert** are written to the *active debug group* (the top of the debug group stack). Debug groups are strictly hierarchical. Their sequences may be nested within other debug groups but can not overlap. If no debug group has been pushed by the application then the active debug group is the default debug group.

The command

```
void PushDebugGroup( enum source, uint id, sizei length,
    const char *message );
```

pushes a debug group described by the string *message* into the command stream. The value of *id* specifies the ID of messages generated. The parameter *length* contains the number of characters in *message*. If *length* is negative, it is implied that *message* contains a null terminated string. The message has the specified *source* and *id*, *type* DEBUG_TYPE_PUSH_GROUP, and *severity* DEBUG_SEVERITY_NOTIFICATION. The GL will put a new debug group on top of the debug group stack which inherits control of the volume of debug output of the debug group previously residing on the top of the debug group stack. Because debug groups are strictly hierarchical, any additional control of the debug output volume will only apply within the active debug group and the debug groups pushed on top of the active debug group.

Errors

An INVALID_ENUM error is generated if the value of *source* is neither DEBUG_SOURCE_APPLICATION nor DEBUG_SOURCE_THIRD_PARTY.

An INVALID_VALUE error is generated if *length* is negative and the number of characters in *message*, excluding the null-terminator, is not less than the value of MAX_DEBUG_MESSAGE_LENGTH.

A STACK_OVERFLOW error is generated if **PushDebugGroup** is called and the stack contains the value of MAX_DEBUG_GROUP_STACK_DEPTH minus one elements.

The command

```
void PopDebugGroup( void );
```

pops the active debug group. After popping a debug group, the GL will also generate a debug output message describing its cause based on the *message* string, the *source*, and an *id* submitted to the associated **PushDebugGroup** command. DEBUG_TYPE_PUSH_GROUP and DEBUG_TYPE_POP_GROUP share a single namespace for message *id*. *severity* has the value DEBUG_SEVERITY_NOTIFICATION and *type* has the value DEBUG_TYPE_POP_GROUP. Popping a debug group restores the debug output volume control of the parent debug group.

Errors

A STACK_UNDERFLOW error is generated if **PopDebugGroup** is called and only the default debug group is on the stack.

Identifier	Object Type
BUFFER	buffer
DISPLAY_LIST	display list
FRAMEBUFFER	framebuffer
PROGRAM_PIPELINE	program pipeline
PROGRAM	program
QUERY	query
RENDERBUFFER	renderbuffer
SAMPLER	sampler
SHADER	shader
TEXTURE	texture
TRANSFORM_FEEDBACK	transform feedback
VERTEX_ARRAY	vertex array

Table 20.4: Object namespace identifiers and the corresponding object types.

20.7 Debug Labels

Debug labels provide a method for annotating any object (texture, buffer, shader, etc.) with a descriptive text label. These labels may then be used by the debug output (see chapter 20) or an external tool such as a debugger or profiler to describe labelled objects.

The command

```
void ObjectLabel( enum identifier, uint name, sizei length,
    const char *label );
```

labels the object identified by *name* and its namespace *identifier*. *identifier* must be one of the tokens in table 20.4, indicating the type of the object corresponding to *name*.

label contains a string used to label an object. *length* contains the number of characters in *label*. If *length* is negative, then *label* contains a null-terminated string. If *label* is NULL, any debug label is effectively removed from the object.

Errors

An INVALID_ENUM error is generated if *identifier* is not one of the object types listed in table 20.4.

An INVALID_VALUE error is generated if *name* is not the name of a valid object of the type specified by *identifier*.

An INVALID_VALUE error is generated if the number of characters in *label*, excluding the null terminator when *length* is negative, is not less than the value of MAX_LABEL_LENGTH.

The command

```
void ObjectPtrLabel( void *ptr, sizei length, const
   char *label );
```

labels the sync object identified by *ptr*. *length* and *label* match the corresponding arguments of **ObjectLabel**.

Errors

An INVALID_VALUE error is generated if *ptr* is not the name of a sync object.

An INVALID_VALUE error is generated if the number of characters in *label*, excluding the null terminator when *length* is negative, is not less than the value of MAX_LABEL_LENGTH.

A label is part of the state of the object to which it is associated. The initial state of an object's label is the empty string. Labels need not be unique.

20.8 Asynchronous and Synchronous Debug Output

The behavior of how and when the GL driver is allowed to generate debug messages, and subsequently either call back to the application or place the message in the debug message log, is affected by the state DEBUG_OUTPUT_SYNCHRONOUS. This state can be modified by the **Enable** and **Disable** commands. Its initial value is FALSE.

When DEBUG_OUTPUT_SYNCHRONOUS is disabled, the driver is optionally allowed to concurrently call the debug callback routine from potentially multiple threads, including threads that the context that generated the message is not currently bound to. The implementation may also call the callback routine asynchronously after the GL command that generated the message has already returned. The application is fully responsible for ensuring thread safety due to debug callbacks under these circumstances. In this situation the *userParam* value may be helpful in identifying which application thread's command originally generated the debug callback.

When DEBUG_OUTPUT_SYNCHRONOUS is enabled, the driver guarantees synchronous calls to the callback routine by the context. When synchronous callbacks are enabled, all calls to the callback routine will be made by the thread that owns the current context; all such calls will be made serially by the current context; and each call will be made before the GL command that generated the debug message is allowed to return.

When no callback is specified and DEBUG_OUTPUT_SYNCHRONOUS is disabled, the driver can still asynchronously place messages in the debug message log, even after the context thread has returned from the GL function that generated those messages. When DEBUG_OUTPUT_SYNCHRONOUS is enabled, the driver guarantees that all messages are added to the log before the GL function returns.

Enabling synchronous debug output greatly simplifies the responsibilities of the application for making its callback functions thread-safe, but may potentially result in drastically reduced driver performance.

The DEBUG_OUTPUT_SYNCHRONOUS only guarantees intra-context synchronization for the callbacks of messages generated by that context, and does not guarantee synchronization across multiple contexts. If multiple contexts are concurrently used by the application, it is allowed for those contexts to also concurrently call their designated callbacks, and the application is responsible for handling thread safety in that situation even if DEBUG_OUTPUT_SYNCHRONOUS is enabled in all contexts.

20.9 Debug Output Queries

Pointers set with debug output commands are queried with the generic **GetPointerv** command (see section 22.2). *pname*s DEBUG_CALLBACK_FUNCTION and DEBUG_CALLBACK_USER_PARAM respectively query the current callback function and the user parameter to that function set with **DebugMessageCallback**.

When no debug callback is set, debug messages are stored in a debug message log as described in section 20.3. Messages may be queried from the log by calling

```
uint GetDebugMessageLog( uint count, sizei bufSize,
    enum *sources, enum *types, uint *ids, enum *severities,
    sizei *lengths, char *messageLog );
```

GetDebugMessageLog fetches a maximum of *count* messages from the message log, and will return the number of messages successfully fetched.

Messages will be fetched from the log in order of oldest to newest. Those messages that were fetched will be removed from the log.

The sources, types, severities, IDs, and string lengths of fetched messages will be stored in the application-provided arrays *sources*, *types*, *severities*, *ids*, and *lengths*, respectively. The application is responsible for allocating enough space for each array to hold up to *count* elements. The string representations of all fetched messages are stored in the *messageLog* array. If multiple messages are fetched, their strings are concatenated into the same *messageLog* array and will be separated by single null terminators. The last string in the array will also be null-terminated. The maximum size of *messageLog*, including the space used by all null terminators, is given by *bufSize*.

If a message's string, including its null terminator, can not fully fit within the *messageLog* array's remaining space, then that message and any subsequent messages will not be fetched and will remain in the log. The string lengths stored in the array *lengths* include the space for the null terminator of each string.

Any or all of the arrays *sources*, *types*, *ids*, *severities*, *lengths* and *messageLog* can also be `NULL` pointers, which causes attributes for such arrays to be discarded when messages are fetched. However, those messages will still be removed from the log. Thus to simply delete up to *count* messages from the message log while ignoring their attributes, the application can call **GetDebugMessageLog** with `NULL` pointers for all attribute arrays.

If the context is not a debug context, then the GL can opt to never add messages to the message log, so that **GetDebugMessageLog** will always return zero.

Errors

An `INVALID_VALUE` error is generated if *bufSize* is negative and *messageLog* is not `NULL`.

The command

```
void GetObjectLabel( enum identifier, uint name,
    sizei bufSize, sizei *length, char *label );
```

returns in *label* the string labelling an object. *identifier* and *name* specify the namespace and name of the object, and match the corresponding arguments of **ObjectLabel** (see section 20.7).

label will be null-terminated. The actual number of characters written into *label*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, no length is returned. The maximum number of characters that may be written into *label*, including the null terminator, is specified by *bufSize*. If no debug label was specified for the object then *label* will contain a null-terminated empty string, and

zero will be returned in *length*. If *label* is `NULL` and *length* is non-`NULL` then no string will be returned and the length of the label will be returned in *length*.

Errors

An `INVALID_ENUM` error is generated is *identifier* is not one of the object types listed in table 20.4.

An `INVALID_VALUE` error is generated if *name* is not the name of a valid object of the type specified by *identifier*.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

The command

```
void GetObjectPtrLabel( void *ptr, sizei bufSize,
    size *length, char *label );
```

returns in *label* the string labelling the sync object identified by *ptr*. *bufSize*, *length*, and *label* match the corresponding arguments of **GetObjectLabel**.

Errors

An `INVALID_VALUE` error is generated if *ptr* is not the name of a sync object.

An `INVALID_VALUE` error is generated if *bufSize* is negative.

Chapter 21

Special Functions

This chapter describes additional functionality that does not fit easily into any of the preceding chapters, including evaluators used to model curves and surfaces (see section 21.1), selection used to locate rendered primitives on the screen (see section 21.2), feedback which returns GL results before rasterization (see section 21.3), display lists used to store a group of GL commands for later execution by the GL (see section 21.4), and hints influencing GL behavior (see section 21.5).

21.1 Evaluators

Evaluators provide a means to use a polynomial or rational polynomial mapping to produce vertex, normal, and texture coordinates, and colors. The values so produced are sent on to further stages of the GL as if they had been provided directly by the client. Transformations, lighting, primitive assembly, rasterization, and per-pixel operations are not affected by the use of evaluators.

Consider the R^k-valued polynomial $\mathbf{p}(u)$ defined by

$$\mathbf{p}(u) = \sum_{i=0}^{n} B_i^n(u)\mathbf{R}_i \tag{21.1}$$

with $\mathbf{R}_i \in R^k$ and

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i},$$

the ith Bernstein polynomial of degree n (recall that $0^0 \equiv 1$ and $\binom{n}{0} \equiv 1$). Each $\mathbf{R}_i$ is a *control point*. The relevant command is

```
void Map1{fd}( enum target, T u1, T u2, int stride,
    int order, const T *points );
```

target	k	Values
MAP1_VERTEX_3	3	x, y, z vertex coordinates
MAP1_VERTEX_4	4	x, y, z, w vertex coordinates
MAP1_INDEX	1	color index
MAP1_COLOR_4	4	R, G, B, A
MAP1_NORMAL	3	x, y, z normal coordinates
MAP1_TEXTURE_COORD_1	1	s texture coordinate
MAP1_TEXTURE_COORD_2	2	s, t texture coordinates
MAP1_TEXTURE_COORD_3	3	s, t, r texture coordinates
MAP1_TEXTURE_COORD_4	4	s, t, r, q texture coordinates

Table 21.1: Values specified by the *target* to **Map1**. Values are given in the order in which they are taken.

target is a symbolic constant indicating the range of the defined polynomial. Its possible values, along with the evaluations that each indicates, are given in table 21.1. *points* is a pointer to a set of *order* blocks of storage. Each block begins with k single-precision floating-point or double-precision floating-point values for **Map1f** and **Map2f**, respectively. The rest of the block may be filled with arbitrary data. Table 21.1 indicates how k depends on *target* and what the k values represent in each case.

stride is the number of single- or double-precision values (as appropriate) in each block of storage. The order of the polynomial, *order*, is also the number of blocks of storage containing control points. *order* is equal to $n + 1$.

u1 and *u2* give two floating-point values that define the endpoints of the preimage of the map. When a value u is presented for evaluation, the formula used is

$$\mathbf{p}'(u) = \mathbf{p}(\frac{u - u1}{u2 - u1}).$$

Errors

An INVALID_ENUM error is generated if *target* is not one of the values in table 21.1.

An INVALID_VALUE error is generated if *order* is less than one or greater than the value of MAX_EVAL_ORDER, if *stride* is less than k, or if $u1 = u2$.

An INVALID_OPERATION error is generated if the value of ACTIVE_TEXTURE is not TEXTURE0.

Map2 is analogous to **Map1**, except that it describes bivariate polynomials of the form

$$\mathbf{p}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)\mathbf{R}_{ij}.$$

The form of the **Map2** command is

```
void Map2{fd}( enum target, T u1, T u2, int ustride,
    int uorder, T v1, T v2, int vstride, int vorder, const
    T *points );
```

target is a range type selected from the same group as is used for **Map1**, except that the string MAP1 is replaced with MAP2. *points* is a pointer to $uorder \times vorder$ blocks of storage. The values comprising $\mathbf{R}_{ij}$ are located

$$ustride \times i + vstride \times j$$

values (either single- or double-precision floating-point, as appropriate) past the first value pointed to by *points*. *u1*, *u2*, *v1*, and *v2* define the pre-image rectangle of the map; a domain point (u, v) is evaluated as

$$\mathbf{p}'(u,v) = \mathbf{p}(\frac{u-u1}{u2-u1}, \frac{v-v1}{v2-v1}).$$

The evaluation of a defined map is enabled or disabled with **Enable** and **Disable** using the constant corresponding to the map as described above. The evaluator map generates only coordinates for texture unit TEXTURE0.

uorder is equal to $n+1$ and *vorder* is equal to $m+1$.

Errors

An INVALID_ENUM error is generated if *target* is not one of the values in table 21.1, with MAP1 replaced by MAP2.

An INVALID_VALUE error is generated if either *uorder* or *vorder* is less than one or greater than the value of MAX_EVAL_ORDER, if *ustride* or *vstride* is less than k, if $u1 = u2$, or if $v1 = v2$.

An INVALID_OPERATION error is generated if the value of ACTIVE_TEXTURE is not TEXTURE0.

Figure 21.1 describes map evaluation schematically; an evaluation of enabled maps is effected in one of two ways. The first way is to use

```
void EvalCoord1{fd}( T u );
```

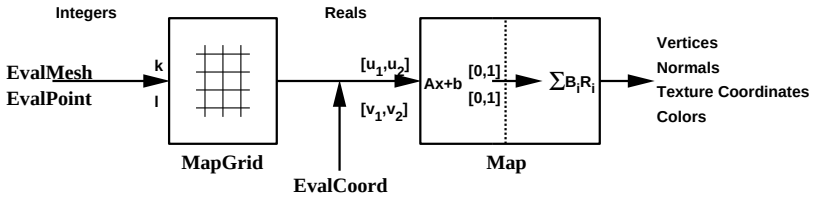


Figure 21.1. Map Evaluation.

```
void EvalCoord12{fd}v( const T *arg );
void EvalCoord2{fd}( T u, T v );
void EvalCoord12{fd}v( const T *arg );
```

EvalCoord1 causes evaluation of the enabled one-dimensional maps. The argument is the value or a pointer to the value of the domain coordinate u. **EvalCoord2** causes evaluation of the enabled two-dimensional maps. The arguments are the two values of, or a pointer to the two values of the domain coordinates u and v.

When one of the **EvalCoord** commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if a corresponding GL command were issued with the resulting coordinates, with one important difference. The difference is that when an evaluation is performed, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise, the current values are used). The order of the effective commands is immaterial, except that **Vertex** (for vertex coordinate evaluation) must be issued last. Use of evaluators has no effect on the current color, normal, or texture coordinates. If **ColorMaterial** is enabled, evaluated color values affect the result of the lighting equation as if the current color was being modified, but no change is made to the tracking lighting parameters or to the current color.

No command is effectively issued if the corresponding map (of the indicated dimension) is not enabled. If more than one evaluation is enabled for a particular dimension (e.g. `MAP1_TEXTURE_COORD_1` and `MAP1_TEXTURE_COORD_2`), then only the result of the evaluation of the map with the highest number of coordinates is used.

Finally, if either `MAP2_VERTEX_3` or `MAP2_VERTEX_4` is enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal

generation is enabled, then this computed normal is used as the normal associated with a generated vertex. Automatic normal generation is controlled by calling **Enable** and **Disable** with *target* `AUTO_NORMAL`. If automatic normal generation is disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used).

For `MAP2_VERTEX_3`, let $\mathbf{q} = \mathbf{p}$. For `MAP2_VERTEX_4`, let

$$\mathbf{q} = (\frac{x}{w} \; \frac{y}{w} \; \frac{z}{w})$$

, where

$$(x \; y \; z \; w) = \mathbf{p}$$

. Then let

$$\mathbf{m} = \frac{\partial \mathbf{q}}{\partial u} \times \frac{\partial \mathbf{q}}{\partial v}.$$

Then the generated analytic normal, $\mathbf{n}$, is given by $\mathbf{n} = \mathbf{m}$ if a vertex shader is active, or else by $\mathbf{n} = \frac{\mathbf{m}}{\|\mathbf{m}\|}$.

The second way to carry out evaluations is to use a set of commands that provide for efficient specification of a series of evenly spaced values to be mapped. This method proceeds in two steps. The first step is to define a grid in the domain.

For a one-dimensional map, the grid is defined with the command

`void` **MapGrid1**{**fd**}(`int` *n*, `T` *u1*, `T` *u2*);

u1 and *u2* describe an interval, while *n* describes the number of partitions of the interval.

Errors

An `INVALID_VALUE` error is generated if *n* is less than or equal to zero.

For a two-dimensional map, the grid is defined with the command

`void` **MapGrid2**{**fd**}(`int` *un*, `T` *u1*, `T` *u2*, `int` *vn*, `T` *v1*, `T` *v2*);

$(u1, v1)$ specifies one two-dimensional point and $(u2, v2)$ specifies another. *un* gives the number of partitions between *u1* and *u2*, and *vn* gives the number of partitions between *v1* and *v2*.

Errors

An INVALID_VALUE error is generated if either *un* or *vn* is less than or equal to zero.

Once a grid is defined, an evaluation on a rectangular subset of that grid may be carried out by calling

```
void EvalMesh1( enum mode, int i1, int i2 );
```

mode is either POINT or LINE. The effect is equivalent (assuming no errors are generated) to:

```
Begin(type);
for (i = i1; i <= i2; i++)
    EvalCoord1*(i × Δu + u1);
End();
```

where

$$\Delta u = \frac{u2 - u1}{n}$$

and **EvalCoord1f** or **EvalCoord1d** are substituted for **EvalCoord1*** as appropriate. If *mode* is POINT or LINE, then *type* is respectively POINTS or LINE_STRIP.

It must be the case that $0 \times \Delta u + u1 = u1$, $n \times \Delta u + u1 = u2$.

The corresponding commands for two-dimensional maps are

```
void EvalMesh2( enum mode, int i1, int i2, int j1,
    int j2 );
```

mode must be FILL, LINE, or POINT. When *mode* is FILL, then these commands are equivalent (assuming no errors are generated) to:

```
for (i = j1; i <= j2 - 1; i++) {
    Begin(QUAD_STRIP);
    for (j = i1; j <= i2; j++) {
        EvalCoord2(j × Δu + u1, i × Δv + v1);
        EvalCoord2(j × Δu + u1, (i + 1) × Δv + v1);
    }
    End();
}
```

where

$$\Delta u = \frac{u2 - u1}{un}$$

and

$$\Delta v = \frac{v2 - v1}{vn}.$$

If *mode* is `LINE`, then a call to **EvalMesh2** is equivalent (assuming no errors are generated) to:

```
for (i = j1; i <= j2; i++) {
    Begin(LINE_STRIP);
    for (j = i1; j <= i2; j += 1.0
        EvalCoord2(j × Δu + u1, i × Δv + v1);
    End();
}
for (i = i1; i <= i2; i+) {
    Begin(LINE_STRIP);
    for (j = j1; j <= j2; j++)
        EvalCoord2(i × Δu + u1, j × Δv + v1);
    End();
}
```

If *mode* is `POINT`, then a call to **EvalMesh2** is equivalent (assuming no errors are generated) to:

```
Begin(POINTS);
for (i = j1; i <= j2; i++)
    for (j = i1; j <= i2; j++)
        EvalCoord2(j × Δu + u1, i × Δv + v1);
End();
```

It must be the case that $0 \times \Delta u + u1 = u1$, $n \times \Delta u + u1 = u2$, $0 \times \Delta v + v1 = v1$, and $vn \times \Delta v + v1 = v2$.

An evaluation of a single point on the grid may also be carried out. The command

```
void EvalPoint1( int i );
```

is equivalent to

```
EvalCoord1(i × Δu + u1);
```

with Δu and $u1$ defined as for **EvalMesh1**.
The command

```
void EvalPoint2( int i, int j );
```

is equivalent to

EvalCoord2 $(i \times \Delta u + u1,\ \ j \times \Delta v + v1)$;

with Δu, $u1$, Δv, and $v1$ defined as for **EvalMesh2**.

The state required for evaluators potentially consists of 9 one-dimensional map specifications and 9 two-dimensional map specifications, as well as corresponding flags for each specification indicating which are enabled. Each map specification consists of one or two orders, an appropriately sized array of control points, and a set of two values (for a one-dimensional map) or four values (for a two-dimensional map) to describe the domain. The maximum possible order, for either u or v, is implementation-dependent (one maximum applies to both u and v), but must be at least 8. Each control point consists of between one and four floating-point values (depending on the type of the map). Initially, all maps have order one (making them constant maps). All vertex coordinate maps produce the coordinates $(0, 0, 0, 1)$ (or the appropriate subset); all normal coordinate maps produce $(0, 0, 1)$; RGBA maps produce $(1, 1, 1, 1)$; color index maps produce 1.0; and texture coordinate maps produce $(0, 0, 0, 1)$. In the initial state, all maps are disabled. A flag indicates whether or not automatic normal generation is enabled for two-dimensional maps. In the initial state, automatic normal generation is disabled. Also required are two floating-point values and an integer number of grid divisions for the one-dimensional grid specification and four floating-point values and two integer grid divisions for the two-dimensional grid specification. In the initial state, the bounds of the domain interval for 1-D is 0 and 1.0, respectively; for 2-D, they are $(0, 0)$ and $(1.0, 1.0)$, respectively. The number of grid divisions is one for 1-D and one in both directions for 2-D. If any evaluation command is issued when no vertex map is enabled for the map dimension being evaluated, nothing happens.

21.1.1 Evaluator Queries

The commands

```
void GetMap{ifd}v( enum target, enum query, T *data );
void GetnMap{ifd}v( enum map, enum value, sizei bufSize,
   T data );
```

return information about *query* for evaluator map *target* in *data*. *target* must be one of the map types described in section 21.1, and *query* must be one of ORDER, COEFF, or DOMAIN.

Errors

An INVALID_ENUM error is generated if *target* is not one of the valid *target* parameters to **Map1*** or **Map2***, or if *query* is not one of ORDER, COEFF, or DOMAIN.

An INVALID_VALUE error is generated if *bufSize* is negative.

An INVALID_OPERATION error is generated by **GetnMap*** if the buffer size required to store the requested data is greater than *bufSize*.

21.2 Selection

Selection is used to determine which primitives are drawn into some region of a window. The region is defined by the current model-view and perspective matrices.

Selection works by returning an array of integer-valued *names*. This array represents the current contents of the *name stack*. This stack is controlled with the commands

```
void InitNames( void );
void PopName( void );
void PushName( uint name );
void LoadName( uint name );
```

InitNames empties (clears) the name stack. **PopName** pops one name off the top of the name stack. **PushName** causes *name* to be pushed onto the name stack. **LoadName** replaces the value on the top of the stack with *name*.

The maximum allowable depth of the name stack is implementation-dependent, but must be at least 64.

Errors

An INVALID_OPERATION error is generated if **LoadName** is called and the stack is empty.

A STACK_UNDERFLOW error is generated if **PopName** is called and the stack is empty.

A STACK_OVERFLOW error is generated if **PushName** is called and the stack is full.

In selection mode, framebuffer updates as described in chapter 9 are not performed. The GL is placed in selection mode with

```
int RenderMode( enum mode );
```

mode is a symbolic constant: one of RENDER, SELECT, or FEEDBACK. RENDER is the default, corresponding to rendering as described until now. SELECT specifies selection mode, and FEEDBACK specifies feedback mode (described below). Use of any of the name stack manipulation commands while the GL is not in selection mode has no effect.

Errors

An INVALID_ENUM error is generated if *mode* is not RENDER, SELECT, or FEEDBACK.

An INVALID_OPERATION error is generated if **RenderMode** is called with *mode* SELECT before **SelectBuffer** (see below) has been called.

An INVALID_OPERATION error is generated if **RenderMode** is called with *mode* FEEDBACK before **FeedbackBuffer** (see section 21.3) has been called.

Selection is controlled using

```
void SelectBuffer( sizei n, uint *buffer );
```

buffer is a pointer to an array of unsigned integers (called the selection array) to be potentially filled with names, and *n* is an integer indicating the maximum number of values that can be stored in that array.

Errors

An INVALID_VALUE error is generated if *n* is negative.

An INVALID_OPERATION error is generated if the GL is in selection mode (see **RenderMode**).

In selection mode, if a point, line, or polygon that would otherwise be sent to the rasterizer intersects with the clip volume (see section 13.7), then this primitive causes a selection hit. Coordinates produced by a **RasterPos** command that intersect the clip volume also cause a selection hit, as do the coordinates from a **WindowPos** command.

In the case of polygons, no hit occurs if the polygon would have been culled, but selection is based on the polygon itself, regardless of the setting of **PolygonMode**. When in selection mode, whenever a name stack manipulation command is executed or **RenderMode** is called and there has been a hit since the last time the stack was manipulated or **RenderMode** was called, then a *hit record* is written into the selection array.

A hit record consists of the following items in order: a non-negative integer giving the number of elements on the name stack at the time of the hit, a minimum depth value, a maximum depth value, and the name stack with the bottommost element first. The minimum and maximum depth values are the minimum and maximum taken over all the window coordinate z values of each (post-clipping) vertex of each primitive that intersects the clipping volume since the last hit record was written. The minimum and maximum (each of which lies in the range $[0,1]$) are each multiplied by $2^{32}-1$ and rounded to the nearest unsigned integer to obtain the values that are placed in the hit record. No depth offset arithmetic (section 14.6.5) is performed on these values.

Hit records are placed in the selection array by maintaining a pointer into that array. When selection mode is entered, the pointer is initialized to the beginning of the array. Each time a hit record is copied, the pointer is updated to point at the array element after the one into which the topmost element of the name stack was stored. If copying the hit record into the selection array would cause the total number of values to exceed n, then as much of the record as fits in the array is written and an overflow flag is set.

Selection mode is exited by calling **RenderMode** with an argument value other than `SELECT`. When called while in selection mode, **RenderMode** returns the number of hit records copied into the selection array and resets the **SelectBuffer** pointer to its last specified value. Values are not guaranteed to be written into the selection array until **RenderMode** is called. If the selection array overflow flag was set, then **RenderMode** returns -1 and clears the overflow flag. The name stack is cleared and the stack pointer reset whenever **RenderMode** is called.

The state required for selection consists of the address of the selection array and its maximum size, the name stack and its associated pointer, a minimum and maximum depth value, and several flags. One flag indicates the current **RenderMode** value. In the initial state, the GL is in the `RENDER` mode. Another flag is used to indicate whether or not a hit has occurred since the last name stack manipulation. This flag is reset upon entering selection mode and whenever a name stack manipulation takes place. One final flag is required to indicate whether the maximum number of copied names would have been exceeded. This flag is reset upon entering selection mode. This flag, the address of the selection array, and its maximum size are GL client state.

21.3 Feedback

The GL is placed in feedback mode by calling **RenderMode** with *mode* `FEEDBACK`. When in feedback mode, framebuffer updates as described in chapter 9 are not performed. Instead, information about primitives that would have otherwise been rasterized is returned to the application via the *feedback buffer*.

Feedback is controlled using

`void` **FeedbackBuffer**(`sizei` *n*, `enum` *type*, `float` **buffer*);

buffer is a pointer to an array of floating-point values into which feedback information will be placed, and *n* is a number indicating the maximum number of values that can be written to that array. *type* is a symbolic constant describing the information to be fed back for each vertex (see figure 21.2).

Errors

An `INVALID_VALUE` error is generated if *n* is negative.

An `INVALID_ENUM` error is generated if *type* is not one of the tokens in table 21.2.

An `INVALID_OPERATION` error is generated if the GL is in feedback mode (see **RenderMode**).

While in feedback mode, each primitive that would be rasterized (or bitmap or call to **DrawPixels** or **CopyPixels**, if the raster position is valid) generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all). The first block of values generated after the GL enters feedback mode is placed at the beginning of the feedback array, with subsequent blocks following. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling (section 14.6.1) and **PolygonMode** interpretation of polygons (section 14.6.4) has taken place. It may also occur after polygons with more than three edges are broken up into triangles (if the GL implementation renders polygons by performing this decomposition). x, y, and z coordinates returned by feedback are window coordinates; if w is returned, it is in clip coordinates. No depth offset arithmetic (section 14.6.5) is performed on the z values. In the case of bitmaps and pixel rectangles, the coordinates returned are those of the current raster position.

Type	coordinates	color	texture	total values
2D	x, y	–	–	2
3D	x, y, z	–	–	3
3D_COLOR	x, y, z	k	–	$3 + k$
3D_COLOR_TEXTURE	x, y, z	k	4	$7 + k$
4D_COLOR_TEXTURE	x, y, z, w	k	4	$8 + k$

Table 21.2: Correspondence of feedback type to number of values per vertex. k is 1 in color index mode and 4 in RGBA mode.

The texture coordinates and colors returned are those resulting from the clipping operations described in section 13.7.1. Only coordinates for texture unit TEXTURE0 are returned even for implementations which support multiple texture units. The colors returned are the primary colors.

The ordering rules for GL command interpretation also apply in feedback mode. Each command must be fully interpreted and its effects on both GL state and the values to be written to the feedback buffer completed before a subsequent command may be executed.

Feedback mode is exited by calling **RenderMode** with an argument value other than FEEDBACK. When called while in feedback mode, **RenderMode** returns the number of values placed in the feedback array and resets the feedback array pointer to be *buffer*. The return value never exceeds the maximum number of values passed to **FeedbackBuffer**.

If writing a value to the feedback buffer would cause more values to be written than the specified maximum number of values, then the value is not written and an overflow flag is set. In this case, **RenderMode** returns -1 when it is called, after which the overflow flag is reset. While in feedback mode, values are not guaranteed to be written into the feedback buffer before **RenderMode** is called.

Figure 21.2 gives a grammar for the array produced by feedback. Each primitive is indicated with a unique identifying value followed by some number of vertices. A vertex is fed back as some number of floating-point values determined by the feedback *type*. Table 21.2 gives the correspondence between *type* and the number of values returned for each vertex.

The command

```
void PassThrough( float token );
```

may be used as a marker in feedback mode. *token* is returned as if it were a primitive; it is indicated with its own unique identifying value. The ordering of any

PassThrough commands with respect to primitive specification is maintained by feedback. **PassThrough** may not occur between **Begin** and **End**. It has no effect when the GL is not in feedback mode.

The state required for feedback is the pointer to the feedback array, the maximum number of values that may be placed there, and the feedback *type*. An overflow flag is required to indicate whether the maximum allowable number of feedback values has been written; initially this flag is cleared. These state variables are GL client state. Feedback also relies on the same mode flag as selection to indicate whether the GL is in feedback, selection, or normal rendering mode.

21.4 Display Lists

A display list is a group of GL commands and arguments that has been stored for subsequent execution. The GL may be instructed to process a particular display list (possibly repeatedly) by providing a number that uniquely specifies it. Doing so causes the commands within the list to be executed just as if they were given normally. The only exception pertains to commands that rely upon client state. When such a command is accumulated into the display list (that is, when issued, not when executed), the client state in effect at that time applies to the command. Only server state is affected when the command is executed. As always, pointers which are passed as arguments to commands are dereferenced when the command is issued. (Vertex array pointers are dereferenced when the commands **ArrayElement**, **DrawArrays**, **DrawElements**, or **DrawRangeElements** are accumulated into a display list.)

A display list is begun by calling

```
void NewList( uint n, enum mode );
```

n is a positive integer to which the display list that follows is assigned, and *mode* is a symbolic constant that controls the behavior of the GL during display list creation. If *mode* is `COMPILE`, then commands are not executed as they are placed in the display list. If *mode* is `COMPILE_AND_EXECUTE` then commands are executed as they are encountered, then placed in the display list.

Errors

An `INVALID_ENUM` error is generated if *mode* is not `COMPILE` or `COMPILE_AND_EXECUTE`.

An `INVALID_VALUE` error is generated if *n* is zero.

feedback-list:
    feedback-item feedback-list
    feedback-item

feedback-item:
    point
    line-segment
    polygon
    bitmap
    pixel-rectangle
    passthrough

point:
    `POINT_TOKEN` vertex

line-segment:
    `LINE_TOKEN` vertex vertex
    `LINE_RESET_TOKEN` vertex vertex

polygon:
    `POLYGON_TOKEN` n polygon-spec

polygon-spec:
    polygon-spec vertex
    vertex vertex vertex

bitmap:
    `BITMAP_TOKEN` vertex

pixel-rectangle:
    `DRAW_PIXEL_TOKEN` vertex
    `COPY_PIXEL_TOKEN` vertex

passthrough:
    `PASS_THROUGH_TOKEN` f

vertex:
`2D`:
    f f
`3D`:
    f f f
`3D_COLOR`:
    f f f color
`3D_COLOR_TEXTURE`:
    f f f color tex
`4D_COLOR_TEXTURE`:
    f f f f color tex

color:
    f f f f
    f

tex:
    f f f f

Figure 21.2: Feedback syntax. f is a floating-point number. n is a floating-point integer giving the number of vertices in a polygon. The symbols ending with `_TOKEN` are symbolic floating-point constants. The labels under the "vertex" rule show the different data returned for vertices depending on the feedback *type*. `LINE_TOKEN` and `LINE_RESET_TOKEN` are identical except that the latter is returned only when the line stipple is reset for that line segment.

An INVALID_OPERATION error is generated if **NewList** is called a second time before calling **EndList**.

After calling **NewList**, all subsequent GL commands are placed in the display list (in the order the commands are issued) until a call to

```
void EndList( void );
```

occurs, after which the GL returns to its normal command execution state. It is only when **EndList** occurs that the specified display list is actually associated with the index indicated with **NewList**.

Errors

An INVALID_OPERATION error is generated if **EndList** is called without a previous matching **NewList**.

An OUT_OF_MEMORY error is generated if **EndList** is called and the specified display list cannot be stored because insufficient memory is available. In this case no change is made to the previous contents of the display list, if any, and no other change is made to the GL state, except for the state changed by execution of GL commands when the display list mode is COMPILE_AND_EXECUTE.

Once defined, a display list is executed by calling

```
void CallList( uint n );
```

n gives the index of the display list to be called. This causes the commands saved in the display list to be executed, in order, just as if they were issued without using a display list.

Errors

An INVALID_VALUE error is generated if *n* is zero.

The command

```
void CallLists( sizei n, enum type, const void *lists );
```

provides an efficient means for executing a number of display lists. *n* is an integer indicating the number of display lists to be called, and *lists* is a pointer that points to an array of offsets. Each offset is constructed as determined by

lists as follows. First, *type* may be one of the constants BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, or FLOAT indicating that the array pointed to by *lists* is an array of bytes, unsigned bytes, shorts, unsigned shorts, integers, unsigned integers, or floats, respectively. In this case each offset is found by simply converting each array element to an integer (floating-point values are truncated to negative infinity). Further, *type* may be one of 2_BYTES, 3_BYTES, or 4_BYTES, indicating that the array contains sequences of 2, 3, or 4 unsigned bytes, in which case each integer offset is constructed according to the following algorithm:

$offset \leftarrow 0$
for $i = 1$ **to** b
    $offset \leftarrow offset$ shifted left 8 bits
    $offset \leftarrow offset + byte$
    advance to next *byte* in the array

b is 2, 3, or 4, as indicated by *type*. If $n = 0$, **CallLists** does nothing.

Each of the n constructed offsets is taken in order and added to a display list base to obtain a display list number. For each number, the indicated display list is executed.

Errors

An INVALID_VALUE error is generated if *n* is negative.

The display list base is set by calling

```
void ListBase( uint base );
```

to specify the offset.

Indicating a display list index that does not correspond to any display list has no effect. **CallList** or **CallLists** may appear inside a display list. (If the *mode* supplied to **NewList** is COMPILE_AND_EXECUTE, then the appropriate lists are executed, but the **CallList** or **CallLists**, rather than those lists' constituent commands, is placed in the list under construction.) To avoid the possibility of infinite recursion resulting from display lists calling one another, an implementation-dependent limit is placed on the nesting level of display lists during display list execution. This limit may be determined by calling **GetIntegerv** with *pname* MAX_LIST_NESTING, and must be at least 64.

Two commands are provided to manage display list indices.

```
uint GenLists( sizei s );
```

returns an integer n such that the indices $n, \ldots, n+s-1$ are previously unused (i.e. there are s previously unused display list indices starting at n). **GenLists** also has the effect of creating an empty display list for each of the indices $n, \ldots, n+s-1$, so that these indices all become used. **GenLists** returns zero if there is no group of s contiguous previously unused display list indices, or if $s = 0$.

Errors

An `INVALID_VALUE` error is generated if *s* is negative.

`boolean` **IsList**(`uint` *list*);

returns `TRUE` if *list* is the index of some display list.

A contiguous group of display lists may be deleted by calling

`void` **DeleteLists**(`uint` *list*, `sizei` *range*);

where *list* is the index of the first display list to be deleted and *range* is the number of display lists to be deleted. All information about the display lists is lost, and the indices become unused. Indices to which no display list corresponds are ignored. If *range* is zero, nothing happens.

Errors

An `INVALID_VALUE` error is generated if *range* is negative.

21.4.1 Commands Not Usable In Display Lists

Certain commands, when called while compiling a display list, are not compiled into the display list but are executed immediately. These commands fall in several categories including

Display lists: **GenLists** and **DeleteLists**.

Render modes: **FeedbackBuffer**, **SelectBuffer**, and **RenderMode**.

Vertex arrays: **ClientActiveTexture**, **ColorPointer**, **EdgeFlagPointer**, **FogCoordPointer**, **IndexPointer**, **InterleavedArrays**, **NormalPointer**, **SecondaryColorPointer**, **TexCoordPointer**, **VertexAttribPointer**, **VertexAttribIPointer**, **VertexAttribLPointer**, **VertexPointer**, **PrimitiveRestartIndex**, **GenVertexArrays**, **DeleteVertexArrays**, and **BindVertexArray**.

Client state: **EnableClientState**, **DisableClientState**, **EnableVertexAttribArray**, **DisableVertexAttribArray**, **PushClientAttrib**, and **PopClientAttrib**.

Pixels and textures: **PixelStore**, **ReadPixels**, **GenTextures**, **DeleteTextures**, **AreTexturesResident**, **TexBuffer**, **TexStorage1D**, **TexStorage2D**, **TexStorage3D**, and **GenerateMipmap**.

Occlusion queries: **GenQueries** and **DeleteQueries**.

Buffer objects: **GenBuffers**, **DeleteBuffers**, **BindBuffer**, **BindBufferRange**, **BindBufferBase**, **TransformFeedbackVaryings**, **MemoryBarrier**, **BufferData**, **BufferSubData**, **MapBuffer**, **MapBufferRange**, **FlushMappedBufferRange**, and **UnmapBuffer**.

Sampler objects: **GenSamplers** and **DeleteSamplers**.

Transform feedback objects: **GenTransformFeedbacks** and **DeleteTransformFeedbacks**.

Framebuffer and renderbuffer objects: **GenFramebuffers**, **BindFramebuffer**, **DeleteFramebuffers**, **CheckFramebufferStatus**, **GenRenderbuffers**, **BindRenderbuffer**, **DeleteRenderbuffers**, **RenderbufferStorage**, **RenderbufferStorageMultisample**, **FramebufferTexture**, **FramebufferTexture1D**, **FramebufferTexture2D**, **FramebufferTexture3D**, **FramebufferTextureLayer**, **FramebufferRenderbuffer**, and **BlitFramebuffer**.

Program and shader objects: **CreateProgram**, **CreateShader**, **CreateShaderProgram**, **GenProgramPipelines**, **BindProgramPipelines DeleteProgramPipelines**, **DeleteProgram**, **DeleteShader**, **AttachShader**, **DetachShader**, **BindAttribLocation**, **BindFragDataLocation**, **CompileShader**, **ShaderSource**, **LinkProgram**, and **ValidateProgram**.

GL command stream management: **ClientWaitSync**, **FenceSync**, **Finish**, and **Flush**.

Debug output: **DebugMessageControl**, **DebugMessageInsert**, **DebugMessageCallback**, and **GetDebugMessageLog**.

Multi-object binds: **BindBuffersBase**, **BindBuffersRange**, **BindTextures**, **BindSamplers**, **BindImageTextures**, and **BindVertexBuffers**.

Other queries: All query commands whose names begin with **Get** and **Is**.

Errors

An `INVALID_OPERATION` error is generated if **DispatchComputeIndirect** or any of the drawing commands ***DrawArraysIndirect**, ***DrawElementsIndirect**, **DrawArraysInstanced*** or **DrawElementsInstanced*** (see section 10.4) are called during display list compilation.

Commands that source data from buffer objects dereference the buffer object data in question at display list compile time, rather than encoding the buffer ID and buffer offset into the display list. Only GL commands that are executed immedi-

ately, rather than being compiled into a display list, are permitted to use a buffer object as a data sink.

TexImage3D, **TexImage2D**, **TexImage1D**, **Histogram**, and **ColorTable** are executed immediately when called with the corresponding proxy arguments PROXY_TEXTURE_3D, PROXY_TEXTURE_2D_ARRAY, or PROXY_TEXTURE_CUBE_MAP_ARRAY; PROXY_TEXTURE_2D, PROXY_TEXTURE_1D_ARRAY, or PROXY_TEXTURE_CUBE_MAP; PROXY_TEXTURE_1D; PROXY_HISTOGRAM; and PROXY_COLOR_TABLE, PROXY_POST_CONVOLUTION_COLOR_TABLE, or PROXY_POST_COLOR_MATRIX_COLOR_TABLE.

When a program object is in use, a display list may be executed whose vertex attribute calls do not match up exactly with what is expected by the vertex shader contained in that program object. Handling of this mismatch is described in section 11.1.1.

Display lists require one bit of state to indicate whether a GL command should be executed immediately or placed in a display list. In the initial state, commands are executed immediately. If the bit indicates display list creation, an index is required to indicate the current display list being defined. Another bit indicates, during display list creation, whether or not commands should be executed as they are compiled into the display list. One integer is required for the current **ListBase** setting; its initial value is zero. Finally, state must be maintained to indicate which integers are currently in use as display list indices. In the initial state, no indices are in use.

21.5 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

```
void Hint( enum target, enum hint );
```

target is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. The possible *target*s are described in table 21.3. For each *target*, *hint* must be one of FASTEST, indicating that the most efficient option should be chosen; NICEST, indicating that the highest quality option should be chosen; and DONT_CARE, indicating no preference in the matter.

For the texture compression hint, a *hint* of FASTEST indicates that texture images should be compressed as quickly as possible, while NICEST indicates that the texture images should be compressed with as little image degradation as possible. FASTEST should be used for one-time texture compression, and NICEST should be

Target	Hint description
PERSPECTIVE_CORRECTION_HINT	Quality of parameter interpolation
POINT_SMOOTH_HINT	Point sampling quality
LINE_SMOOTH_HINT	Line sampling quality
POLYGON_SMOOTH_HINT	Polygon sampling quality
FOG_HINT	Fog quality (calculated per-pixel or per-vertex)
GENERATE_MIPMAP_HINT	Quality and performance of automatic mipmap level generation
TEXTURE_COMPRESSION_HINT	Quality and performance of texture image compression
FRAGMENT_SHADER_DERIVATIVE_HINT	Derivative accuracy for fragment processing built-in functions dFdx, dFdy and fwidth

Table 21.3: Hint targets and descriptions.

used if the compression results are to be retrieved by **GetCompressedTexImage** (section 8.11) for reuse.

The interpretation of hints is implementation-dependent. An implementation may ignore them entirely.

The initial value of all hints is DONT_CARE.

Errors

An INVALID_ENUM error is generated if *target* is not one of the values in table 21.3.

An INVALID_ENUM error is generated if *hint* is not FASTEST, NICEST, or DONT_CARE.

21.6 Saving and Restoring State

Besides providing a means to obtain the values of state variables, the GL also provides a means to save and restore groups of state variables. The **PushAttrib**, **PushClientAttrib**, **PopAttrib** and **PopClientAttrib** commands are used for this purpose. The commands

```
void PushAttrib( bitfield mask );
```

```
void PushClientAttrib( bitfield mask );
```

take a bitwise OR of symbolic constants indicating which groups of state variables to push onto an attribute stack. **PushAttrib** uses a server attribute stack while **PushClientAttrib** uses a client attribute stack. Each constant refers to a group of state variables. The classification of each variable into a group is indicated in the following tables of state variables.

Bits set in *mask* that do not correspond to an attribute group are ignored. The special *mask* values ALL_ATTRIB_BITS and CLIENT_ALL_ATTRIB_BITS may be used to push all stackable server and client state, respectively.

Errors

A STACK_OVERFLOW error is generated if **PushAttrib** is called and the attribute stack depth is equal to the value of MAX_ATTRIB_STACK_DEPTH.

A STACK_OVERFLOW error is generated if **PushClientAttrib** is called and the client attribute stack depth is equal to the value of MAX_CLIENT_ATTRIB_STACK_DEPTH.

The commands

```
void PopAttrib( void );
void PopClientAttrib( void );
```

reset the values of those state variables that were saved with the last corresponding **PushAttrib** or **PopClientAttrib**. Those not saved remain unchanged.

Errors

A STACK_UNDERFLOW error is generated if **PopAttrib** is called and the attribute stack depth is zero.

A STACK_UNDERFLOW error is generated if **PopClientAttrib** is called and the client attribute stack depth is zero.

Table 21.4 shows the attribute groups with their corresponding symbolic constant names and stacks.

When **PushAttrib** is called with TEXTURE_BIT set, the priorities, border colors, filter modes, wrap modes, and other state of the currently bound texture objects (see tables 23.19- 23.20), as well as the current texture bindings and enables, are pushed onto the attribute stack. (Unbound texture objects are not pushed or restored.) When an attribute set that includes texture information is popped, the

Stack	Attribute	Constant
server	accum-buffer	ACCUM_BUFFER_BIT
server	color-buffer	COLOR_BUFFER_BIT
server	current	CURRENT_BIT
server	depth-buffer	DEPTH_BUFFER_BIT
server	enable	ENABLE_BIT
server	eval	EVAL_BIT
server	fog	FOG_BIT
server	hint	HINT_BIT
server	lighting	LIGHTING_BIT
server	line	LINE_BIT
server	list	LIST_BIT
server	multisample	MULTISAMPLE_BIT
server	pixel	PIXEL_MODE_BIT
server	point	POINT_BIT
server	polygon	POLYGON_BIT
server	polygon-stipple	POLYGON_STIPPLE_BIT
server	scissor	SCISSOR_BIT
server	stencil-buffer	STENCIL_BUFFER_BIT
server	texture	TEXTURE_BIT
server	transform	TRANSFORM_BIT
server	viewport	VIEWPORT_BIT
server		ALL_ATTRIB_BITS
client	vertex-array	CLIENT_VERTEX_ARRAY_BIT
client	pixel-store	CLIENT_PIXEL_STORE_BIT
client	select	can't be pushed or popped
client	feedback	can't be pushed or popped
client		CLIENT_ALL_ATTRIB_BITS

Table 21.4: Attribute groups

bindings and enables are first restored to their pushed values, then the bound texture object's parameters are restored to their pushed values. The bindings and state for sampler objects are not pushed or popped.

Operations on attribute groups push or pop texture state within that group for all texture units. When state for a group is pushed, all state corresponding to `TEXTURE0` is pushed first, followed by state corresponding to `TEXTURE1`, and so on up to and including the state corresponding to `TEXTURE`k where $k + 1$ is the value of `MAX_TEXTURE_UNITS`. When state for a group is popped, texture state is restored in the opposite order that it was pushed, starting with state corresponding to `TEXTURE`k and ending with `TEXTURE0`. Identical rules are observed for client texture state push and pop operations. Matrix stacks are never pushed or popped with **PushAttrib**, **PushClientAttrib**, **PopAttrib**, or **PopClientAttrib**.

The depth of each attribute stack is implementation-dependent but must be at least 16. The state required for each attribute stack is potentially 16 copies of each state variable, 16 masks indicating which groups of variables are stored in each stack entry, and an attribute stack pointer. In the initial state, both attribute stacks are empty.

Chapter 22

Context State Queries

The state required to describe the GL machine is enumerated in chapter 23, and is set using commands described in previous chapters.

State that is part of GL objects can usually be queried using commands described together with the commands to set that state. Such commands operate either directly on a named object, or indirectly through a binding in the GL context (such as a currently bound framebuffer object).

The commands in this chapter describe queries for state directly associated with the context, rather than with an object. Data conversions may be done when querying context state, as described in section 2.2.2.

22.1 Simple Queries

Much of the GL state is completely identified by symbolic constants. The values of these state variables can be obtained using a set of **Get** commands.

Valid values of the symbolic constants allowed as parameter names to the various queries in this section are not summarized here, because there are many allowed parameters. Instead they are described elsewhere in the Specification together with the commands such state is relevant to, as well as in the state tables in chapter 23.

There are five commands for obtaining simple state variables:

```
void GetBooleanv( enum pname, boolean *data );
void GetIntegerv( enum pname, int *data );
void GetInteger64v( enum pname, int64 *data );
void GetFloatv( enum pname, float *data );
void GetDoublev( enum pname, double *data );
```

The commands obtain boolean, integer, 64-bit integer, floating-point, or double-precision state variables. *pname* is a symbolic constant indicating the state variable to return. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data.

Errors

An `INVALID_ENUM` error is generated if *pname* is not state queriable with these commands.

Indexed simple state variables are queried with the commands

```
void GetBooleani_v( enum target, uint index,
   boolean *data );
void GetIntegeri_v( enum target, uint index, int *data );
void GetFloati_v( enum target, uint index, float *data );
void GetDoublei_v( enum target, uint index, double *data );
void GetInteger64i_v( enum target, uint index,
   int64 *data );
```

target is the name of the indexed state and *index* is the index of the particular element being queried. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data.

Errors

An `INVALID_ENUM` error is generated if *target* is not indexed state queriable with these commands.

An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

Finally,

```
boolean IsEnabled( enum cap );
```

can be used to determine if *cap* is currently enabled (as with **Enable**) or disabled.

Errors

An `INVALID_ENUM` error is generated if *cap* is not enable state queriable with **IsEnabled**.

`boolean` **IsEnabledi**(`enum` *target*, `uint` *index*);

can be used to determine if the indexed state corresponding to *target* and *index* is enabled or disabled.

Errors

An `INVALID_ENUM` error is generated if *target* is not indexed enable state queriable with **IsEnabledi**.

An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

22.2 Pointer, String, and Related Context Queries

Pointers in the current GL context are queried with the command

`void` **GetPointerv**(`enum` *pname*, `void` **params*);

pname is a symbolic constant indicating the pointer to return. *params* is a pointer to a variable in which to place the single returned pointer value.

*pname*s of `DEBUG_CALLBACK_FUNCTION` and `DEBUG_CALLBACK_USER_PARAM`, return debug output state as described in section 20.9. *pname*s `SELECTION_BUFFER_POINTER` and `FEEDBACK_BUFFER_POINTER`, return the pointers set with **SelectBuffer** and **FeedbackBuffer**, respectively. *pname*s `VERTEX_ARRAY_POINTER`, `NORMAL_ARRAY_POINTER`, `COLOR_ARRAY_POINTER`, `SECONDARY_COLOR_ARRAY_POINTER`, `INDEX_ARRAY_POINTER`, `TEXTURE_COORD_ARRAY_POINTER`, `FOG_COORD_ARRAY_POINTER`, are `EDGE_FLAG_ARRAY_POINTER`, respectively return the corresponding value stored in the currently bound vertex array object.

Errors

An `INVALID_ENUM` error is generated if *pname* is not one of the names described above.

String queries return pointers to UTF-8 encoded, null-terminated static strings describing properties of the current GL context [1]. The command

[1] Applications making copies of these static strings should never use a fixed-length buffer, because the strings may grow unpredictably between releases, resulting in buffer overflow when copying. This is particularly true of the `EXTENSIONS` string, which has become extremely long in some GL implementations.

```
ubyte *GetString( enum name );
```

accepts *name* values of RENDERER, VENDOR, EXTENSIONS, VERSION, and SHADING_LANGUAGE_VERSION. The format of the RENDERER and VENDOR strings is implementation-dependent. The EXTENSIONS string contains a space separated list of extension names (the extension names themselves do not contain any spaces). The VERSION and SHADING_LANGUAGE_VERSION strings are laid out as follows:

<version number><space><vendor-specific information>

The version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The *minor_number* for SHADING_LANGUAGE_VERSION is always two digits, matching the OpenGL Shading Language Specification release number. For example, this query might return the string "4.20" while the corresponding VERSION query returns "4.2". The *release_number* and vendor specific information are optional. However, if present, then they pertain to the server and their format and contents are implementation-dependent.

GetString returns the version number (in the VERSION string) and the extension names (in the EXTENSIONS string) that can be supported by the current GL context. Thus, if the client and server support different versions and/or extensions, a compatible version and list of extensions is returned.

Errors

An INVALID_ENUM error is generated if *name* is not RENDERER, VENDOR, EXTENSIONS, VERSION, or SHADING_LANGUAGE_VERSION.

The context version may also be queried by calling **GetIntegerv** with *pname* MAJOR_VERSION and MINOR_VERSION, which respectively return the same values as *major_number* and *minor_number* in the VERSION string.

The profile implemented by the context may be queried by calling **GetIntegerv** with *value* CONTEXT_PROFILE_MASK, which returns a mask containing one of the bits in table 22.1, corresponding to the API profile implemented by the context (see appendix E.1).

Flags defining additional properties of the context may be queried by calling **GetIntegerv** with *pname* CONTEXT_FLAGS.

If CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT is set in CONTEXT_FLAGS, then the context is a forward-compatible context as defined in appendix E, and the deprecated features described in that appendix are not supported; otherwise the

Value	OpenGL Profile
CONTEXT_CORE_PROFILE_BIT	Core
CONTEXT_COMPATIBILITY_PROFILE_BIT	Compatibility

Table 22.1: Context profile bits returned by the CONTEXT_PROFILE_MASK query.

context is a full context, and all features described in the specification are supported.

If CONTEXT_FLAG_DEBUG_BIT is set in CONTEXT_FLAGS, then the context is a *debug context*, enabling full support for debug output as described in chapter 20.

If CONTEXT_FLAG_ROBUST_ACCESS_BIT is set in CONTEXT_FLAGS, then robust buffer access will be enabled for drawing commands using vertex arrays, as described in section 10.3.7.

If CONTEXT_FLAG_NO_ERROR_BIT is set in CONTEXT_FLAGS, then no error behavior is enabled, as described in section 2.3.1.1.

The behavior of the context when it is made no longer current (released) by the platform binding may be queried by calling **GetIntegerv** with *pname* CONTEXT_RELEASE_BEHAVIOR. If the behavior is CONTEXT_RELEASE_BEHAVIOR_FLUSH, any pending commands on the context will be flushed. If the behavior is NONE, pending commands are not flushed.

The default value is CONTEXT_RELEASE_BEHAVIOR_FLUSH, and may in some cases be changed using platform-specific context creation extensions.

Indexed strings are queried with the command

```
ubyte *GetStringi( enum name, uint index );
```

name is the name of the indexed state and *index* is the index of the particular element being queried.

If *name* is EXTENSIONS, the extension name corresponding to the *index*th supported extension will be returned. *index* may range from zero to the value of NUM_EXTENSIONS minus one. All extension names, and only the extension names returned in **GetString**(EXTENSIONS) will be returned as individual names, but there is no defined relationship between the order in which names appear in the non-indexed string and the order in which they appear in the indexed query. There is no defined relationship between any particular extension name and the *index* values; an extension name may correspond to a different *index* in different GL contexts and/or implementations.

If *name* is SHADING_LANGUAGE_VERSION, a version string for one of the supported versions of the OpenGL Shading Language and OpenGL ES Shading Lan-

guage is returned. *index* may range from zero to the value of NUM_SHADING_LANGUAGE_VERSIONS minus one. The format of the returned string is identical to the text that may follow #version in shader program source and is formatted as the version number followed, for versions in which language profiles are defined, by a space and a profile name. For example, a returned string containing "420 core" indicates support for OpenGL Shading Language 4.20, core profile. An empty string indicates support for OpenGL Shading Language 1.10, which did not include the #version compiler directive. The profile string will always be present in the returned string when it is accepted by that version of the OpenGL Shading Language, even though there is a default profile string in versions 1.50 and greater. Version strings 100, 300 es, and 310 es correspond to OpenGL ES Shading Language versions 1.00, 3.00 and 3.10, respectively.

An *index* of zero will always return the string for the version of the most recent shading language supported by the GL and the profile of the shading language corresponding to the profile of the API (e.g. the first entry returned in an OpenGL 4.30 core profile context will be "430 core" and the first entry returned in an OpenGL 4.30 compatibility profile context will be "430 compatibility"). There is no defined ordering of the returned strings for other values of *index*.

If *name* is SPIR_V_EXTENSIONS, the SPIR-V extension name corresponding to the *index*th supported SPIR-V extension (specified as the Name String in the corresponding SPIR-V extension document) will be returned. *index* may range from zero to the value of NUM_SPIR_V_EXTENSIONS minus one. The value of NUM_SPIR_V_EXTENSIONS can be queried using the **GetInteger** command. There is no defined relationship between any particular extension name and the *index* values; a SPIR-V extension name may correspond to a different *index* in different GL contexts and/or implementations.

Errors

An INVALID_ENUM error is generated if *name* is not EXTENSIONS, SHADING_LANGUAGE_VERSION or SPIR_V_EXTENSIONS.

An INVALID_VALUE error is generated if *index* is outside the valid range for the indexed state *name*.

22.3 Internal Format Queries

Information about implementation-dependent support for internal formats can be queried with the command

Target	Usage
TEXTURE_1D	1D texture
TEXTURE_1D_ARRAY	1D array texture
TEXTURE_2D	2D texture
TEXTURE_2D_ARRAY	2D array texture
TEXTURE_2D_MULTISAMPLE	2D multisample texture
TEXTURE_2D_MULTISAMPLE_ARRAY	2D multisample array texture
TEXTURE_3D	3D texture
TEXTURE_BUFFER	buffer texture
TEXTURE_CUBE_MAP	cube map texture
TEXTURE_CUBE_MAP_ARRAY	cube map array texture
TEXTURE_RECTANGLE	rectangle texture
RENDERBUFFER	renderbuffer

Table 22.2: Possible targets that *internalformat* can be used with and the corresponding usage meaning.

```
void GetInternalformativ( enum target, enum internalformat,
    enum pname, sizei count, int *params );
void GetInternalformati64v( enum target,
    enum internalformat, enum pname, sizei count,
    int64 *params );
```

internalformat can be any value. The INTERNALFORMAT_SUPPORTED *pname* can be used to determine if the internal format is supported, and the other *pnames* are defined in terms of whether or not the format is supported.

target indicates the usage of the *internalformat*, and must be one the targets listed in table 22.2.

No more than *count* integers will be written into *params*. If more data are available, they will be ignored and no error will be generated.

pname indicates the information to query. The following subsections list the valid values for *pname* and define their meaning and the values that may be returned. In the following descriptions, the term *resource* is used to generically refer to an object of the appropriate type that has been created with *internalformat* and *target*. If the particular *target* and *internalformat* combination does not make sense, or if a particular type of *target* is not supported by the implementation the *unsupported* answer should be given. This is not an error.

All properties may be queried via either **GetInternalformat*** command. Data conversions are done as defined in section 2.2.2.

22.3.1 Supported Operation Queries

Queries that return information about supported types of operations will return one of the following values in *params*:

- NONE: the requested resource or operation is not supported at all by the implementation.
- CAVEAT_SUPPORT: the requested operation is supported by the implementation, but there may be some implementation-specific caveats that make support less than optimal. For example using the feature may result in reduced performance (relative to other formats or features), such as software rendering or other mechanisms of emulating the desired feature.

  If a query reports that there is a caveat and the debug output functionality is enabled (see section 20), the GL will generate a debug output message describing the caveat. The message has the source DEBUG_SOURCE_API, the type DEBUG_TYPE_PERFORMANCE, and an implementation-dependent ID.
- FULL_SUPPORT: the requested operation is fully supported by the implementation.

Possible *pname*s for supported types of operations, and their meanings, include:

- AUTO_GENERATE_MIPMAP: Support for automatic generation of mipmaps for the resource is returned in *params*.
- CLEAR_BUFFER: Support for using the resource with **ClearBuffer*Data** commands is returned in *params*.
- CLEAR_TEXTURE: Support for using the resource with **ClearTex*Image** commands is returned in *params*.
- COMPUTE_TEXTURE: Support for using the resource as a source for texture sampling in a compute shader is written to *params*.
- FILTER: Support for filter types other than NEAREST or NEAREST_MIPMAP_NEAREST for the resource is written to *params*. This indicates if sampling from such resources supports setting the MIN/MAG filters to LINEAR values.
- FRAGMENT_TEXTURE: Support for using the resource as a source for texture sampling in a fragment shader is written to *params*.

- FRAMEBUFFER_BLEND: Support for rendering to the resource via framebuffer attachment when blending is enabled is returned in *params*.
- FRAMEBUFFER_RENDERABLE: Support for rendering to the resource via framebuffer attachment is returned in *params*.
- FRAMEBUFFER_RENDERABLE_LAYERED: Support for layered rendering to the resource via framebuffer attachment is returned in *params*.
- GEOMETRY_TEXTURE: Support for using the resource as a source for texture sampling in a geometry shader is written to *params*.
- MANUAL_GENERATE_MIPMAP: Support for manually generating mipmaps for the resource is returned in *params*.
- READ_PIXELS: Support for reading pixels from the resource when it is attached to a framebuffer is returned in *params*.
- SHADER_IMAGE_ATOMIC: Support for using the resource with atomic memory operations from shaders is written to *params*.
- SHADER_IMAGE_LOAD: Support for using the resource with image load operations in shaders is written to *params*. In this case the *internalformat* is the value of the *format* parameter that would be passed to **BindImageTexture**.
- SHADER_IMAGE_STORE: Support for using the resource with image store operations in shaders is written to *params*. In this case the *internalformat* is the value of the *format* parameter that is passed to **BindImageTexture**.
- SIMULTANEOUS_TEXTURE_AND_DEPTH_TEST: Support for using the resource both as a source for texture sampling while it is bound as a buffer for depth test is written to *params*. For example, a depth (or stencil) texture could be bound simultaneously for texturing while it is bound as a depth (and/or stencil) buffer without causing a feedback loop, provided that depth writes are disabled.
- SIMULTANEOUS_TEXTURE_AND_DEPTH_WRITE: Support for using the resource both as a source for texture sampling while performing depth writes to the resources is written to *params*. For example, a depth-stencil texture could be bound simultaneously for stencil texturing while it is bound as a depth buffer. Feedback loops cannot occur because sampling a stencil texture only returns the stencil portion, and thus writes to the depth buffer do not modify the stencil portions.

- SIMULTANEOUS_TEXTURE_AND_STENCIL_TEST: Support for using the resource both as a source for texture sampling while it is bound as a buffer for stencil test is written to *params*. For example, a depth (or stencil) texture could be bound simultaneously for texturing while it is bound as a depth (and/or stencil) buffer without causing a feedback loop, provided that stencil writes are disabled.

- SIMULTANEOUS_TEXTURE_AND_STENCIL_WRITE: Support for using the resource both as a source for texture sampling while performing stencil writes to the resources is written to *params*. For example, a depth-stencil texture could be bound simultaneously for depth-texturing while it is bound as a stencil buffer. Feedback loops cannot occur because sampling a depth texture only returns the depth portion, and thus writes to the stencil buffer could not modify the depth portions.

- SRGB_READ: Support for converting from sRGB colorspace on read operations (see section 8.24) from the resource is returned in *params*.

- SRGB_WRITE: Support for converting to sRGB colorspace on write operations to the resource is returned in *params*. This indicates that writing to framebuffers with this internal format will encode to sRGB color spaces when FRAMEBUFFER_SRGB is enabled (see section 17.3.7).

- TESS_CONTROL_TEXTURE: Support for using the resource as a source for texture sampling in a tessellation control shader is written to *params*.

- TESS_EVALUATION_TEXTURE: Support for using the resource as a source for texture sampling in a tessellation evaluation shader is written to *params*.

- TEXTURE_GATHER: Support for using the resource with texture gather operations is written to *params*.

- TEXTURE_GATHER_SHADOW: Support for using resource with texture gather operations with shadow samplers is written to *params*.

- TEXTURE_SHADOW: Support for using the resource with shadow samplers is written to *params*.

- TEXTURE_VIEW: Support for using the resource with the **TextureView** command is returned in *params*.

- VERTEX_TEXTURE: Support for using the resource as a source for texture sampling in a vertex shader is written to *params*.

22.3.2 Other Internal Format Queries

Other supported values for *pname*, their meanings, and their possible return values include:

- `COLOR_COMPONENTS`: If the internal format contains any color components (R, G, B, or A), `TRUE` is returned in *params*. If the internal format is unsupported or contains no color components, `FALSE` is returned.
- `COLOR_ENCODING`: The color encoding for the resource is returned in *params*. Possible values for color buffers are `LINEAR` or `SRGB`, for linear or sRGB-encoded color components, respectively. For non-color formats (such as depth or stencil), or for unsupported resources, the value `NONE` is returned.
- `COLOR_RENDERABLE`: If *internalformat* is color-renderable (as defined in section 9.4), `TRUE` is returned in *params*. If the internal format is unsupported, or the internal format is not color-renderable, `FALSE` is returned.
- `DEPTH_COMPONENTS`: If the internal format contains a depth component (D), `TRUE` is returned in *params*. If the internal format is unsupported or contains no depth component, `FALSE` is returned.
- `DEPTH_RENDERABLE`: If *internalformat* is depth-renderable (as defined in section 9.4), `TRUE` is returned in *params*. If the internal format is unsupported, or if the internal format is not depth-renderable, `FALSE` is returned.
- `GET_TEXTURE_IMAGE_FORMAT`: The implementation-preferred *format* to pass to **GetTexImage** when querying texture image data from this resource. Possible values include any value that is legal to pass for the *format* parameter to **GetTexImage**, or `NONE` if the resource does not support this operation, or if **GetTexImage** is not supported.
- `GET_TEXTURE_IMAGE_TYPE`: The implementation-preferred *type* to pass to **GetTexImage** when querying texture image data from this resource. Possible values include any value that is legal to pass for the *type* parameter to **GetTexImage**, or `NONE` if the resource does not support this operation, or if **GetTexImage** is not supported.
- `IMAGE_COMPATIBILITY_CLASS`: The compatibility class of the resource when used as an image texture is returned in *params*. This corresponds to the value from the **Class** column in table 8.35. The possible values returned are `IMAGE_CLASS_4_X_32`, `IMAGE_CLASS_2_X_32`,

IMAGE_CLASS_1_X_32, IMAGE_CLASS_4_X_16, IMAGE_CLASS_2_X_16, IMAGE_CLASS_1_X_16, IMAGE_CLASS_4_X_8, IMAGE_CLASS_2_X_8, IMAGE_CLASS_1_X_8, IMAGE_CLASS_11_11_10, and IMAGE_CLASS_10_10_10_2, which correspond to the 4x32, 2x32, 1x32, 4x16, 2x16, 1x16, 4x8, 2x8, 1x8, the class (a) 11/11/10 packed floating-point format, and the class (b) 10/10/10/2 packed formats, respectively. If the resource is not supported for image textures, or if image textures are not supported, NONE is returned.

- IMAGE_FORMAT_COMPATIBILITY_TYPE: The matching criteria use for the resource when used as an image textures is returned in *params*. This is equivalent to calling **GetTexParameter** with *pname* set to IMAGE_FORMAT_COMPATIBILITY_TYPE. Possible values are IMAGE_FORMAT_COMPATIBILITY_BY_SIZE or IMAGE_FORMAT_COMPATIBILITY_BY_CLASS. If the resource is not supported for image textures, or if image textures are not supported, NONE is returned.

- IMAGE_PIXEL_FORMAT: The pixel format of the resource when used as an image texture is returned in *params*. This is the value from the **Pixel *format*** column in table 8.35. If the resource is not supported for image textures, or if image textures are not supported, NONE is returned.

- IMAGE_PIXEL_TYPE: The pixel type of the resource when used as an image texture is returned in *params*. This is the value from the **Pixel *type*** column in table 8.35. If the resource is not supported for image textures, or if image textures are not supported, NONE is returned.

- IMAGE_TEXEL_SIZE: The size of a texel of the resource when used as an image texture is returned in *params*. This is the value from the **Size** column in table 8.35. If the resource is not supported for image textures, or if image textures are not supported, zero is returned.

- INTERNALFORMAT_PREFERRED: The implementation-preferred internal format for representing resources of the specified *internalformat* is returned in *params*. The preferred internal format should have no less precision than the requested one. If the specified *internalformat* is already a preferred format, or if there is no better format that is compatible, the queried *internalformat* value is written to *params*. If the *internalformat* is not supported, NONE is returned.

- INTERNALFORMAT_RED_SIZE, INTERNALFORMAT_GREEN_SIZE, INTERNALFORMAT_BLUE_SIZE, INTERNALFORMAT_ALPHA_SIZE,

INTERNALFORMAT_DEPTH_SIZE, INTERNALFORMAT_STENCIL_SIZE, or INTERNALFORMAT_SHARED_SIZE:

For uncompressed internal formats, queries of these values return the actual resolutions that would be used for storing image components for the resource. For compressed internal formats, the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed algorithm. For textures this query will return the same information as querying **GetTexLevelParameter*** for the corresponding TEXTURE_*_SIZE (except in cases where **GetTexLevelParameter*** does not support such a query). If the internal format is unsupported, or if a particular component is not present in the format, 0 is written to *params*.

- INTERNALFORMAT_RED_TYPE, INTERNALFORMAT_GREEN_TYPE, INTERNALFORMAT_BLUE_TYPE, INTERNALFORMAT_ALPHA_TYPE, INTERNALFORMAT_DEPTH_TYPE, or INTERNALFORMAT_STENCIL_TYPE:

  For uncompressed internal formats, queries for these values return the data type used to store the component. For compressed internal formats the types returned specify how components are interpreted after decompression. For textures this query returns the same information as querying **GetTexLevelParameter*** for the corresponding TEXTURE_*_TYPE (except in cases where **GetTexLevelParameter*** does not support such a query). Possible values returned include NONE, SIGNED_NORMALIZED, UNSIGNED_NORMALIZED, FLOAT, INT, and UNSIGNED_INT, representing missing, signed normalized fixed-point, unsigned normalized fixed-point, floating-point, signed unnormalized integer, and unsigned unnormalized integer components respectively. NONE is returned for all component types if the format is unsupported.

- INTERNALFORMAT_SUPPORTED: If *internalformat* is an internal format that is supported by the implementation in at least some subset of possible operations, TRUE is written to *params*. If *internalformat* if not a valid token for any internal format usage, FALSE is returned.

  internalformats that must be supported include:

  - sized internal formats from tables 8.19- 8.21 and 8.24,
  - any specific compressed internal format from table 8.22,
  - any image unit format from table 8.34,

- any generic compressed internal format from table 8.22, if the implementation accepts it for any texture specification commands, and
- any unsized or base internal format, if the implementation accepts it for texture or image specification.

In other words, any *internalformat* accepted by any of the commands: **ClearBufferData**, **ClearBufferSubData**, **CompressedTexImage1D**, **CompressedTexImage2D**, **CompressedTexImage3D**, **CopyTexImage1D**, **CopyTexImage2D**, **RenderbufferStorage**, **RenderbufferStorageMultisample**, **TexBuffer**, **TexImage1D**, **TexImage2D**, **TexImage3D**, **TexImage2DMultisample**, **TexImage3DMultisample**, **TexStorage1D**, **TexStorage2D**, **TexStorage3D**, **TexStorage2DMultisample**, **TexStorage3DMultisample**, and **TextureView**, and any valid *format* accepted by **BindImageTexture**, must be supported.

- `MAX_COMBINED_DIMENSIONS`: The maximum combined dimensions for the resource is returned in *params*. The combined dimensions is the product of the individual dimensions of the resource. For multisampled surfaces the number of samples is considered an additional dimension. Note that the value returned can be $\geq 2^{32}$ and should be queried with **GetInternalformati64v**.

  This value should be considered a recommendation for applications. There may be system-dependant reasons why allocations larger than this size may fail, even if there might appear to be sufficient memory available when queried via some other means. This also does not provide a guarantee that allocations smaller than this will succeed because this value is not affected by existing resource allocations.

  For one-dimensional targets this is the maximum single dimension. For one-dimensional array targets this is the maximum combined width and layers. For two-dimensional targets this is the maximum combined width and height. For two-dimensional multisample targets this is the combined width, height and samples. For two-dimensional array targets this is the max combined width, height and layers. For two-dimensional multisample array targets, this is the max combined width, height, layers and samples. For three-dimensional targets this is the maximum combined width, height and depth. For cube map targets this is the maximum combined width, height and faces. For cube map array targets this is the maximum width, height and layer-faces. If the resource is unsupported, zero is returned.

- `MAX_DEPTH`: The maximum supported depth for the resource is returned in

params. For resources with three or more dimensions, the third dimension is considered the depth. If the resource does not have at least three dimensions, or if the resource is unsupported, zero is returned.

- MAX_HEIGHT: The maximum supported height for the resource is returned in *params*. For resources with two or more dimensions, the second dimension is considered the height. If the resource does not have at least two dimensions, or if the resource is unsupported, zero is returned.

- MAX_LAYERS: The maximum supported number of layers for the resource is returned in *params*. For 1D array targets, the value returned is the same as the MAX_HEIGHT. For 2D and cube array targets, the value returned is the same as the MAX_DEPTH. If the resource does not support layers, or if the resource is unsupported, zero is returned.

- MAX_WIDTH: The maximum supported width for the resource is returned in *params*. For resources with only one dimension, that dimension is considered the width. If the resource is unsupported, zero is returned.

- MIPMAP: If the resource supports mipmaps, TRUE is returned in *params*. If the resource is not supported, or if mipmaps are not supported for this type of resource, FALSE is returned.

- NUM_SAMPLE_COUNTS: The number of sample counts that would be returned by querying SAMPLES is returned in *params*.

  – If *internalformat* is not color-renderable, depth-renderable, or stencil-renderable (as defined in section 9.4), or if *target* does not support multiple samples (is not TEXTURE_2D_MULTISAMPLE, TEXTURE_2D_MULTISAMPLE_ARRAY, or RENDERBUFFER), zero is returned.

- READ_PIXELS_FORMAT: The *format* to pass to **ReadPixels** to obtain the best performance and image quality when reading from framebuffers with *internalformat* is returned in *params*. Possible values include any value that is legal to pass for the *format* parameter to **ReadPixels**, or NONE if *internalformat* is not supported or can never be a valid source for **ReadPixels**.

- READ_PIXELS_TYPE: The *type* to pass to **ReadPixels** to obtain the best performance and image quality when reading from framebuffers with *internalformat* is returned in *params*. Possible values include any value that is legal to pass for the *type* parameter to **ReadPixels**, or NONE if the internal format is not supported or can never be a source for **ReadPixels**.

- SAMPLES: The sample counts supported for *internalformat* and *target* are written into *params*, in descending numeric order. Only positive values are returned.
  - Note that querying SAMPLES with a *count* of one will return just the maximum supported number of samples for this format.
  - The maximum value in SAMPLES is guaranteed to be at least the lowest of the following:
    * The value of MAX_INTEGER_SAMPLES, if *internalformat* is a signed or unsigned integer format.
    * The value of MAX_DEPTH_TEXTURE_SAMPLES, if *internalformat* is a depth/stencil-renderable format and *target* is TEXTURE_2D_MULTISAMPLE or TEXTURE_2D_MULTISAMPLE_ARRAY.
    * The value of MAX_COLOR_TEXTURE_SAMPLES, if *internalformat* is a color-renderable format and *target* is TEXTURE_2D_MULTISAMPLE or TEXTURE_2D_MULTISAMPLE_ARRAY.
    * The value of MAX_SAMPLES.
  - If *internalformat* is not color-renderable, depth-renderable, or stencil-renderable (as defined in section 9.4), or if *target* does not support multiple samples (i.e. other than TEXTURE_2D_MULTISAMPLE, TEXTURE_2D_MULTISAMPLE_ARRAY, or RENDERBUFFER), *params* is not modified.
- STENCIL_COMPONENTS: If the internal format contains a stencil component (S), TRUE is returned in *params*. If the internal format is unsupported or contains no stencil component, FALSE is returned.
- STENCIL_RENDERABLE: If *internalformat* is stencil-renderable (as defined in section 9.4), TRUE is returned in *params*. If the internal format is unsupported, or if the internal format is not stencil-renderable, FALSE is returned.
- TEXTURE_COMPRESSED: If *internalformat* is a compressed format that is supported for this type of resource, TRUE is returned in *params*. If the internal format is not compressed, or the type of resource is not supported, FALSE is returned.
- TEXTURE_COMPRESSED_BLOCK_HEIGHT: If the resource contains a compressed format, the height of a compressed block (in texels) is returned in *params*. If the internal format is not compressed, or the resource is not supported, 0 is returned.

- TEXTURE_COMPRESSED_BLOCK_SIZE: If the resource contains a compressed format, the number of bytes per block is returned in *params*. If the internal format is not compressed, or the resource is not supported, 0 is returned. Together with the block width and height queries this allows the bitrate to be computed, and may be useful in conjunction with ARB_compressed_texture_pixel_storage).

- TEXTURE_COMPRESSED_BLOCK_WIDTH: If the resource contains a compressed format, the width of a compressed block (in texels) is returned in *params*. If the internal format is not compressed, or the resource is not supported, 0 is returned.

- TEXTURE_IMAGE_FORMAT: The implementation-preferred *format* to pass to **TexImage*D** or **TexSubImage*D** when specifying texture image data for this resource is returned in *params*. Possible values include any value that is legal to pass for the *format* parameter to the **Tex*Image*D** commands, or NONE if the resource is not supported for this operation.

- TEXTURE_IMAGE_TYPE: The implementation-preferred *type* to pass to **TexImage*D** or **TexSubImage*D** when specifying texture image data for this resource is returned in *params*. Possible values include any value that is legal to pass for the *type* parameter to the **Tex*Image*D** commands, or NONE if the resource is not supported for this operation.

- VIEW_COMPATIBILITY_CLASS: The compatibility class of the resource when used as a texture view is returned in *params*. The compatibility class is one of the values from the **Class** column of table 8.30. If the resource has no other formats that are compatible, the resource does not support views, or if texture views are not supported, NONE is returned.

Errors

An INVALID_ENUM error is generated if *target* is not one of the targets in table 22.2, or if *pname* is not one of the parameters described above.

An INVALID_VALUE error is generated if *count* is negative.

22.4 Transform Feedback State Queries

State of the currently bound transform feedback object may be queried by calling **GetIntegerv**, **GetIntegeri_v**, **GetInteger64i_v**, **GetBooleanv**, or other query functions with the query target set to one of the tokens listed in table 23.59.

Alternatively, the state of a transform feedback object may be queried with the commands

```
void GetTransformFeedbackiv( uint xfb, enum pname,
    int *param );
void GetTransformFeedbacki_v( uint xfb, enum pname,
    uint index, int *param );
void GetTransformFeedbacki64_v( uint xfb, enum pname,
    uint index, int64 *param );
```

xfb must be zero, indicating the default transform feedback object, or the name of an existing transform feedback object. *pname* must be one of the tokens listed in table 23.59, depending on the command name as shown in the errors section below. For indexed state, *index* is the index of the transform feedback stream. *param* is the address of a variable to receive the result of the query.

Errors

An INVALID_OPERATION error is generated by **GetTransformFeedback*** if *xfb* is not zero or the name of an existing transform feedback object.

An INVALID_ENUM error is generated by **GetTransformFeedbackiv** if *pname* is not TRANSFORM_FEEDBACK_PAUSED or TRANSFORM_FEEDBACK_ACTIVE.

An INVALID_ENUM error is generated by **GetTransformFeedbacki_v** if *pname* is not TRANSFORM_FEEDBACK_BUFFER_BINDING.

An INVALID_ENUM error is generated by **GetTransformFeedbacki64_v** if *pname* is not TRANSFORM_FEEDBACK_BUFFER_START or TRANSFORM_FEEDBACK_BUFFER_SIZE.

An INVALID_VALUE error is generated by **GetTransformFeedbacki_v** and **GetTransformFeedbacki64_v** if *index* is greater than or equal to the number of binding points for transform feedback, as described in section 6.7.1.

22.5 Indexed Binding State Queries

The name of the texture object bound to the active texture unit may be queried by calling **GetIntegerv** with *pname* TEXTURE_BINDING_1D, TEXTURE_BINDING_1D_ARRAY, TEXTURE_BINDING_2D, TEXTURE_BINDING_2D_ARRAY, TEXTURE_BINDING_2D_MULTISAMPLE, TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY,

TEXTURE_BINDING_3D, TEXTURE_BINDING_BUFFER, TEXTURE_BINDING_-CUBE_MAP, TEXTURE_BINDING_CUBE_MAP_ARRAY, or TEXTURE_BINDING_-RECTANGLE. Likewise, the current sampler bound to the active texture unit may be queried by calling **GetIntegerv** with *pname* SAMPLER_BINDING.

To query the bound texture or sampler object bound to a specific texture unit without changing the active texture selector, call **GetIntegeri_v** with one of the valid *pname*s listed above, and with *index* set to the zero-based texture unit index to be queried.

Chapter 23

State Tables

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of the simple queries in section 22.1 are listed with just one of these commands – the one that is most appropriate given the type of the data to be returned. These state variables cannot be obtained using **IsEnabled**. However, state variables for which **IsEnabled** is listed as the query command can also be obtained using any of the simple queries. State variables for which any other command is listed as the query command can be obtained by using that command or any of its typed variants, although information may be lost when not using the listed command. Unless otherwise specified, when floating-point state is returned as integer values or integer state is returned as floating-point values it is converted in the fashion described in section 2.2.2.

State table entries which are required only by the imaging subset (see section 8.4.2) are typeset against a gray background.

State table entries indicate a type for each variable. Table 23.1 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with all matrices, where only the top entry on the stack is returned; with clip planes, where only the selected clip plane is returned; with parameters describing lights, where only the value pertaining to the selected light is returned; with evaluator maps, where only the selected map is returned; and with textures, where only the selected texture or texture parameter is returned. Finally, a "–" in the attribute column indicates that the indicated value is not included in any attribute group (and thus can not be pushed or popped with **PushAttrib**, **PushClientAttrib**, **PopAttrib**, or **PopClientAttrib**).

The M and m entries for initial minmax table values represent the maximum

and minimum possible representable values, respectively.

The abbreviations *max.*, *min.*, and *no.* are used interchangeably with *maximum*, *minimum*, and *number*, respectively, to help fit tables without overflowing pages.

Type code	Explanation
B	Boolean
BMU	Basic machine units
C	Color (floating-point R, G, B, and A values)
E	Enumerated value (as described in spec body)
CI	Color index (floating-point index value)
T	Texture coordinates (floating-point (s, t, r, q) values)
N	Normal coordinates (floating-point (x, y, z) values)
V	Vertex, including associated data
Z	Integer
Z^+	Non-negative integer or enumerated value
Z_k, Z_{k*}	k-valued integer ($k*$ indicates k is minimum)
R	Floating-point number
R^+	Non-negative floating-point number
$R^{[a,b]}$	Floating-point number in the range $[a, b]$
R^k	k-tuple of floating-point numbers
P	Position ((x, y, z, w) floating-point coordinates)
D	Direction ((x, y, z) floating-point coordinates)
M^4	4×4 floating-point matrix
S	null-terminated string
I	Image
A	Attribute stack entry, including mask
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is minimum)

Table 23.1: State Variable Types

Table 23.2: GL Internal State (inaccessible)(for **Begin/End** state variables)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
-	E	–	0	When $\neq 0$, indicates primitive *mode*	10.1	–
-	V	–	–	Previous vertex in **Begin/End** line	10.1	–
-	B	–	–	Indicates if *line-vertex* is the first	10.1	–
-	V	–	–	First vertex of a **Begin/End** line loop	10.1	–
-	Z^+	–	–	Line stipple counter	14.5	–
-	$n \times V$	–	–	Vertices inside of **Begin/End** polygon	10.1	–
-	Z^+	–	–	No. of *polygon-vertices*	10.1	–
-	$2 \times V$	–	–	Previous two vertices in a **Begin/End** triangle strip	10.1	–
-	Z_3	–	–	No. of vertices so far in triangle strip: 0, 1, or more	10.1	–
-	Z_2	–	–	Triangle strip A/B vertex pointer	10.1	–
-	$3 \times V$	–	–	Vertices of the quad under construction	10.1	–
-	Z_4	–	–	No. of vertices so far in quad strip: 0, 1, 2, or more	10.1	–

Table 23.3: Current Values and Associated Data

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR	C	**GetFloatv**	1.0,1.0,1.0,1.0	Current color	10.2	current
CURRENT_SECONDARY_COLOR	C	**GetFloatv**	0.0,0.0,0.0,1.0	Current secondary color	10.2	current
CURRENT_INDEX	CI	**GetIntegerv**	1	Current color index	10.2	current
CURRENT_TEXTURE_COORDS	$8 * \times T$	**GetFloatv**	0,0,0,1	Current texture coordinates	10.2	current
CURRENT_NORMAL	N	**GetFloatv**	0,0,1	Current normal	10.2	current
CURRENT_FOG_COORD	R	**GetFloatv**	0	Current fog coordinate	10.2	current
–	C	–	-	Color associated with last vertex	10.1	–
–	CI	–	-	Color index associated with last vertex	10.1	–
–	T	–	-	Texture coordinates associated with last vertex	10.1	–
CURRENT_RASTER_POSITION	R^4	**GetFloatv**	0,0,0,1	Current raster position	14.7	current
CURRENT_RASTER_DISTANCE	R^+	**GetFloatv**	0	Current raster distance	14.7	current
CURRENT_RASTER_COLOR	C	**GetFloatv**	1.0,1.0,1.0,1.0	Color associated with raster position	14.7	current
CURRENT_RASTER_SECONDARY_COLOR	C	**GetFloatv**	0.0,0.0,0.0,1.0	Secondary color associated with raster position	14.7	current
CURRENT_RASTER_INDEX	CI	**GetIntegerv**	1	Color index associated with raster position	14.7	current
CURRENT_RASTER_TEXTURE_COORDS	$8 * \times T$	**GetFloatv**	0,0,0,1	Texture coordinates associated with raster position	14.7	current
CURRENT_RASTER_POSITION_VALID	B	**GetBooleanv**	TRUE	Raster position valid bit	14.7	current
EDGE_FLAG	B	**GetBooleanv**	TRUE	Edge flag	10.1.17	current
PATCH_VERTICES	Z^+	**GetIntegerv**	3	No. of vertices in input patch	10.1	current
PATCH_DEFAULT_OUTER_LEVEL	$4 \times R$	**GetFloatv**	$(1.0, 1.0, 1.0, 1.0)$	Default outer tess. level w/o control shader	11.2.2	–
PATCH_DEFAULT_INNER_LEVEL	$2 \times R$	**GetFloatv**	$(1.0, 1.0)$	Default inner tess. level w/o control shader	11.2.2	–

Table 23.4: Vertex Array Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY	B	**IsEnabled**	FALSE	Vertex array enable	10.3	vertex-array
VERTEX_ARRAY_SIZE	Z^+	**GetIntegerv**	4	Coordinates per vertex	10.3	vertex-array
VERTEX_ARRAY_TYPE	E	**GetIntegerv**	FLOAT	Type of vertex coordinates	10.3	vertex-array
VERTEX_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between vertices	10.3	vertex-array
VERTEX_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the vertex array	10.3	vertex-array
NORMAL_ARRAY	B	**IsEnabled**	FALSE	Normal array enable	10.3	vertex-array
NORMAL_ARRAY_TYPE	E	**GetIntegerv**	FLOAT	Type of normal coordinates	10.3	vertex-array
NORMAL_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between normals	10.3	vertex-array
NORMAL_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the normal array	10.3	vertex-array
FOG_COORD_ARRAY	B	**IsEnabled**	FALSE	Fog coord array enable	10.3	vertex-array
FOG_COORD_ARRAY_TYPE	E	**GetIntegerv**	FLOAT	Type of fog coord components	10.3	vertex-array
FOG_COORD_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between fog coords	10.3	vertex-array
FOG_COORD_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the fog coord array	10.3	vertex-array
COLOR_ARRAY	B	**IsEnabled**	FALSE	Color array enable	10.3	vertex-array
COLOR_ARRAY_SIZE	Z_3	**GetIntegerv**	4	Color components per vertex	10.3	vertex-array
COLOR_ARRAY_TYPE	E	**GetIntegerv**	FLOAT	Type of color components	10.3	vertex-array
COLOR_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between colors	10.3	vertex-array
COLOR_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the color array	10.3	vertex-array

Table 23.5: Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SECONDARY_COLOR_ARRAY	B	**IsEnabled**	FALSE	Secondary color array enable	10.3	vertex-array
SECONDARY_COLOR_ARRAY_SIZE	Z_2	**GetIntegerv**	3	Secondary color components per vertex	10.3	vertex-array
SECONDARY_COLOR_ARRAY_TYPE	E	**GetIntegerv**	FLOAT	Type of secondary color components	10.3	vertex-array
SECONDARY_COLOR_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between secondary colors	10.3	vertex-array
SECONDARY_COLOR_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the secondary color array	10.3	vertex-array
INDEX_ARRAY	B	**IsEnabled**	FALSE	Index array enable	10.3	vertex-array
INDEX_ARRAY_TYPE	E	**GetIntegerv**	FLOAT	Type of indices	10.3	vertex-array
INDEX_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between indices	10.3	vertex-array
INDEX_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the index array	10.3	vertex-array
TEXTURE_COORD_ARRAY	$8 * \times B$	**IsEnabled**	FALSE	Texture coordinate array enable	10.3	vertex-array
TEXTURE_COORD_ARRAY_SIZE	$8 * \times Z^+$	**GetIntegerv**	4	Coordinates per element	10.3	vertex-array
TEXTURE_COORD_ARRAY_TYPE	$8 * \times E$	**GetIntegerv**	FLOAT	Type of texture coordinates	10.3	vertex-array
TEXTURE_COORD_ARRAY_STRIDE	$8 * \times Z^+$	**GetIntegerv**	0	Stride between texture coordinates	10.3	vertex-array
TEXTURE_COORD_ARRAY_POINTER	$8 * \times Y$	**GetPointerv**	0	Pointer to the texture coordinate array	10.3	vertex-array

Table 23.6: Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ATTRIB_ARRAY_ENABLED	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array enable	10.3	vertex-array
VERTEX_ATTRIB_ARRAY_SIZE	$16 * \times Z_5$	**GetVertexAttribiv**	4	Vertex attrib array size	10.3	vertex-array
VERTEX_ATTRIB_ARRAY_STRIDE	$16 * \times Z^+$	**GetVertexAttribiv**	0	Vertex attrib array stride	10.3	vertex-array
VERTEX_ATTRIB_ARRAY_TYPE	$16 * \times E$	**GetVertexAttribiv**	FLOAT	Vertex attrib array type	10.3	vertex-array
VERTEX_ATTRIB_ARRAY_NORMALIZED	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array normalized	10.3	vertex-array
VERTEX_ATTRIB_ARRAY_INTEGER	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array has unconverted integers	10.3	vertex-array
VERTEX_ATTRIB_ARRAY_LONG	$16 * \times B$	**GetVertexAttribiv**	FALSE	Vertex attrib array has unconverted doubles	10.3	vertex-array
VERTEX_ATTRIB_ARRAY_DIVISOR	$16 * \times Z^+$	**GetVertexAttribiv**	0	Vertex attrib array instance divisor	10.4	vertex-array
VERTEX_ATTRIB_ARRAY_POINTER	$16 * \times Y$	**GetVertex-AttribPointerv**	NULL	Vertex attrib array pointer	10.3	vertex-array
EDGE_FLAG_ARRAY	B	**IsEnabled**	FALSE	Edge flag array enable	10.3	vertex-array
EDGE_FLAG_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between edge flags	10.3	vertex-array
EDGE_FLAG_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the edge flag array	10.3	vertex-array
–	S	**GetObjectLabel**	empty	Debug label	20.9	–

Table 23.7: Vertex Array Object State (cont.)
† The ith attribute defaults to a value of i.

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Vertex array buffer binding	6	vertex-array
NORMAL_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Normal array buffer binding	6	vertex-array
COLOR_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Color array buffer binding	6	vertex-array
INDEX_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Index array buffer binding	6	vertex-array
TEXTURE_COORD_ARRAY_BUFFER_BINDING	$8 * \times Z^+$	**GetIntegerv**	0	Texcoord array buffer binding	6	vertex-array
EDGE_FLAG_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Edge flag array buffer binding	6	vertex-array
SECONDARY_COLOR_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Secondary color array buffer binding	6	vertex-array
FOG_COORD_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Fog coordinate array buffer binding	6	vertex-array
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Element array buffer binding	10.3.10	vertex-array
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	$16 * \times Z^+$	**GetVertexAttribiv**	0	Attribute array buffer binding	6	vertex-array
VERTEX_ATTRIB_BINDING	$16 \times Z_{16*}$	**GetVertexAttribiv**	i†	Vertex buffer binding used by vertex attrib i	10.3	vertex-array
VERTEX_ATTRIB_RELATIVE_OFFSET	$16 \times Z^+$	**GetVertexAttribiv**	0	Byte offset added to vertex binding offset for this attribute	10.3	vertex-array
VERTEX_BINDING_OFFSET	$16 \times Z$	**GetInteger64i_v**	0	Byte offset of the first element in data store of the buffer bound to vertex binding i	10.3	vertex-array
VERTEX_BINDING_STRIDE	$16 \times Z$	**GetIntegeri_v**	16	Stride between elements in vertex binding i	10.3	vertex-array
VERTEX_BINDING_DIVISOR	$16 \times Z$	**GetIntegeri_v**	0	Instance divisor for vertex binding i	10.3	vertex-array
VERTEX_BINDING_BUFFER	$16 \times Z$	**GetIntegeri_v**	0	Name of buffer bound to vertex binding i	10.3	vertex-array

Table 23.8: Vertex Array Data (not in Vertex Array objects)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CLIENT_ACTIVE_TEXTURE	E	**GetIntegerv**	TEXTURE0	Client active texture unit selector	10.2	vertex-array
ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current buffer binding	6	vertex-array
DRAW_INDIRECT_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Indirect command buffer binding	10.3.11	–
VERTEX_ARRAY_BINDING	Z^+	**GetIntegerv**	0	Current vertex array object binding	10.3.1	vertex-array
PARAMETER_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Indirect parameter buffer binding	10.4	–
PRIMITIVE_RESTART	B	**IsEnabled**	FALSE	Primitive restart enable	10.3	vertex-array
PRIMITIVE_RESTART_FIXED_INDEX	B	**IsEnabled**	FALSE	Primitive restart fixed index enable	10.3.6	vertex-array
PRIMITIVE_RESTART_INDEX	Z^+	**GetIntegerv**	0	Primitive restart index	10.3.6	vertex-array

Table 23.9: Buffer Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
–	$n \times BMU$	**GetBufferSubData**	-	Buffer data	6	–
BUFFER_SIZE	$n \times Z^+$	**GetBufferParameteri64v**	0	Buffer data size	6	–
BUFFER_USAGE	$n \times E$	**GetBufferParameteriv**	STATIC_DRAW	Buffer usage pattern	6	–
BUFFER_ACCESS	$n \times E$	**GetBufferParameteriv**	READ_WRITE	Buffer access flag	6.3	–
BUFFER_ACCESS_FLAGS	$n \times Z^+$	**GetBufferParameteriv**	0	Extended buffer access flag	6.3	–
BUFFER_IMMUTABLE_STORAGE	B	**GetBufferParameteriv**	FALSE	TRUE if buffer's data store is immutable, FALSE otherwise	6	–
BUFFER_STORAGE_FLAGS	Z^+	**GetBufferParameteriv**	0	Buffer object storage flags	6	–
BUFFER_MAPPED	$n \times B$	**GetBufferParameteriv**	FALSE	Buffer map flag	6.3	–
BUFFER_MAP_POINTER	$n \times Y$	**GetBufferPointerv**	NULL	Mapped buffer pointer	6.3	–
BUFFER_MAP_OFFSET	$n \times Z^+$	**GetBufferParameteri64v**	0	Start of mapped buffer range	6.3	–
BUFFER_MAP_LENGTH	$n \times Z^+$	**GetBufferParameteri64v**	0	Size of mapped buffer range	6.3	–
–	S	**GetObjectLabel**	empty	Debug label	20.9	–

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
COLOR_MATRIX (TRANSPOSE_COLOR_MATRIX)	$2 * \times M^4$	**GetFloatv**	Identity	Color matrix stack	8.4.3	–
MODELVIEW_MATRIX (TRANSPOSE_MODELVIEW_MATRIX)	$32 * \times M^4$	**GetFloatv**	Identity	Model-view matrix stack	12.1.1	–
PROJECTION_MATRIX (TRANSPOSE_PROJECTION_MATRIX)	$2 * \times M^4$	**GetFloatv**	Identity	Projection matrix stack	12.1.1	–
TEXTURE_MATRIX (TRANSPOSE_TEXTURE_MATRIX)	$8 * \times 2 * \times M^4$	**GetFloatv**	Identity	Texture matrix stack	12.1.1	–
VIEWPORT	$16 * \times 4 \times R$	**GetFloati_v**	See sec. 13.8.1	Viewport origin & extent	13.8.1	viewport
DEPTH_RANGE	$16 * \times 2 \times R^{[0,1]}$	**GetDoublei_v**	0,1	Depth range near & far	13.8.1	viewport
COLOR_MATRIX_STACK_DEPTH	Z^+	**GetIntegerv**	1	Color matrix stack pointer	8.4.3	–
MODELVIEW_STACK_DEPTH	Z^+	**GetIntegerv**	1	Model-view matrix stack pointer	12.1.1	–
PROJECTION_STACK_DEPTH	Z^+	**GetIntegerv**	1	Projection matrix stack pointer	12.1.1	–
TEXTURE_STACK_DEPTH	$8 * \times Z^+$	**GetIntegerv**	1	Texture matrix stack pointer	12.1.1	–
MATRIX_MODE	E	**GetIntegerv**	MODELVIEW	Current matrix mode	12.1.1	transform
NORMALIZE	B	**IsEnabled**	FALSE	Current normal normalization on/off	12.1.2	transform/enable
RESCALE_NORMAL	B	**IsEnabled**	FALSE	Current normal rescaling on/off	12.1.2	transform/enable
CLIP_PLANEi	$6 * \times R^4$	**GetClipPlane**	0,0,0,0	User clipping plane coefficients	13.7	transform
CLIP_DISTANCEi	$8* \times B$	**IsEnabled**	FALSE	ith user clipping plane enabled	13.7	transform/enable
CLIP_ORIGIN	Z_2	**GetIntegerv**	LOWER_LEFT	Clip origin	13.7	xform
CLIP_DEPTH_MODE	Z_2	**GetIntegerv**	NEGATIVE_ONE_TO_ONE	Clip depth mode	13.7	xform
DEPTH_CLAMP	B	**IsEnabled**	FALSE	Depth clamping enabled	13.7	transform/enable
TRANSFORM_FEEDBACK_BINDING	Z^+	**GetIntegerv**	0	Object bound for transform feedback operations	13.3	–
TRANSFORM_FEEDBACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to generic bind point for transform feedback	6.7	–

Table 23.11: Coloring

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FOG_COLOR	C	**GetFloatv**	0.0,0.0,0.0,0.0	Fog color	16.4	fog
FOG_INDEX	CI	**GetFloatv**	0	Fog index	16.4	fog
FOG_DENSITY	R	**GetFloatv**	1.0	Exponential fog density	16.4	fog
FOG_START	R	**GetFloatv**	0.0	Linear fog start	16.4	fog
FOG_END	R	**GetFloatv**	1.0	Linear fog end	16.4	fog
FOG_MODE	E	**GetIntegerv**	EXP	Fog mode	16.4	fog
FOG	B	**IsEnabled**	FALSE	True if fog enabled	16.4	fog/enable
FOG_COORD_SRC	E	**GetIntegerv**	FRAGMENT_DEPTH	Source of coordinate for fog calculation	16.4	fog
COLOR_SUM	B	**IsEnabled**	FALSE	True if color sum enabled	16.3	fog/enable
SHADE_MODEL	E	**GetIntegerv**	SMOOTH	**ShadeModel** setting	13.6	lighting
CLAMP_VERTEX_COLOR	E	**GetIntegerv**	TRUE	Vertex color clamping	13.1	lighting/enable
CLAMP_FRAGMENT_COLOR	E	**GetIntegerv**	FIXED_ONLY	Fragment color clamping	18.1.1	color-buffer/enable
CLAMP_READ_COLOR	E	**GetIntegerv**	FIXED_ONLY	Read color clamping	18.2.8	color-buffer/enable
PROVOKING_VERTEX	E	**GetIntegerv**	LAST_VERTEX_CONVENTION	Provoking vertex convention	13.6	lighting

Table 23.12: Lighting(see also table 12.1 for defaults)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
LIGHTING	B	**IsEnabled**	FALSE	True if lighting is enabled	12.2.1	lighting/enable
COLOR_MATERIAL	B	**IsEnabled**	FALSE	True if color tracking is enabled	12.2.3	lighting/enable
COLOR_MATERIAL_PARAMETER	E	**GetIntegerv**	AMBIENT_AND_DIFFUSE	Material properties tracking current color	12.2.3	lighting
COLOR_MATERIAL_FACE	E	**GetIntegerv**	FRONT_AND_BACK	Face(s) affected by color tracking	12.2.3	lighting
AMBIENT	$2 \times C$	**GetMaterialfv**	(0.2,0.2,0.2,1.0)	Ambient material color	12.2.1	lighting
DIFFUSE	$2 \times C$	**GetMaterialfv**	(0.8,0.8,0.8,1.0)	Diffuse material color	12.2.1	lighting
SPECULAR	$2 \times C$	**GetMaterialfv**	(0.0,0.0,0.0,1.0)	Specular material color	12.2.1	lighting
EMISSION	$2 \times C$	**GetMaterialfv**	(0.0,0.0,0.0,1.0)	Emissive mat. color	12.2.1	lighting
SHININESS	$2 \times R$	**GetMaterialfv**	0.0	Specular exponent of material	12.2.1	lighting
LIGHT_MODEL_AMBIENT	C	**GetFloatv**	(0.2,0.2,0.2,1.0)	Ambient scene color	12.2.1	lighting
LIGHT_MODEL_LOCAL_VIEWER	B	**GetBooleanv**	FALSE	Viewer is local	12.2.1	lighting
LIGHT_MODEL_TWO_SIDE	B	**GetBooleanv**	FALSE	Use two-sided lighting	12.2.1	lighting
LIGHT_MODEL_COLOR_CONTROL	E	**GetIntegerv**	SINGLE_COLOR	Color control	12.2.1.1	lighting

Table 23.13: Lighting (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
AMBIENT	$8 * \times C$	**GetLightfv**	(0.0,0.0,0.0,1.0)	Ambient intensity of light i	12.2.1	lighting
DIFFUSE	$8 * \times C$	**GetLightfv**	See table 12.1	Diffuse intensity of light i	12.2.1	lighting
SPECULAR	$8 * \times C$	**GetLightfv**	See table 12.1	Specular intensity of light i	12.2.1	lighting
POSITION	$8 * \times P$	**GetLightfv**	(0.0,0.0,1.0,0.0)	Position of light i	12.2.1	lighting
CONSTANT_ATTENUATION	$8 * \times R^+$	**GetLightfv**	1.0	Constant atten. factor	12.2.1	lighting
LINEAR_ATTENUATION	$8 * \times R^+$	**GetLightfv**	0.0	Linear atten. factor	12.2.1	lighting
QUADRATIC_ATTENUATION	$8 * \times R^+$	**GetLightfv**	0.0	Quadratic atten. factor	12.2.1	lighting
SPOT_DIRECTION	$8 * \times D$	**GetLightfv**	(0.0,0.0,-1.0)	Spotlight direction of light i	12.2.1	lighting
SPOT_EXPONENT	$8 * \times R^+$	**GetLightfv**	0.0	Spotlight exponent of light i	12.2.1	lighting
SPOT_CUTOFF	$8 * \times R^+$	**GetLightfv**	180.0	Spot. angle of light i	12.2.1	lighting
LIGHTi	$8 * \times B$	**IsEnabled**	FALSE	True if light i enabled	12.2.1	lighting/enable
COLOR_INDEXES	$2 \times 3 \times R$	**GetMaterialfv**	0,1,1	a_m, d_m, and s_m for color index lighting	12.2.1	lighting

Table 23.14: Rasterization

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RASTERIZER_DISCARD	B	**IsEnabled**	FALSE	Discard primitives before rasterization	14.1	–
POINT_SIZE	R^+	**GetFloatv**	1.0	Point size	14.4	point
POINT_SMOOTH	B	**IsEnabled**	FALSE	Point antialiasing on	14.4	point/enable
POINT_SPRITE	B	**IsEnabled**	FALSE	Point sprite enable	14.4	point/enable
POINT_SIZE_MIN	R^+	**GetFloatv**	0.0	Attenuated min point size	14.4	point
POINT_SIZE_MAX	R^+	**GetFloatv**	†	Attenuated max point size. † Max. of the impl. dependent max. aliased and smooth point sizes	14.4	point
POINT_FADE_THRESHOLD_SIZE	R^+	**GetFloatv**	1.0	Threshold for alpha attenuation	14.4	point
POINT_DISTANCE_ATTENUATION	$3 \times R^+$	**GetFloatv**	1,0,0	Attenuation coefficients	14.4	point
POINT_SPRITE_COORD_ORIGIN	E	**GetIntegerv**	UPPER_LEFT	Origin orientation for point sprites	14.4	point
LINE_WIDTH	R^+	**GetFloatv**	1.0	Line width	14.5	line
LINE_SMOOTH	B	**IsEnabled**	FALSE	Line antialiasing on	14.5	line/enable
LINE_STIPPLE_PATTERN	Z^+	**GetIntegerv**	1's	Line stipple	14.5.2.1	line
LINE_STIPPLE_REPEAT	Z^+	**GetIntegerv**	1	Line stipple repeat	14.5.2.1	line
LINE_STIPPLE	B	**IsEnabled**	FALSE	Line stipple enable	14.5.2.1	line/enable

Table 23.15: Rasterization (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CULL_FACE	B	**IsEnabled**	FALSE	Polygon culling enabled	14.6.1	polygon/enable
CULL_FACE_MODE	E	**GetIntegerv**	BACK	Cull front-/back-facing polygons	14.6.1	polygon
FRONT_FACE	E	**GetIntegerv**	CCW	Polygon frontface CW/CCW indicator	14.6.1	polygon
POLYGON_SMOOTH	B	**IsEnabled**	FALSE	Polygon antialiasing on	14.6	polygon/enable
POLYGON_MODE	$2 \times E$	**GetIntegerv**	FILL	Polygon rasterization mode (front & back)	14.6.4	polygon
POLYGON_OFFSET_CLAMP	R	**GetFloatv**	0	Polygon offset clamp	14.6.5	polygon
POLYGON_OFFSET_FACTOR	R	**GetFloatv**	0	Polygon offset factor	14.6.5	polygon
POLYGON_OFFSET_UNITS	R	**GetFloatv**	0	Polygon offset units	14.6.5	polygon
POLYGON_OFFSET_POINT	B	**IsEnabled**	FALSE	Polygon offset enable for POINT mode rasterization	14.6.5	polygon/enable
POLYGON_OFFSET_LINE	B	**IsEnabled**	FALSE	Polygon offset enable for LINE mode rasterization	14.6.5	polygon/enable
POLYGON_OFFSET_FILL	B	**IsEnabled**	FALSE	Polygon offset enable for FILL mode rasterization	14.6.5	polygon/enable
–	I	**GetPolygonStipple**	1's	Polygon stipple	14.6	polygon-stipple
POLYGON_STIPPLE	B	**IsEnabled**	FALSE	Polygon stipple enable	14.6.2	polygon/enable

Table 23.16: Multisampling

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MULTISAMPLE	B	**IsEnabled**	TRUE	Multisample rasterization	14.3.1	multisample/enable
SAMPLE_ALPHA_TO_COVERAGE	B	**IsEnabled**	FALSE	Modify coverage from alpha	17.3.1	multisample/enable
SAMPLE_ALPHA_TO_ONE	B	**IsEnabled**	FALSE	Set alpha to max	17.3.1	multisample/enable
SAMPLE_COVERAGE	B	**IsEnabled**	FALSE	Mask to modify coverage	14.9.3	multisample/enable
SAMPLE_COVERAGE_VALUE	R^+	**GetFloatv**	1	Coverage mask value	14.9.3	multisample
SAMPLE_COVERAGE_INVERT	B	**GetBooleanv**	FALSE	Invert coverage mask value	14.9.3	multisample
SAMPLE_SHADING	B	**IsEnabled**	FALSE	Sample shading enable	14.9.3	multisample/enable
MIN_SAMPLE_SHADING_VALUE	R^+	**GetFloatv**	0	Fraction of multisamples to use for sample shading	14.3.1.1	multisample
SAMPLE_MASK	B	**IsEnabled**	FALSE	Additional sample mask	14.9.3	–
SAMPLE_MASK_VALUE	$n \times Z^+$	**GetIntegeri_v**	All bits of all words set	Additional sample mask value	14.9.3	–

Table 23.17: Textures (state per texture unit)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_xD	$8 * \times 3 \times B$	**IsEnabled**	FALSE	True if xD texturing is enabled; x is 1, 2, or 3	16.2	texture/enable
TEXTURE_CUBE_MAP	$8 * \times B$	**IsEnabled**	FALSE	True if cube map texturing is enabled	16.2	texture/enable
TEXTURE_RECTANGLE	$8 * \times B$	**IsEnabled**	FALSE	True if rectangle texturing is enabled	16.2	texture/enable
TEXTURE_BINDING_xD	$80 * \times 3 \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_xD	8.1	texture
TEXTURE_BINDING_1D_ARRAY	$80 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_1D_ARRAY	8.1	texture
TEXTURE_BINDING_2D_ARRAY	$80 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_2D_ARRAY	8.1	texture
TEXTURE_BINDING_CUBE_MAP_ARRAY	$80 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_CUBE_MAP_ARRAY	8.1	texture
TEXTURE_BINDING_RECTANGLE	$80 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_RECTANGLE	8.1	texture
TEXTURE_BINDING_BUFFER	$80 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_BUFFER	8.1	texture
TEXTURE_BINDING_CUBE_MAP	$80 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_CUBE_MAP	8.1	texture
TEXTURE_BINDING_2D_MULTISAMPLE	$80 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_2D_MULTISAMPLE	8.22	–
TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY	$80 * \times Z^+$	**GetIntegerv**	0	Texture object bound to TEXTURE_2D_MULTISAMPLE_ARRAY	8.22	–

Table 23.18: Textures (state per texture unit) (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SAMPLER_BINDING	$80 * \times Z^+$	**GetIntegerv**	0	Sampler object bound to active texture unit	8.2	–
TEXTURE_xD	$0 * \times 3 \times I$	**GetTexImage**	See ch. 8	xD texture image at l.o.d. i	8	–
TEXTURE_1D_ARRAY	$0 * \times I$	**GetTexImage**	See ch. 8	1D texture image at row i	8	–
TEXTURE_2D_ARRAY	$0 * \times I$	**GetTexImage**	See ch. 8	2D texture image at slice i	8	–
TEXTURE_CUBE_MAP_ARRAY	$0 * \times I$	**GetTexImage**	See ch. 8	Cube map array texture image at l.o.d. i	8	–
TEXTURE_RECTANGLE	$0 * \times I$	**GetTexImage**	See ch. 8	Rectangle texture image at l.o.d. zero	8	–
TEXTURE_CUBE_MAP_POSITIVE_X	$0 * \times I$	**GetTexImage**	See sec. 8.5	$+x$ face cube map texture image at l.o.d. i	8.5	–
TEXTURE_CUBE_MAP_NEGATIVE_X	$0 * \times I$	**GetTexImage**	See sec. 8.5	$-x$ face cube map texture image at l.o.d. i	8.5	–
TEXTURE_CUBE_MAP_POSITIVE_Y	$0 * \times I$	**GetTexImage**	See sec. 8.5	$+y$ face cube map texture image at l.o.d. i	8.5	–
TEXTURE_CUBE_MAP_NEGATIVE_Y	$0 * \times I$	**GetTexImage**	See sec. 8.5	$-y$ face cube map texture image at l.o.d. i	8.5	–
TEXTURE_CUBE_MAP_POSITIVE_Z	$0 * \times I$	**GetTexImage**	See sec. 8.5	$+z$ face cube map texture image at l.o.d. i	8.5	–
TEXTURE_CUBE_MAP_NEGATIVE_Z	$0 * \times I$	**GetTexImage**	See sec. 8.5	$-z$ face cube map texture image at l.o.d. i	8.5	–

Table 23.19: Textures (state per texture object)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_SWIZZLE_R	E	**GetTexParameteriv**	RED	Red component swizzle	8.10	texture
TEXTURE_SWIZZLE_G	E	**GetTexParameteriv**	GREEN	Green component swizzle	8.10	texture
TEXTURE_SWIZZLE_B	E	**GetTexParameteriv**	BLUE	Blue component swizzle	8.10	texture
TEXTURE_SWIZZLE_A	E	**GetTexParameteriv**	ALPHA	Alpha component swizzle	8.10	texture
TEXTURE_BORDER_COLOR	C	**GetTexParameterfv**	0.0,0.0,0.0,0.0	Border color	8	texture
TEXTURE_MIN_FILTER	E	**GetTexParameteriv**	See sec. 8.22	Minification function	8.14	texture
TEXTURE_MAG_FILTER	E	**GetTexParameteriv**	LINEAR	Magnification function	8.15	texture
TEXTURE_WRAP_S	E	**GetTexParameteriv**	See sec. 8.22	Texcoord s wrap mode	8.14.2	texture
TEXTURE_WRAP_T	E	**GetTexParameteriv**	See sec. 8.22	Texcoord t wrap mode (2D, 3D, cube map textures only)	8.14.2	texture
TEXTURE_WRAP_R	E	**GetTexParameteriv**	See sec. 8.22	Texcoord r wrap mode (3D textures only)	8.14.2	texture
TEXTURE_TARGET	E	**GetTextureParameteriv**	NONE	Target of texture object	8.11	–
TEXTURE_PRIORITY	$R^{[0,1]}$	**GetTexParameterfv**	1	Texture object priority	8.1	texture
TEXTURE_RESIDENT	B	**GetTexParameteriv**	See sec.8.1	Texture residency	8.1	texture
TEXTURE_MIN_LOD	R	**GetTexParameterfv**	-1000	Min level-of-detail	8	texture
TEXTURE_MAX_LOD	R	**GetTexParameterfv**	1000	Max. level-of-detail	8	texture
TEXTURE_BASE_LEVEL	Z^+	**GetTexParameterfv**	0	Base texture array	8	texture
TEXTURE_MAX_LEVEL	Z^+	**GetTexParameterfv**	1000	Max. texture array level	8	texture
TEXTURE_LOD_BIAS	R	**GetTexParameterfv**	0.0	Texture level-of-detail bias ($bias_{texobj}$)	8.14	texture

Table 23.20: Textures (state per texture object) (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DEPTH_TEXTURE_MODE	E	**GetTexParameteriv**	LUMINANCE	Depth texture mode	8.12	texture
DEPTH_STENCIL_TEXTURE_MODE	E	**GetTexParameteriv**	DEPTH_COMPONENT	Depth stencil texture mode	8.16	–
TEXTURE_COMPARE_MODE	E	**GetTexParameteriv**	NONE	Comparison mode	8.23	texture
TEXTURE_COMPARE_FUNC	E	**GetTexParameteriv**	LEQUAL	Comparison function	8.23	texture
GENERATE_MIPMAP	B	**GetTexParameteriv**	FALSE	Automatic mipmap generation enabled	8.14	texture
IMAGE_FORMAT_COMPATIBILITY_TYPE	E	**GetTexParameteriv**	See sec. 8.26	Compatibility rules for texture use with image units	8.26	texture
TEXTURE_IMMUTABLE_FORMAT	B	**GetTexParameteriv**	FALSE	Size and format immutable	8.19	–
TEXTURE_IMMUTABLE_LEVELS	Z^+	**GetTexParameteriv**	0	Storage no. of levels	8.18	texture
TEXTURE_VIEW_MIN_LEVEL	Z^+	**GetTexParameteriv**	0	View base texture level	8.18	texture
TEXTURE_VIEW_NUM_LEVELS	Z^+	**GetTexParameteriv**	0	View no. of texture levels	8.18	texture
TEXTURE_VIEW_MIN_LAYER	Z^+	**GetTexParameteriv**	0	View min array layer	8.18	texture
TEXTURE_VIEW_NUM_LAYERS	Z^+	**GetTexParameteriv**	0	View no. of array layers	8.18	texture
–	S	**GetObjectLabel**	empty	Debug label	20.9	–
TEXTURE_MAX_ANISOTROPY	R	**GetTexParameterfv**	1.0	Maximum degree of anisotropy	8.14	texture

Table 23.21: Textures (state per texture image)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_WIDTH	Z^+	**GetTexLevelParameteriv**	0	Specified width	8	–
TEXTURE_HEIGHT	Z^+	**GetTexLevelParameteriv**	0	Specified height (2D/3D)	8	–
TEXTURE_DEPTH	Z^+	**GetTexLevelParameteriv**	0	Specified depth (3D)	8	–
TEXTURE_BORDER	Z^+	**GetTexLevelParameteriv**	0	Specified border width	8	–
TEXTURE_SAMPLES	Z^+	**GetTexLevelParameteriv**	0	No. of samples per texel	8.8	–
TEXTURE_FIXED_SAMPLE_LOCATIONS	B	**GetTexLevelParameteriv**	TRUE	Whether the image uses a fixed sample pattern	8.8	–
TEXTURE_INTERNAL_FORMAT (TEXTURE_COMPONENTS)	E	**GetTexLevelParameteriv**	1 or LUMINANCE8	Internal format (see section 8.22	8	–
TEXTURE_x_SIZE	$8 \times Z^+$	**GetTexLevelParameteriv**	0	Component resolution (x is RED, GREEN, BLUE, ALPHA, LUMINANCE, INTENSITY, DEPTH, or STENCIL)	8	–
TEXTURE_SHARED_SIZE	Z^+	**GetTexLevelParameteriv**	0	Shared exponent field resolution	8	–
TEXTURE_x_TYPE	E	**GetTexLevelParameteriv**	NONE	Component type (x is RED, GREEN, BLUE, ALPHA, LUMINANCE, INTENSITY, or DEPTH)	8.11	–

Table 23.22: Textures (state per texture image) (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_COMPRESSED	B	**GetTexLevelParameteriv**	FALSE	True if image has a compressed internal format	8.7	–
TEXTURE_COMPRESSED_IMAGE_SIZE	Z^+	**GetTexLevelParameteriv**	0	Size (in ubytes) of compressed image	8.7	–
TEXTURE_BUFFER_DATA_STORE_BINDING	Z^+	**GetTexLevelParameteriv**	0	Buffer object bound as the data store for the active image unit's buffer texture	8.1	texture
TEXTURE_BUFFER_OFFSET	$n \times Z$	**GetTexLevelParameteriv**	0	Offset into buffer's data store used for the active image unit's buffer texture	8.9	–
TEXTURE_BUFFER_SIZE	$n \times Z$	**GetTexLevelParameteriv**	0	Size of the buffer's data store used for the active image unit's buffer texture	8.9	–

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BORDER_COLOR	C	**GetSamplerParameterfv**	0.0,0.0,0.0,0.0	Border color	8	–
TEXTURE_COMPARE_FUNC	E	**GetSamplerParameteriv**	LEQUAL	Comparison function	8.23	–
TEXTURE_COMPARE_MODE	E	**GetSamplerParameteriv**	NONE	Comparison mode	8.23	–
TEXTURE_LOD_BIAS	R	**GetSamplerParameterfv**	0.0	Texture level-of-detail bias ($bias_{texobj}$)	8.14	–
TEXTURE_MAX_LOD	R	**GetSamplerParameterfv**	1000	Max. level-of-detail	8	–
TEXTURE_MAG_FILTER	E	**GetSamplerParameteriv**	LINEAR	Magnification function	8.15	–
TEXTURE_MIN_FILTER	E	**GetSamplerParameteriv**	NEAREST_MIPMAP_LINEAR	Minification function	8.14	–
TEXTURE_MIN_LOD	R	**GetSamplerParameterfv**	-1000	Min level-of-detail	8	–
TEXTURE_MAX_ANISOTROPY	R	**GetSamplerParameterfv**	1.0	Maximum degree of anisotropy	8.14	texture
TEXTURE_WRAP_S	E	**GetSamplerParameteriv**	REPEAT	Texcoord s wrap mode	8.14.2	–
TEXTURE_WRAP_T	E	**GetSamplerParameteriv**	REPEAT	Texcoord t wrap mode (2D, 3D, cube map textures only)	8.14.2	–
TEXTURE_WRAP_R	E	**GetSamplerParameteriv**	REPEAT	Texcoord r wrap mode (3D textures only)	8.14.2	–
–	S	**GetObjectLabel**	empty	Debug label	20.9	–

Table 23.24: Texture Environment and Generation

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_TEXTURE	E	**GetIntegerv**	TEXTURE0	Active texture unit selector	10.2	texture
COORD_REPLACE	$2 * \times B$	**GetTexEnviv**	FALSE	Coordinate replacement enable	14.4	point
TEXTURE_ENV_MODE	$2 * \times E$	**GetTexEnviv**	MODULATE	Texture application function	16.1	texture
TEXTURE_ENV_COLOR	$2 * \times C$	**GetTexEnvfv**	0.0,0.0,0.0,0.0	Texture environment color	16.1	texture
TEXTURE_LOD_BIAS	$2 * \times R$	**GetTexEnvfv**	0.0	Texture level-of-detail bias $bias_{texunit}$	8.14	texture
TEXTURE_GEN_x	$2 * \times 4 \times B$	**IsEnabled**	FALSE	Texgen enabled (x is S, T, R, or Q)	12.1.3	texture/enable
EYE_PLANE	$2 * \times 4 \times R^4$	**GetTexGenfv**	See sec. 12.1.3	Texgen plane equation coefficients (for S, T, R, and Q)	12.1.3	texture
OBJECT_PLANE	$2 * \times 4 \times R^4$	**GetTexGenfv**	See sec. 12.1.3	Texgen object linear coefficients (for S, T, R, and Q)	12.1.3	texture
TEXTURE_GEN_MODE	$2 * \times 4 \times E$	**GetTexGeniv**	EYE_LINEAR	Function used for texgen (for S, T, R, and Q	12.1.3	texture
COMBINE_RGB	$2 * \times E$	**GetTexEnviv**	MODULATE	RGB combiner function	16.1	texture
COMBINE_ALPHA	$2 * \times E$	**GetTexEnviv**	MODULATE	Alpha combiner function	16.1	texture

Table 23.25: Texture Environment and Generation (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SRC0_RGB	$2 * \times E$	**GetTexEnviv**	TEXTURE	RGB source 0	16.1	texture
SRC1_RGB	$2 * \times E$	**GetTexEnviv**	PREVIOUS	RGB source 1	16.1	texture
SRC2_RGB	$2 * \times E$	**GetTexEnviv**	CONSTANT	RGB source 2	16.1	texture
SRC0_ALPHA	$2 * \times E$	**GetTexEnviv**	TEXTURE	Alpha source 0	16.1	texture
SRC1_ALPHA	$2 * \times E$	**GetTexEnviv**	PREVIOUS	Alpha source 1	16.1	texture
SRC2_ALPHA	$2 * \times E$	**GetTexEnviv**	CONSTANT	Alpha source 2	16.1	texture
OPERAND0_RGB	$2 * \times E$	**GetTexEnviv**	SRC_COLOR	RGB operand 0	16.1	texture
OPERAND1_RGB	$2 * \times E$	**GetTexEnviv**	SRC_COLOR	RGB operand 1	16.1	texture
OPERAND2_RGB	$2 * \times E$	**GetTexEnviv**	SRC_ALPHA	RGB operand 2	16.1	texture
OPERAND0_ALPHA	$2 * \times E$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 0	16.1	texture
OPERAND1_ALPHA	$2 * \times E$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 1	16.1	texture
OPERAND2_ALPHA	$2 * \times E$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 2	16.1	texture
RGB_SCALE	$2 * \times R^3$	**GetTexEnvfv**	1.0	RGB post-combiner scaling	16.1	texture
ALPHA_SCALE	$2 * \times R^3$	**GetTexEnvfv**	1.0	Alpha post-combiner scaling	16.1	texture

Table 23.26: Pixel Operations

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SCISSOR_TEST	$16 * \times B$	**IsEnabledi**	FALSE	Scissoring enabled	14.9.2	scissor/enable
SCISSOR_BOX	$16 * \times 4 \times Z$	**GetIntegeri_v**	See sec. 14.9.2	Scissor box	14.9.2	scissor
ALPHA_TEST	B	**IsEnabled**	FALSE	Alpha test enabled	17.3.2	color-buffer/enable
ALPHA_TEST_FUNC	E	**GetIntegerv**	ALWAYS	Alpha test function	17.3.2	color-buffer
ALPHA_TEST_REF	R^+	**GetFloatv**	0	Alpha test reference value	17.3.2	color-buffer
STENCIL_TEST	B	**IsEnabled**	FALSE	Stenciling enabled	17.3.3	stencil-buffer/enable
STENCIL_FUNC	E	**GetIntegerv**	ALWAYS	Front stencil function	17.3.3	stencil-buffer
STENCIL_VALUE_MASK	Z^+	**GetIntegerv**	See sec. 17.3.3	Front stencil mask	17.3.3	stencil-buffer
STENCIL_REF	Z^+	**GetIntegerv**	0	Front stencil reference value	17.3.3	stencil-buffer
STENCIL_FAIL	E	**GetIntegerv**	KEEP	Front stencil fail action	17.3.3	stencil-buffer
STENCIL_PASS_DEPTH_FAIL	E	**GetIntegerv**	KEEP	Front stencil depth buffer fail action	17.3.3	stencil-buffer
STENCIL_PASS_DEPTH_PASS	E	**GetIntegerv**	KEEP	Front stencil depth buffer pass action	17.3.3	stencil-buffer
STENCIL_BACK_FUNC	E	**GetIntegerv**	ALWAYS	Back stencil function	17.3.3	stencil-buffer
STENCIL_BACK_VALUE_MASK	Z^+	**GetIntegerv**	See sec. 17.3.3	Back stencil mask	17.3.3	stencil-buffer
STENCIL_BACK_REF	Z^+	**GetIntegerv**	0	Back stencil reference value	17.3.3	stencil-buffer
STENCIL_BACK_FAIL	E	**GetIntegerv**	KEEP	Back stencil fail action	17.3.3	stencil-buffer
STENCIL_BACK_PASS_DEPTH_FAIL	E	**GetIntegerv**	KEEP	Back stencil depth buffer fail action	17.3.3	stencil-buffer
STENCIL_BACK_PASS_DEPTH_PASS	E	**GetIntegerv**	KEEP	Back stencil depth buffer pass action	17.3.3	stencil-buffer
DEPTH_TEST	B	**IsEnabled**	FALSE	Depth buffer enabled	17.3.4	depth-buffer/enable
DEPTH_FUNC	E	**GetIntegerv**	LESS	Depth buffer test function	17.3.4	depth-buffer

Table 23.27: Pixel Operations (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
BLEND	$8 * \times B$	**IsEnabledi**	FALSE	Blending enabled for draw buffer i	17.3.6	color-buffer/enable
BLEND_SRC_RGB	$8 * \times E$	**GetIntegeri_v**	ONE	Blending source RGB function for draw buffer i	17.3.6	color-buffer
BLEND_SRC_ALPHA	$8 * \times E$	**GetIntegeri_v**	ONE	Blending source A function for draw buffer i	17.3.6	color-buffer
BLEND_DST_RGB	$8 * \times E$	**GetIntegeri_v**	ZERO	Blending dest. RGB function for draw buffer i	17.3.6	color-buffer
BLEND_DST_ALPHA	$8 * \times E$	**GetIntegeri_v**	ZERO	Blending dest. A function for draw buffer i	17.3.6	color-buffer
BLEND_EQUATION_RGB	$8 * \times E$	**GetIntegeri_v**	FUNC_ADD	RGB blending equation for draw buffer i	17.3.6	color-buffer
BLEND_EQUATION_ALPHA	$8 * \times E$	**GetIntegeri_v**	FUNC_ADD	Alpha blending equation for draw buffer i	17.3.6	color-buffer
BLEND_COLOR	C	**GetFloatv**	0.0,0.0,0.0,0.0	Constant blend color	17.3.6	color-buffer
FRAMEBUFFER_SRGB	B	**IsEnabled**	FALSE	sRGB update and blending enable	17.3.6	color-buffer/enable
DITHER	B	**IsEnabled**	TRUE	Dithering enabled	17.3.8	color-buffer/enable
INDEX_LOGIC_OP	B	**IsEnabled**	FALSE	Index logic op enabled	17.3.9	color-buffer/enable
COLOR_LOGIC_OP	B	**IsEnabled**	FALSE	Color logic op enabled	17.3.9	color-buffer/enable
LOGIC_OP_MODE	E	**GetIntegerv**	COPY	Logic op function	17.3.9	color-buffer

Table 23.28: Framebuffer Control

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
INDEX_WRITEMASK	Z^+	**GetIntegerv**	1's	Color index writemask	17.4.2	color-buffer
COLOR_WRITEMASK	$8 * \times 4 \times B$	**GetBooleani_v**	(TRUE,TRUE,TRUE,TRUE)	Color write enables (R,G,B,A) for draw buffer i	17.4.2	color-buffer
DEPTH_WRITEMASK	B	**GetBooleanv**	TRUE	Depth buffer enabled for writing	17.4.2	depth-buffer
STENCIL_WRITEMASK	Z^+	**GetIntegerv**	1's	Front stencil buffer writemask	17.4.2	stencil-buffer
STENCIL_BACK_WRITEMASK	Z^+	**GetIntegerv**	1's	Back stencil buffer writemask	17.4.2	stencil-buffer
COLOR_CLEAR_VALUE	C	**GetFloatv**	0.0,0.0,0.0,0.0	Color buffer clear value (RGBA mode)	17.4.3	color-buffer
INDEX_CLEAR_VALUE	CI	**GetFloatv**	0	Color buffer clear value (color index mode)	17.4.3	color-buffer
DEPTH_CLEAR_VALUE	R^+	**GetFloatv**	1	Depth buffer clear value	17.4.3	depth-buffer
STENCIL_CLEAR_VALUE	Z^+	**GetIntegerv**	0	Stencil clear value	17.4.3	stencil-buffer
ACCUM_CLEAR_VALUE	$4 \times R^+$	**GetFloatv**	0	Accumulation buffer clear value	17.4.3	accum-buffer

Table 23.29: Framebuffer (state per target binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DRAW_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to `DRAW_FRAMEBUFFER`	9.2	–
READ_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to `READ_FRAMEBUFFER`	9.2	–

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
DRAW_BUFFERi	$8 * \times E$	**GetIntegerv**	See sec. 17.4.1	Draw buffer selected for color output i	17.4.1	color-buffer
READ_BUFFER	E	**GetIntegerv**	See sec. 18.2	Read source buffer †	18.2	pixel
FRAMEBUFFER_DEFAULT_WIDTH	Z^+	**GetFramebuffer-Parameteriv**	0	Default width of framebuffer w/o attachments	9.2.1	–
FRAMEBUFFER_DEFAULT_HEIGHT	Z^+	**GetFramebuffer-Parameteriv**	0	Default height of framebuffer w/o attachments	9.2.1	–
FRAMEBUFFER_DEFAULT_LAYERS	Z^+	**GetFramebuffer-Parameteriv**	0	Default layer count of framebuffer w/o attachments	9.2.1	–
FRAMEBUFFER_DEFAULT_SAMPLES	Z^+	**GetFramebuffer-Parameteriv**	0	Default sample count of framebuffer w/o attachments	9.2.1	–
FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS	B	**GetFramebuffer-Parameteriv**	FALSE	Default sample location pattern of framebuffer w/o attachments	9.2.1	–
–	S	**GetObjectLabel**	empty	Debug label	20.9	–

Table 23.30: Framebuffer (state per framebuffer object)
† This state is queried from the currently bound read framebuffer.

Table 23.31: Framebuffer (state per attachment point)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE	E	**GetFramebuffer-Attachment-Parameteriv**	NONE	Type of image attached to framebuffer attachment point	9.2.2	–
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME	Z^+	**GetFramebuffer-Attachment-Parameteriv**	0	Name of object attached to framebuffer attachment point	9.2.2	–
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL	Z^+	**GetFramebuffer-Attachment-Parameteriv**	0	Mipmap level of texture image attached, if object attached is texture	9.2.8	–
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE	E	**GetFramebuffer-Attachment-Parameteriv**	NONE	Cubemap face of texture image attached, if object attached is cubemap texture	9.2.8	–
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER	Z	**GetFramebuffer-Attachment-Parameteriv**	0	Layer of texture image attached, if object attached is 3D texture	9.2.8	–
FRAMEBUFFER_ATTACHMENT_LAYERED	B	**GetFramebuffer-Attachment-Parameteriv**	FALSE	Framebuffer attachment is layered	9.8	–
FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING	E	**GetFramebuffer-Attachment-Parameteriv**	-	Encoding of components in the attached image	9.2.3	–
FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE	E	**GetFramebuffer-Attachment-Parameteriv**	-	Data type of components in the attached image	9.2.3	–
FRAMEBUFFER_ATTACHMENT_x_SIZE	Z^+	**GetFramebuffer-Attachment-Parameteriv**	-	Size in bits of attached image's x component; x is RED, GREEN, BLUE, ALPHA, DEPTH, or STENCIL	9.2.3	–

Table 23.32: Renderbuffer (state per target and binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RENDERBUFFER_BINDING	Z	**GetIntegerv**	0	Renderbuffer object bound to `RENDERBUFFER`	9.2.4	–

Table 23.33: Renderbuffer (state per renderbuffer object)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RENDERBUFFER_WIDTH	Z^+	**GetRenderbufferParameteriv**	0	Width of renderbuffer	9.2.4	–
RENDERBUFFER_HEIGHT	Z^+	**GetRenderbufferParameteriv**	0	Height of renderbuffer	9.2.4	–
RENDERBUFFER_INTERNAL_FORMAT	E	**GetRenderbufferParameteriv**	RGBA	Internal format of renderbuffer	9.2.4	–
RENDERBUFFER_RED_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's red component	9.2.4	–
RENDERBUFFER_GREEN_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's green component	9.2.4	–
RENDERBUFFER_BLUE_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's blue component	9.2.4	–
RENDERBUFFER_ALPHA_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's alpha component	9.2.4	–
RENDERBUFFER_DEPTH_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's depth component	9.2.4	–
RENDERBUFFER_STENCIL_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's stencil component	9.2.4	–
RENDERBUFFER_SAMPLES	Z^+	**GetRenderbufferParameteriv**	0	No. of samples	9.2.4	–
–	S	**GetObjectLabel**	empty	Debug label	20.9	–

Table 23.34: Pixels

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNPACK_SWAP_BYTES	B	**GetBooleanv**	FALSE	Value of UNPACK_SWAP_BYTES	8.4.1	pixel-store
UNPACK_LSB_FIRST	B	**GetBooleanv**	FALSE	Value of UNPACK_LSB_FIRST	8.4.1	pixel-store
UNPACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of UNPACK_IMAGE_HEIGHT	8.4.1	pixel-store
UNPACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_IMAGES	8.4.1	pixel-store
UNPACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of UNPACK_ROW_LENGTH	8.4.1	pixel-store
UNPACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_ROWS	8.4.1	pixel-store
UNPACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_PIXELS	8.4.1	pixel-store
UNPACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of UNPACK_ALIGNMENT	8.4.1	pixel-store
UNPACK_COMPRESSED_BLOCK_WIDTH	Z^+	**GetIntegerv**	0	Value of UNPACK_COMPRESSED_BLOCK_WIDTH	8.4.1	–
UNPACK_COMPRESSED_BLOCK_HEIGHT	Z^+	**GetIntegerv**	0	Value of UNPACK_COMPRESSED_BLOCK_HEIGHT	8.4.1	–
UNPACK_COMPRESSED_BLOCK_DEPTH	Z^+	**GetIntegerv**	0	Value of UNPACK_COMPRESSED_BLOCK_DEPTH	8.4.1	–
UNPACK_COMPRESSED_BLOCK_SIZE	Z^+	**GetIntegerv**	0	Value of UNPACK_COMPRESSED_BLOCK_SIZE	8.4.1	–
PIXEL_UNPACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel unpack buffer binding	6.7	pixel-store

Table 23.35: Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
PACK_SWAP_BYTES	B	**GetBooleanv**	FALSE	Value of PACK_SWAP_BYTES	18.2	pixel-store
PACK_LSB_FIRST	B	**GetBooleanv**	FALSE	Value of PACK_LSB_FIRST	18.2	pixel-store
PACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of PACK_IMAGE_HEIGHT	18.2	pixel-store
PACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_IMAGES	18.2	pixel-store
PACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of PACK_ROW_LENGTH	18.2	pixel-store
PACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_ROWS	18.2	pixel-store
PACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_PIXELS	18.2	pixel-store
PACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of PACK_ALIGNMENT	18.2	pixel-store
PACK_COMPRESSED_BLOCK_WIDTH	Z^+	**GetIntegerv**	0	Value of PACK_COMPRESSED_BLOCK_WIDTH	18.2	–
PACK_COMPRESSED_BLOCK_HEIGHT	Z^+	**GetIntegerv**	0	Value of PACK_COMPRESSED_BLOCK_HEIGHT	18.2	–
PACK_COMPRESSED_BLOCK_DEPTH	Z^+	**GetIntegerv**	0	Value of PACK_COMPRESSED_BLOCK_DEPTH	18.2	–
PACK_COMPRESSED_BLOCK_SIZE	Z^+	**GetIntegerv**	0	Value of PACK_COMPRESSED_BLOCK_SIZE	18.2	–
PIXEL_PACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel pack buffer binding	18.2	pixel-store
MAP_COLOR	B	**GetBooleanv**	FALSE	True if colors are mapped	8.4.3	pixel
MAP_STENCIL	B	**GetBooleanv**	FALSE	True if stencil values are mapped	8.4.3	pixel
INDEX_SHIFT	Z	**GetIntegerv**	0	Value of INDEX_SHIFT	8.4.3	pixel
INDEX_OFFSET	Z	**GetIntegerv**	0	Value of INDEX_OFFSET	8.4.3	pixel
x_SCALE	R	**GetFloatv**	1	Value of x_SCALE; x is RED, GREEN, BLUE, ALPHA, or DEPTH	8.4.3	pixel
x_BIAS	R	**GetFloatv**	0	Value of x_BIAS	8.4.3	pixel

Table 23.36: Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
COLOR_TABLE	B	**IsEnabled**	FALSE	True if color table lookup is done	8.4.3	pixel/enable
POST_CONVOLUTION_COLOR_TABLE	B	**IsEnabled**	FALSE	True if post convolution color table lookup is done	8.4.3	pixel/enable
POST_COLOR_MATRIX_COLOR_TABLE	B	**IsEnabled**	FALSE	True if post color matrix color table lookup is done	8.4.3	pixel/enable
COLOR_TABLE	I	**GetColorTable**	*empty*	Color table	8.4.3	–
POST_CONVOLUTION_COLOR_TABLE	I	**GetColorTable**	*empty*	Post convolution color table	8.4.3	–
POST_COLOR_MATRIX_COLOR_TABLE	I	**GetColorTable**	*empty*	Post color matrix color table	8.4.3	–
COLOR_TABLE_FORMAT	$2 \times 3 \times E$	**GetColorTable-Parameteriv**	RGBA	Color tables' internal image format	8.4.3	–
COLOR_TABLE_WIDTH	$2 \times 3 \times Z^+$	**GetColorTable-Parameteriv**	0	Color tables' specified width	8.4.3	–
COLOR_TABLE_x_SIZE	$6 \times 2 \times 3 \times Z^+$	**GetColorTable-Parameteriv**	0	Color table component resolution; x is RED, GREEN, BLUE, ALPHA, LUMINANCE, or INTENSITY	8.4.3	–
COLOR_TABLE_SCALE	$3 \times R^4$	**GetColorTable-Parameterfv**	1,1,1,1	Scale factors applied to color table entries	8.4.3	pixel
COLOR_TABLE_BIAS	$3 \times R^4$	**GetColorTable-Parameterfv**	0,0,0,0	Bias factors applied to color table entries	8.4.3	pixel

Table 23.37: Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CONVOLUTION_1D	B	**IsEnabled**	FALSE	True if 1D convolution is done	8.4.3	pixel/enable
CONVOLUTION_2D	B	**IsEnabled**	FALSE	True if 2D convolution is done	8.4.3	pixel/enable
SEPARABLE_2D	B	**IsEnabled**	FALSE	True if separable 2D convolution is done	8.4.3	pixel/enable
CONVOLUTION_xD	$2 \times I$	**GetConvolution-Filter**	*empty*	Convolution filters; x is 1 or 2	8.4.3	–
SEPARABLE_2D	$2 \times I$	**GetSeparable-Filter**	*empty*	Separable convolution filter	8.4.3	–
CONVOLUTION_BORDER_COLOR	$3 \times C$	**GetConvolution-Parameterfv**	0.0,0.0,0.0,0.0	Convolution border color	8.4.5	pixel
CONVOLUTION_BORDER_MODE	$3 \times E$	**GetConvolution-Parameteriv**	REDUCE	Convolution border mode	8.4.5	pixel
CONVOLUTION_FILTER_SCALE	$3 \times R^4$	**GetConvolution-Parameterfv**	1,1,1,1	Scale factors applied to convolution filter entries	8.4.3	pixel
CONVOLUTION_FILTER_BIAS	$3 \times R^4$	**GetConvolution-Parameterfv**	0,0,0,0	Bias factors applied to convolution filter entries	8.4.3	pixel
CONVOLUTION_FORMAT	$3 \times E$	**GetConvolution-Parameteriv**	RGBA	Convolution filter internal format	8.4.5	–
CONVOLUTION_WIDTH	$3 \times Z^+$	**GetConvolution-Parameteriv**	0	Convolution filter width	8.4.5	–
CONVOLUTION_HEIGHT	$2 \times Z^+$	**GetConvolution-Parameteriv**	0	Convolution filter height	8.4.5	–

Table 23.38: Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
POST_CONVOLUTION_x_SCALE	R	**GetFloatv**	1	Component scale factors after convolution; x is RED, GREEN, BLUE, or ALPHA	8.4.3	pixel
POST_CONVOLUTION_x_BIAS	R	**GetFloatv**	0	Component bias factors after convolution	8.4.3	pixel
POST_COLOR_MATRIX_x_SCALE	R	**GetFloatv**	1	Component scale factors after color matrix	8.4.3	pixel
POST_COLOR_MATRIX_x_BIAS	R	**GetFloatv**	0	Component bias factors after color matrix	8.4.3	pixel
HISTOGRAM	B	**IsEnabled**	FALSE	True if histogramming is enabled	8.4.3	pixel/enable
HISTOGRAM	I	**GetHistogram**	*empty*	Histogram table	8.4.3	–
HISTOGRAM_WIDTH	$2 \times Z^+$	**GetHistogram-Parameteriv**	0	Histogram table width	8.4.3	–
HISTOGRAM_FORMAT	$2 \times E$	**GetHistogram-Parameteriv**	RGBA	Histogram table internal format	8.4.3	–
HISTOGRAM_x_SIZE	$5 \times 2 \times Z^+$	**GetHistogram-Parameteriv**	0	Histogram table component resolution; x is RED, GREEN, BLUE, ALPHA, or LUMINANCE	8.4.3	–
HISTOGRAM_SINK	B	**GetHistogram-Parameteriv**	FALSE	True if histogramming consumes pixel groups	8.4.3	–

Table 23.39: Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MINMAX	B	**IsEnabled**	FALSE	True if minmax is enabled	8.4.3	pixel/enable
MINMAX	R^n	**GetMinmax**	(M,M,M,M),(m,m,m,m)	Minmax table	8.4.3	–
MINMAX_FORMAT	E	**GetMinmax-Parameteriv**	RGBA	Minmax table internal format	8.4.3	–
MINMAX_SINK	B	**GetMinmax-Parameteriv**	FALSE	True if minmax consumes pixel groups	8.4.3	–
ZOOM_X	R	**GetFloatv**	1.0	x zoom factor	18.1	pixel
ZOOM_Y	R	**GetFloatv**	1.0	y zoom factor	18.1	pixel
x	$8 \times 32 * \times R$	**GetPixelMap**	0's	RGBA **PixelMap** translation tables; x is a map name from table 8.3	8.4.3	–
x	$2 \times 32 * \times Z$	**GetPixelMap**	0's	Index **PixelMap** translation tables; x is a map name from table 8.3	8.4.3	–
x_SIZE	Z^+	**GetIntegerv**	1	Size of table x	8.4.3	–

Table 23.40: Evaluators(**GetMap** takes a map name)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ORDER	$9 \times E$	**GetMapiv**	1	1d map order	21.1	–
ORDER	$9 \times 2 \times E$	**GetMapiv**	1,1	2d map orders	21.1	–
COEFF	$9 \times 8 * \times R^n$	**GetMapfv**	See sec. 21.1	1d control points	21.1	–
COEFF	$9 \times 8 * \times 8 * \times R^n$	**GetMapfv**	See sec. 21.1	2d control points	21.1	–
DOMAIN	$9 \times 2 \times R$	**GetMapfv**	See sec. 21.1	1d domain endpoints	21.1	–
DOMAIN	$9 \times 4 \times R$	**GetMapfv**	See sec. 21.1	2d domain endpoints	21.1	–
MAP1_x	$9 \times B$	**IsEnabled**	FALSE	1d map enables: x is map type	21.1	eval/enable
MAP2_x	$9 \times B$	**IsEnabled**	FALSE	2d map enables: x is map type	21.1	eval/enable
MAP1_GRID_DOMAIN	$2 \times R$	**GetFloatv**	0,1	1d grid endpoints	21.1	eval
MAP2_GRID_DOMAIN	$4 \times R$	**GetFloatv**	0,1;0,1	2d grid endpoints	21.1	eval
MAP1_GRID_SEGMENTS	Z^+	**GetFloatv**	1	1d grid divisions	21.1	eval
MAP2_GRID_SEGMENTS	$2 \times Z^+$	**GetFloatv**	1,1	2d grid divisions	21.1	eval
AUTO_NORMAL	B	**IsEnabled**	FALSE	True if automatic normal generation enabled	21.1	eval/enable

Table 23.41: Shader Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SHADER_TYPE	E	**GetShaderiv**	-	Type of shader (see table 7.1	7.1	–
DELETE_STATUS	B	**GetShaderiv**	FALSE	Shader flagged for deletion	7.1	–
COMPILE_STATUS	B	**GetShaderiv**	FALSE	Last compile succeeded	7.1	–
–	S	**GetShaderInfoLog**	Empty string	Info log for shader objects	7.14	–
INFO_LOG_LENGTH	Z^+	**GetShaderiv**	0	Length of info log	7.14	–
–	S	**GetShaderSource**	Empty string	Source code for a shader	7.1	–
SHADER_SOURCE_LENGTH	Z^+	**GetShaderiv**	0	Length of source code	7.14	–
–	S	**GetObjectLabel**	empty	Debug label	20.9	–
SPIR_V_BINARY	B	**GetShaderiv**	FALSE	Shader is associated with a SPIR-V module.	7.2	–

Table 23.42: Program Pipeline Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_PROGRAM	Z^+	**GetProgramPipelineiv**	0	Program object updated by **Uniform*** when PPO bound	7.4	–
VERTEX_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current vertex shader program object	7.4	–
GEOMETRY_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current geometry shader program object	7.4	–
FRAGMENT_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current fragment shader program object	7.4	–
COMPUTE_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current compute shader program object	7.4	–
TESS_CONTROL_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current TCS program object	7.4	–
TESS_EVALUATION_SHADER	Z^+	**GetProgramPipelineiv**	0	Name of current TES program object	7.4	–
VALIDATE_STATUS	B	**GetProgramPipelineiv**	FALSE	Validate status of program pipeline object	7.4	–
–	S	**GetProgramPiplineInfoLog**	empty	Info log for program pipeline object	7.14	–
INFO_LOG_LENGTH	Z^+	**GetProgramPipelineiv**	0	Length of info log	7.4	–
–	S	**GetObjectLabel**	empty	Debug label	20.9	–

Table 23.43: Program Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CURRENT_PROGRAM	Z^+	**GetIntegerv**	0	Name of current program object	7.3	–
PROGRAM_PIPELINE_BINDING	Z^+	**GetIntegerv**	0	Current program pipeline object binding	7.4	–
PROGRAM_SEPARABLE	B	**GetProgramiv**	FALSE	Program object can be bound for separate pipeline stages	7.3	–
DELETE_STATUS	B	**GetProgramiv**	FALSE	Program object deleted	7.3	–
LINK_STATUS	B	**GetProgramiv**	FALSE	Last link attempt succeeded	7.3	–
VALIDATE_STATUS	B	**GetProgramiv**	FALSE	Last validate attempt succeeded	7.3	–
ATTACHED_SHADERS	Z^+	**GetProgramiv**	0	No. of attached shader objects	7.14	–
–	$0 * \times Z^+$	**GetAttachedShaders**	empty	Shader objects attached	7.14	–
–	S	**GetProgramInfoLog**	empty	Info log for program object	7.14	–
INFO_LOG_LENGTH	Z^+	**GetProgramiv**	0	Length of info log	7.3	–
PROGRAM_BINARY_LENGTH	Z^+	**GetProgramiv**	0	Length of program binary	7.5	–
PROGRAM_BINARY_RETRIEVABLE_HINT	B	**GetProgramiv**	FALSE	Retrievable binary hint enabled	7.5	–
–	$0 * \times BMU$	**GetProgramBinary**	–	Binary representation of program	7.5	–
COMPUTE_WORK_GROUP_SIZE	$3 \times Z^+$	**GetProgramiv**	$\{0, \dots\}$	Local work size of a linked compute program	19	–
–	S	**GetObjectLabel**	empty	Debug label	20.9	–

Table 23.44: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_UNIFORMS	Z^+	**GetProgramiv**	0	No. of active uniforms	7.6	–
–	$0 * \times Z$	**GetUniformLocation**	–	Location of active uniforms	7.14	–
–	$0 * \times Z^+$	**GetActiveUniform**	–	Size of active uniform	7.6	–
–	$0 * \times Z^+$	**GetActiveUniform**	–	Type of active uniform	7.6	–
–	$0 * \times$ `char`	**GetActiveUniform**	empty	Name of active uniform	7.6	–
ACTIVE_UNIFORM_MAX_LENGTH	Z^+	**GetProgramiv**	0	Max. active uniform name length	7.14	–
–	–	**GetUniform**	0	Uniform value	7.6	–
ACTIVE_ATTRIBUTES	Z^+	**GetProgramiv**	0	No. of active attributes	11.1.1	–
–	$0 * \times Z$	**GetAttribLocation**	–	Location of active generic attribute	11.1.1	–
–	$0 * \times Z^+$	**GetActiveAttrib**	–	Size of active attribute	11.1.1	–
–	$0 * \times Z^+$	**GetActiveAttrib**	–	Type of active attribute	11.1.1	–
–	$0 * \times$ `char`	**GetActiveAttrib**	empty	Name of active attribute	11.1.1	–
ACTIVE_ATTRIBUTE_MAX_LENGTH	Z^+	**GetProgramiv**	0	Max. active attribute name length	7.14	–

Table 23.45: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
GEOMETRY_VERTICES_OUT	Z^+	**GetProgramiv**	0	Max. no. of output vertices	11.3.4	–
GEOMETRY_INPUT_TYPE	E	**GetProgramiv**	TRIANGLES	Primitive input type	11.3.1	–
GEOMETRY_OUTPUT_TYPE	E	**GetProgramiv**	TRIANGLE_STRIP	Primitive output type	11.3.2	–
GEOMETRY_SHADER_INVOCATIONS	Z^+	**GetProgramiv**	1	No. of times a geom. shader should be executed for each input primitive	11.3.4.2	–
TRANSFORM_FEEDBACK_BUFFER_MODE	E	**GetProgramiv**	INTERLEAVED_ATTRIBS	Transform feedback mode for the program	7.14	–
TRANSFORM_FEEDBACK_VARYINGS	Z^+	**GetProgramiv**	0	No. of outputs to stream to buffer object(s)	7.14	–
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH	Z^+	**GetProgramiv**	0	Max. transform feedback output variable name length	7.14	–
–	Z^+	**GetTransformFeedbackVarying**	-	Size of each transform feedback output variable	11.1.2.1	–
–	Z^+	**GetTransformFeedbackVarying**	-	Type of each transform feedback output variable	11.1.2.1	–
–	$0^+ \times$ char	**GetTransformFeedbackVarying**	-	Name of each transform feedback output variable	11.1.2.1	–

Table 23.46: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_UNIFORM_BLOCKS	Z^+	**GetProgramiv**	0	No. of active uniform blocks in a program	7.6.2	–
ACTIVE_UNIFORM_BLOCK_MAX_-NAME_LENGTH	Z^+	**GetProgramiv**	0	Length of longest active uniform block name	7.6.2	–
UNIFORM_TYPE	$0 * \times E$	**GetActiveUniformsiv**	-	Type of active uniform	7.6.2	–
UNIFORM_SIZE	$0 * \times Z^+$	**GetActiveUniformsiv**	-	Size of active uniform	7.6.2	–
UNIFORM_NAME_LENGTH	$0 * \times Z^+$	**GetActiveUniformsiv**	-	Uniform name length	7.6.2	–
UNIFORM_BLOCK_INDEX	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform block index	7.6.2	–
UNIFORM_OFFSET	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer offset	7.6.2	–

Table 23.47: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNIFORM_ARRAY_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer array stride	7.6.2	–
UNIFORM_MATRIX_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer intra-matrix stride	7.6.2	–
UNIFORM_IS_ROW_MAJOR	$0 * \times B$	**GetActiveUniformsiv**	-	Whether uniform is a row-major matrix	7.6.2	–
UNIFORM_BLOCK_BINDING	Z^+	**GetActive-UniformBlockiv**	0	Uniform buffer binding point associated with the specified uniform block	7.6.2	–
UNIFORM_BLOCK_DATA_SIZE	Z^+	**GetActive-UniformBlockiv**	-	Size of the storage needed to hold this uniform block's data	7.6.2	–
UNIFORM_BLOCK_NAME_LENGTH	Z^+	**GetActive-UniformBlockiv**	-	Uniform block name length	7.6.2	–
UNIFORM_BLOCK_ACTIVE_UNIFORMS	Z^+	**GetActive-UniformBlockiv**	-	Count of active uniforms in the specified uniform block	7.6.2	–
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES	$n \times Z^+$	**GetActive-UniformBlockiv**	-	Array of active uniform indices of the specified uniform block	7.6.2	–
UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER	B	**GetActive-UniformBlockiv**	0	True if uniform block is actively referenced by the vertex stage	7.6.2	–
UNIFORM_BLOCK_REFERENCED_BY_TESS_CONTROL_SHADER	B	**GetActive-UniformBlockiv**	0	True if uniform block is actively referenced by tess. control stage	7.6.2	–
UNIFORM_BLOCK_REFERENCED_BY_TESS_EVALUTION_SHADER	B	**GetActive-UniformBlockiv**	0	True if uniform block is actively referenced by tess. evaluation stage	7.6.2	–

Table 23.48: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNIFORM_BLOCK_REFERENCED_BY_GEOMETRY_SHADER	B	**GetActiveUniformBlockiv**	0	True if uniform block is actively referenced by the geometry stage	7.6.2	–
UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER	B	**GetActiveUniformBlockiv**	0	True if uniform block is actively referenced by the fragment stage	7.6.2	–
UNIFORM_BLOCK_REFERENCED_BY_COMPUTE_SHADER	B	**GetActiveUniformBlockiv**	FALSE	True if uniform block is referenced by the compute stage	7.6.2	–
TESS_CONTROL_OUTPUT_VERTICES	Z^+	**GetProgramiv**	0	Output patch size for tess. control shader	11.2.1	–
TESS_GEN_MODE	E	**GetProgramiv**	QUADS	Base primitive type for tess. prim. generator	11.2.2	–
TESS_GEN_SPACING	E	**GetProgramiv**	EQUAL	Spacing of tess. prim. generator edge subdivision	11.2.2	–
TESS_GEN_VERTEX_ORDER	E	**GetProgramiv**	CCW	Order of vertices in primitives generated by tess. primitive generator	11.2.2	–
TESS_GEN_POINT_MODE	B	**GetProgramiv**	FALSE	Tess. prim. generator emits points?	11.2.2	–

Table 23.49: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS	$6 \times Z^+$	**GetProgramStageiv**	0	No. of subroutine unif. locations in the shader	7.10	–
ACTIVE_SUBROUTINE_UNIFORMS	$6 \times Z^+$	**GetProgramStageiv**	0	No. of subroutine unif. variables in the shader	7.10	–
ACTIVE_SUBROUTINES	$6 \times Z^+$	**GetProgramStageiv**	0	No. of subroutine functions in the shader	7.10	–
ACTIVE_SUBROUTINE_UNIFORM_MAX_LENGTH	$6 \times Z^+$	**GetProgramStageiv**	0	Max. subroutine uniform name length	7.10	–
ACTIVE_SUBROUTINE_MAX_LENGTH	$6 \times Z^+$	**GetProgramStageiv**	0	Max. subroutine name length	7.10	–
NUM_COMPATIBLE_SUBROUTINES	$6 \times 0 * \times Z^+$	**GetActive-SubroutineUniformiv**	-	No. of subroutines compatible with a sub. uniform	7.10	–
COMPATIBLE_SUBROUTINES	$6 \times 0 * \times 0 * \times Z^+$	**GetActive-SubroutineUniformiv**	-	List of subroutines compatible with a sub. uniform	7.10	–
UNIFORM_SIZE	$6 \times 0 * \times Z^+$	**GetActive-SubroutineUniformiv**	-	No. of elements in sub. uniform array	7.10	–
UNIFORM_NAME_LENGTH	$6 \times 0 * \times Z^+$	**GetActive-SubroutineUniformiv**	-	Length of sub. uniform name	7.10	–
–	$6 \times 0 * \times S$	**GetActive-Subroutine-UniformName**	-	Sub. uniform name string	7.10	–
–	$6 \times 0 * \times S$	**GetActive-SubroutineName**	-	Length of subroutine name	7.10	–
–	$6 \times 0 * \times S$	**GetActive-SubroutineName**	-	Subroutine name string	7.10	–

Table 23.50: Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_ATOMIC_COUNTER_BUFFERS	Z^+	**GetProgramiv**	0	No. of active atomic counter buffers (AACBs) used by a program	7.7	–
ATOMIC_COUNTER_BUFFER_BINDING	$n \times Z^+$	**GetActiveAtomicCounterBufferiv**	-	Binding point associated with an AACB	7.7	–
ATOMIC_COUNTER_BUFFER_DATA_SIZE	$n \times Z^+$	**GetActiveAtomicCounterBufferiv**	-	Min size required by an AACB	7.7	–
ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTERS	$n \times Z^+$	**GetActiveAtomicCounterBufferiv**	-	No. of active atomic counters in an AACB	7.7	–
ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTER_INDICES	$m \times n \times Z^+$	**GetActiveAtomicCounterBufferiv**	-	List of active atomic counters in an AACB	7.7	–
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_VERTEX_SHADER	$n \times B$	**GetActiveAtomicCounterBufferiv**	FALSE	AACB has a counter used by vertex shaders	7.7	–
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_TESS_CONTROL_SHADER	$n \times B$	**GetActiveAtomicCounterBufferiv**	FALSE	AACB has a counter used by tess. control shaders	7.7	–
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_TESS_EVALUTION_SHADER	$n \times B$	**GetActiveAtomicCounterBufferiv**	FALSE	AACB has a counter used by tess. evaluation shaders	7.7	–
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_GEOMETRY_SHADER	$n \times B$	**GetActiveAtomicCounterBufferiv**	FALSE	AACB has a counter used by geometry shaders	7.7	–
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_FRAGMENT_SHADER	$n \times B$	**GetActiveAtomicCounterBufferiv**	FALSE	AACB has a counter used by fragment shaders	7.7	–
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_COMPUTE_SHADER	B	**GetActiveAtomicCounterBufferiv**	FALSE	AACB has a counter used by compute shaders	7.7	–
UNIFORM_ATOMIC_COUNTER_BUFFER_INDEX	$n \times Z^+$	**GetActiveUniformsiv**	-	AACB associated with an active uniform	7.7	–

Table 23.51: Program Interface State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_RESOURCES	$n \times Z^+$	**GetProgram-Interfaceiv**	0	No. of active resources on an interface	7.3.1	–
MAX_NAME_LENGTH	$n \times Z^+$	**GetProgram-Interfaceiv**	0	Max. name length for active resources	7.3.1	–
MAX_NUM_ACTIVE_VARIABLES	$n \times Z^+$	**GetProgram-Interfaceiv**	0	Max. no. of active variables for active resources	7.3.1	–
MAX_NUM_COMPATIBLE_SUBROUTINES	$n \times Z^+$	**GetProgram-Interfaceiv**	0	Max. no. of compatible subroutines for subroutine uniforms	7.3.1	–

Table 23.52: Program Object Resource State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_VARIABLES	Z^+	**GetProgram-Resourceiv**	-	List of active variables owned by active resource	7.3.1	–
ARRAY_SIZE	Z^+	**GetProgram-Resourceiv**	-	Active resource array size	7.3.1	–
ARRAY_STRIDE	Z^+	**GetProgram-Resourceiv**	-	Active resource array stride in memory	7.3.1	–
ATOMIC_COUNTER_BUFFER_INDEX	Z^+	**GetProgram-Resourceiv**	-	Index of atomic counter buffer owning resource	7.3.1	–
BLOCK_INDEX	Z^+	**GetProgram-Resourceiv**	-	Index of interface block owning resource	7.3.1	–
BUFFER_BINDING	Z^+	**GetProgram-Resourceiv**	-	Buffer binding assigned to active resource	7.3.1	–
BUFFER_DATA_SIZE	Z^+	**GetProgram-Resourceiv**	-	Min. buffer data size required for resource	7.3.1	–
COMPATIBLE_SUBROUTINES	Z^+	**GetProgram-Resourceiv**	-	List of compatible subroutines for active subroutine uniform	7.3.1	–
IS_PER_PATCH	Z^+	**GetProgram-Resourceiv**	-	Is active input/output a per-patch attribute?	7.3.1	–
IS_ROW_MAJOR	Z^+	**GetProgram-Resourceiv**	-	Active resource stored as a row major matrix?	7.3.1	–
LOCATION	Z^+	**GetProgram-Resourceiv**	-	Location assigned to active resource	7.3.1	–
LOCATION_COMPONENT	Z^+	**GetProgram-Resourceiv**	-	Location component assigned to active resources	7.3.1	–
LOCATION_INDEX	Z^+	**GetProgram-Resourceiv**	-	Location index assigned to active resource	7.3.1	–
MATRIX_STRIDE	Z^+	**GetProgram-Resourceiv**	-	Active resource matrix stride in memory	7.3.1	–

Table 23.53: Program Object Resource State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
NAME_LENGTH	Z^+	**GetProgram-Resourceiv**	-	Length of active resource name	7.3.1	–
NUM_ACTIVE_VARIABLES	Z^+	**GetProgram-Resourceiv**	-	No. of active variables owned by active resource	7.3.1	–
NUM_COMPATIBLE_SUBROUTINES	Z^+	**GetProgram-Resourceiv**	-	No. of compatible subroutines for active subroutine uniform	7.3.1	–
OFFSET	Z^+	**GetProgram-Resourceiv**	-	Active resource offset in memory	7.3.1	–
REFERENCED_BY_VERTEX_SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by vertex shader?	7.3.1	–
REFERENCED_BY_TESS_CONTROL_-SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by tess. control shader?	7.3.1	–
REFERENCED_BY_TESS_EVALUA-TION_SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by tess. evaluation shader?	7.3.1	–
REFERENCED_BY_GEOMETRY_-SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by geometry shader?	7.3.1	–
REFERENCED_BY_FRAGMENT_-SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by fragment shader?	7.3.1	–
REFERENCED_BY_COMPUTE_-SHADER	Z^+	**GetProgram-Resourceiv**	-	Active resource used by compute shader?	7.3.1	–
TOP_LEVEL_ARRAY_SIZE	Z^+	**GetProgram-Resourceiv**	-	Array size of top level shd. storage block member	7.3.1	–
TOP_LEVEL_ARRAY_STRIDE	Z^+	**GetProgram-Resourceiv**	-	Array stride of top level shd. storage block member	7.3.1	–
TYPE	Z^+	**GetProgram-Resourceiv**	-	Active resource data type	7.3.1	–

Table 23.54: Vertex and Geometry Shader State(not part of program objects)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_PROGRAM_TWO_SIDE	B	**IsEnabled**	FALSE	Two-sided color mode	12.2.1.1	enable
CURRENT_VERTEX_ATTRIB	$16 * \times R^4$	**GetVertexAttribfv**	0.0,0.0,0.0,1.0	Current generic vertex attribute values	10.2	current
PROGRAM_POINT_SIZE	B	**IsEnabled**	FALSE	Point size mode	14.4	enable

Table 23.55: Query Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
QUERY_RESULT	Z^+	**GetQueryObjectuiv**	0 or FALSE	Query object result	4.2.3	–
QUERY_RESULT_AVAILABLE	B	**GetQueryObjectiv**	TRUE	Is the query object result available?	4.2.3	–
–	S	**GetObjectLabel**	empty	Debug label	20.9	–
QUERY_TARGET	E	**GetQueryObjectiv**	NONE	Target of query object	4.2	–

Table 23.56: Image State (state per image unit)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
IMAGE_BINDING_NAME	$8 * \times Z^+$	**GetIntegeri_v**	0	Name of bound texture object	8.26	–
IMAGE_BINDING_LEVEL	$8 * \times Z^+$	**GetIntegeri_v**	0	Level of bound texture object	8.26	–
IMAGE_BINDING_LAYERED	$8 * \times B$	**GetBooleani_v**	FALSE	Texture object bound with multiple layers	8.26	–
IMAGE_BINDING_LAYER	$8 * \times Z^+$	**GetIntegeri_v**	0	Layer of bound texture, if not layered	8.26	–
IMAGE_BINDING_ACCESS	$8 * \times E$	**GetIntegeri_v**	READ_ONLY	Read and/or write access for bound texture	8.26	–
IMAGE_BINDING_FORMAT	$8 * \times Z^+$	**GetIntegeri_v**	R8	Format used for accesses to bound texture	8.26	–

Table 23.57: Atomic Counter Buffer Binding State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ATOMIC_COUNTER_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current value of generic atomic counter buffer binding	6.8	–
ATOMIC_COUNTER_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Buffer object bound to each atomic counter buffer binding point	6.8	–
ATOMIC_COUNTER_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start offset of binding range for each atomic counter buffer	6.8	–
ATOMIC_COUNTER_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of binding range for each atomic counter buffer	6.8	–

Table 23.58: Shader Storage Buffer Binding State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SHADER_STORAGE_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current value of generic shader storage buffer binding	7.8	–
SHADER_STORAGE_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Buffer object bound to each shader storage buffer binding point	7.8	–
SHADER_STORAGE_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start offset of binding range for each shader storage buffer	7.8	–
SHADER_STORAGE_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of binding range for each shader storage buffer	7.8	–

Table 23.59: Transform Feedback Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TRANSFORM_FEEDBACK_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Buffer object bound to each transform feedback attribute stream	6.7	–
TRANSFORM_FEEDBACK_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start offset of binding range for each transform feedback attrib. stream	6.7	–
TRANSFORM_FEEDBACK_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of binding range for each transform feedback attrib. stream	6.7	–
TRANSFORM_FEEDBACK_PAUSED	B	**GetBooleanv**	FALSE	Is transform feedback paused on this object?	6.7	–
TRANSFORM_FEEDBACK_ACTIVE	B	**GetBooleanv**	FALSE	Is transform feedback active on this object?	6.7	–
-	S	**GetObjectLabel**	empty	Debug label	20.9	–

Table 23.60: Uniform Buffer Binding State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNIFORM_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Uniform buffer object bound to the context for buffer object manipulation	7.6.2	–
UNIFORM_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Uniform buffer object bound to the specified context binding point	7.6.2	–
UNIFORM_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start of bound uniform buffer region	6.7	–
UNIFORM_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of bound uniform buffer region	6.7	–

Table 23.61: Sync Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
OBJECT_TYPE	E	**GetSynciv**	SYNC_FENCE	Type of sync object	4.1	–
SYNC_STATUS	E	**GetSynciv**	UNSIGNALED	Sync object status	4.1	–
SYNC_CONDITION	E	**GetSynciv**	SYNC_GPU_COMMANDS_COMPLETE	Sync object condition	4.1	–
SYNC_FLAGS	Z^+	**GetSynciv**	0	Sync object flags	4.1	–
–	S	**GetObjectPtrLabel**	empty	Debug label	20.9	–

Table 23.62: Hints

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
PERSPECTIVE_CORRECTION_HINT	E	**GetIntegerv**	DONT_CARE	Perspective correction hint	21.5	hint
POINT_SMOOTH_HINT	E	**GetIntegerv**	DONT_CARE	Point smooth hint	21.5	hint
LINE_SMOOTH_HINT	E	**GetIntegerv**	DONT_CARE	Line smooth hint	21.5	hint
POLYGON_SMOOTH_HINT	E	**GetIntegerv**	DONT_CARE	Polygon smooth hint	21.5	hint
FOG_HINT	E	**GetIntegerv**	DONT_CARE	Fog hint	21.5	hint
GENERATE_MIPMAP_HINT	E	**GetIntegerv**	DONT_CARE	Mipmap generation hint	21.5	hint
TEXTURE_COMPRESSION_HINT	E	**GetIntegerv**	DONT_CARE	Texture compression quality hint	21.5	hint
FRAGMENT_SHADER_DERIVATIVE_HINT	E	**GetIntegerv**	DONT_CARE	Fragment shader derivative accuracy hint	21.5	hint

Table 23.63: Compute Dispatch State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DISPATCH_INDIRECT_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Indirect dispatch buffer binding	19	–

Table 23.64: Implementation Dependent Values

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
CONTEXT_RELEASE_BEHAVIOR	E	**GetIntegerv**	See sec. 22.2	Flush behavior when context is released	22.2	–
MAX_CLIP_DISTANCES	Z^+	**GetIntegerv**	8	Max. no. of user clipping planes	13.7	–
MAX_CULL_DISTANCES	Z^+	**GetIntegerv**	8	Max. no. of user culling planes	13.7	–
MAX_COMBINED_CLIP_AND_CULL_DISTANCES	Z^+	**GetIntegerv**	8	Max. combined no. of user clipping	13.7	–
SUBPIXEL_BITS	Z^+	**GetIntegerv**	4	No. of bits of subpixel precision in screen x_w and y_w	14	–
MAX_ELEMENT_INDEX	Z^+	**GetInteger64v**	$2^{32}-1$	Max. element index	10.4	–
PRIMITIVE_RESTART_FOR_PATCHES_SUPPORTED	B	**GetBooleanv**	–	Primitive restart support for `PATCHES`	10.3.6	–
MAX_3D_TEXTURE_SIZE	Z^+	**GetIntegerv**	2048	Max. 3D texture image dimension	8.5	–
MAX_TEXTURE_SIZE	Z^+	**GetIntegerv**	16384	Max. 2D/1D texture image dimension	8.5	–
MAX_ARRAY_TEXTURE_LAYERS	Z^+	**GetIntegerv**	2048	Max. no. of layers for texture arrays	8.5	–
MAX_TEXTURE_LOD_BIAS	R^+	**GetFloatv**	2.0	Max. absolute texture level-of-detail bias	8.14	–
MAX_CUBE_MAP_TEXTURE_SIZE	Z^+	**GetIntegerv**	16384	Max. cube map texture image dimension	8.5	–
MAX_RENDERBUFFER_SIZE	Z^+	**GetIntegerv**	16384	Max. width and height of renderbuffers	9.2.4	–
MAX_TEXTURE_MAX_ANISOTROPY	R	**GetFloatv**	16.0	Limit of maximum degree of anisotropy	8.14	–

Table 23.65: Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_LIGHTS	Z^+	**GetIntegerv**	8	Max. no. of lights	12.2.1	–
MAX_COLOR_MATRIX_STACK_DEPTH	Z^+	**GetIntegerv**	2	Max. color matrix stack depth	8.4.3	–
MAX_MODELVIEW_STACK_DEPTH	Z^+	**GetIntegerv**	32	Max. model-view stack depth	12.1.1	–
MAX_PROJECTION_STACK_DEPTH	Z^+	**GetIntegerv**	2	Max. projection matrix stack depth	12.1.1	–
MAX_TEXTURE_STACK_DEPTH	Z^+	**GetIntegerv**	2	Max. no. depth of texture matrix stack	12.1.1	–
MAX_PIXEL_MAP_TABLE	Z^+	**GetIntegerv**	32	Max. size of a **PixelMap** translation table	8.4.3	–
MAX_NAME_STACK_DEPTH	Z^+	**GetIntegerv**	64	Max. selection name stack depth	21.2	–
MAX_LIST_NESTING	Z^+	**GetIntegerv**	64	Max. display list call nesting	21.4	–
MAX_EVAL_ORDER	Z^+	**GetIntegerv**	8	Max. evaluator polynomial order	21.1	–
MAX_ATTRIB_STACK_DEPTH	Z^+	**GetIntegerv**	16	Max. depth of the server attribute stack	21.6	–
MAX_CLIENT_ATTRIB_STACK_DEPTH	Z^+	**GetIntegerv**	16	Max. depth of the client attribute stack	21.6	–
–	$3 \times Z^+$	-	32	Max. size of a color table	8.4.3	–
–	Z^+	-	32	Max. size of the histogram table	8.4.3	–
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of aliased point sizes	14.4	–
MAX_CONVOLUTION_WIDTH	$3 \times Z^+$	**GetConvolution-Parameteriv**	3	Max. width of convolution filter	18	–
MAX_CONVOLUTION_HEIGHT	$2 \times Z^+$	**GetConvolution-Parameteriv**	3	Max. height of convolution filter	18	–

Table 23.66: Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_VIEWPORT_DIMS	$2 \times Z^+$	**GetFloatv**	See sec. 13.8.1	Max. viewport dimensions	13.8.1	–
MAX_VIEWPORTS	Z^+	**GetIntegerv**	16	Max. no. of active viewports	13.8.1	–
VIEWPORT_SUBPIXEL_BITS	Z^+	**GetIntegerv**	0	No. of bits of subpixel precision for viewport bounds	13.8.1	–
VIEWPORT_BOUNDS_RANGE	$2 \times R$	**GetFloatv**	†	Viewport bounds range $[min, max]$ † (at least $[-32768, 32767]$)	13.8.1	–
LAYER_PROVOKING_VERTEX	E	**GetIntegerv**	See sec. 11.3.4	Vertex convention followed by `gl_Layer`	11.3.4	–
VIEWPORT_INDEX_PROVOKING_VERTEX	E	**GetIntegerv**	See sec. 11.3.4	Vertex convention followed by `gl_ViewportIndex`	11.3.4	–
POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of point sprite sizes	14.4	–
POINT_SIZE_GRANULARITY	R^+	**GetFloatv**	–	Point sprite size granularity	14.4	–
ALIASED_LINE_WIDTH_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of aliased line widths	14.5	–
SMOOTH_LINE_WIDTH_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of antialiased line widths	14.5	–
SMOOTH_LINE_WIDTH_GRANULARITY	R^+	**GetFloatv**	–	Antialiased line width granularity	14.5	–
MAX_ELEMENTS_INDICES	Z^+	**GetIntegerv**	–	Recommended max no. of **DrawRangeElements** indices	10.3	–
MAX_ELEMENTS_VERTICES	Z^+	**GetIntegerv**	–	Recommended max no. of **DrawRangeElements** vertices	10.3	–

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_VERTEX_ATTRIB_RELATIVE_OFFSET	Z	**GetIntegerv**	2047	Max. offset added to vertex buffer binding offset	10.3	–
MAX_VERTEX_ATTRIB_BINDINGS	Z_{16*}	**GetIntegerv**	16	Max. no. of vertex buffers	10.3	–
MAX_VERTEX_ATTRIB_STRIDE	Z	**GetIntegerv**	2048	Max. vertex attribute stride	10.3	–
NUM_COMPRESSED_TEXTURE_FORMATS	Z^+	**GetIntegerv**	18	No. of compressed texture formats	8.7	–
COMPRESSED_TEXTURE_FORMATS	$18 * \times Z^+$	**GetIntegerv**	-	Enumerated compressed texture formats	8.7	–
MAX_TEXTURE_BUFFER_SIZE	Z^+	**GetIntegerv**	65536	No. of addressable texels for buffer textures	8.9	–
MAX_RECTANGLE_TEXTURE_SIZE	Z^+	**GetIntegerv**	16384	Max. width & height of rectangle textures	8.5	–
NUM_PROGRAM_BINARY_FORMATS	Z^+	**GetIntegerv**	0	No. of program binary formats	7.5	–
PROGRAM_BINARY_FORMATS	$0 * \times Z^+$	**GetIntegerv**	N/A	Enumerated program binary formats	7.5	–
NUM_SHADER_BINARY_FORMATS	Z^+	**GetIntegerv**	0	No. of shader binary formats	7.2	–
SHADER_BINARY_FORMATS	$0 * \times Z^+$	**GetIntegerv**	-	Enumerated shader binary formats	7.2	–
SHADER_COMPILER	B	**GetBooleanv**	TRUE	Shader compiler supported	7	–
MIN_MAP_BUFFER_ALIGNMENT	Z^+	**GetIntegerv**	64	Min byte alignment of pointers returned by **Map*Buffer**	6.3	–
TEXTURE_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	256†	Min. required alignment for texture buffer offsets	8.9	–

Table 23.67: Implementation Dependent Values (cont.)
† The value of TEXTURE_BUFFER_OFFSET_ALIGNMENT is the maximum allowed, not the minimum.

Table 23.68: Implementation Dependent Version and Extension Support

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAJOR_VERSION	Z^+	**GetIntegerv**	–	Major version no. supported	22.2	–
MINOR_VERSION	Z^+	**GetIntegerv**	–	Minor version no. supported	22.2	–
CONTEXT_FLAGS	Z^+	**GetIntegerv**	–	Context full/forward-compatible flag	22.2	–
CONTEXT_PROFILE_MASK	Z^+	**GetIntegerv**	–	Context profile mask	22.2	–
EXTENSIONS	S	**GetString**	–	Supported extension names	22.2	–
EXTENSIONS	$n \times S$	**GetStringi**	–	Supported individual extension names	22.2	–
NUM_EXTENSIONS	Z^+	**GetIntegerv**	0	No. of individual extension names	22.2	–
RENDERER	S	**GetString**	–	Renderer string	22.2	–
SHADING_LANGUAGE_VERSION	S	**GetString**	–	Latest Shading Language version supported	22.2	–
SHADING_LANGUAGE_VERSION	$n \times S$	**GetStringi**	–	Supported Shading Language versions	22.2	–
NUM_SHADING_LANGUAGE_VERSIONS	Z^+	**GetIntegerv**	3	No. of supported Shading Language versions	22.2	–
SPIR_V_EXTENSIONS	$n \times S$	**GetStringi**	-	Supported SPIR-V extension names	22.2	–
NUM_SPIR_V_EXTENSIONS	Z^+	**GetIntegerv**	0	No. of supported SPIR-V extension names	22.2	–
VENDOR	S	**GetString**	–	Vendor string	22.2	–
VERSION	S	**GetString**	–	OpenGL version supported	22.2	–

Table 23.69: Implementation Dependent Vertex Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_VERTEX_ATTRIBS	Z^+	**GetIntegerv**	16	No. of active vertex attributes	10.2	–
MAX_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of components for vertex shader uniform variables	7.6	–
MAX_VERTEX_UNIFORM_VECTORS	Z^+	**GetIntegerv**	256	No. of vectors for vertex shader uniform variables	7.6	–
MAX_VERTEX_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	14*	Max. no. of vertex uniform buffers per program	7.6.2	–
MAX_VERTEX_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	64	Max. no. of components of outputs written by a vertex shader	11.1.2.1	–
MAX_VERTEX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of texture image units accessible by a vertex shader	11.1.3.5	–
MAX_VERTEX_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	0	No. of atomic counter buffers accessed by a vertex shader	7.7	–
MAX_VERTEX_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	0	No. of atomic counters accessed by a vertex shader	11.1.3.6	–
MAX_VERTEX_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	0	No. of shader storage blocks accessed by a vertex shader	7.8	–

Table 23.70: Implementation Dependent Tessellation Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_TESS_GEN_LEVEL	Z^+	**GetIntegerv**	64	Max. level supported by tess. primitive generator	11.2.2	–
MAX_PATCH_VERTICES	Z^+	**GetIntegerv**	32	Max. patch size	10.1	–
MAX_TESS_CONTROL_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of words for tess. control shader (TCS) uniforms	11.2.1.1	–
MAX_TESS_CONTROL_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of tex. image units for TCS	11.1.3	–
MAX_TESS_CONTROL_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	128	No. components for TCS per-vertex outputs	11.2.1.2	–
MAX_TESS_PATCH_COMPONENTS	Z^+	**GetIntegerv**	120	No. components for TCS per-patch outputs	11.2.1.2	–
MAX_TESS_CONTROL_TOTAL_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	4096	Total no. components for TCS outputs	11.2.1.2	–
MAX_TESS_CONTROL_INPUT_COMPONENTS	Z^+	**GetIntegerv**	128	No. components for TCS per-vertex inputs	11.2.1.2	–
MAX_TESS_CONTROL_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	14$*$	No. of supported uniform blocks for TCS	7.6.2	–
MAX_TESS_CONTROL_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	0	No. of atomic counter (AC) buffers accessed by a TCS	7.7	–
MAX_TESS_CONTROL_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	0	No. of ACs accessed by a TCS	7.7	–
MAX_TESS_CONTROL_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	0	No. of shader storage blocks accessed by a tess. control shader	7.8	–

Table 23.71: Implementation Dependent Tessellation Shader Limits (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_TESS_EVALUATION_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of words for tess. evaluation shader (TES) uniforms	11.2.3.1	–
MAX_TESS_EVALUATION_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of tex. image units for TES	11.1.3	–
MAX_TESS_EVALUATION_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	128	No. components for TES per-vertex outputs	11.2.3.2	–
MAX_TESS_EVALUATION_INPUT_COMPONENTS	Z^+	**GetIntegerv**	128	No. components for TES per-vertex inputs	11.2.3.2	–
MAX_TESS_EVALUATION_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	14*	No. of supported uniform blocks for TES	7.6.2	–
MAX_TESS_EVALUATION_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	0	No. of AC buffers accessed by a TES	11.1.3.6	–
MAX_TESS_EVALUATION_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	0	No. of ACs accessed by a TES	11.1.3.6	–
MAX_TESS_EVALUATION_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	0	No. of shader storage blocks accessed by a tess. evaluation shader	7.8	–

Table 23.72: Implementation Dependent Geometry Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_GEOMETRY_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of components for geometry shader (GS) uniform variables	11.3.3	–
MAX_GEOMETRY_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	14∗	Max. no. of GS uniform buffers per program	7.6.2	–
MAX_GEOMETRY_INPUT_COMPONENTS	Z^+	**GetIntegerv**	64	Max. no. of components of inputs read by a GS	11.3.4.4	–
MAX_GEOMETRY_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	128	Max. no. of components of outputs written by a GS	11.3.4.5	–
MAX_GEOMETRY_OUTPUT_VERTICES	Z^+	**GetIntegerv**	256	Max. no. of vertices that any GS can emit	11.3.4	–
MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	1024	Max. no. of total components (all vertices) of active outputs that a GS can emit	11.3.4	–
MAX_GEOMETRY_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of texture image units accessible by a GS	11.3.4	–
MAX_GEOMETRY_SHADER_INVOCATIONS	Z^+	**GetIntegerv**	32	Max. supported GS invocation count	11.3.4.2	–
MAX_VERTEX_STREAMS	Z^+	**GetIntegerv**	4	Total no. of vertex streams	11.3.4.2	–
MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	0	No. of atomic counter buffers accessed by a GS	7.7	–
MAX_GEOMETRY_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	0	No. of atomic counters accessed by a GS	11.1.3.6	–
MAX_GEOMETRY_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	0	No. of shader storage blocks accessed by a GS	7.8	–

Table 23.73: Implementation Dependent Fragment Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of components for fragment shader (FS) uniform variables	15.1	–
MAX_FRAGMENT_UNIFORM_VECTORS	Z^+	**GetIntegerv**	256	No. of vectors for FS uniform variables	15.1	–
MAX_FRAGMENT_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	14*	Max. no. of FS uniform buffers per program	7.6.2	–
MAX_FRAGMENT_INPUT_COMPONENTS	Z^+	**GetIntegerv**	128	Max. no. of components of inputs read by a FS	15.2.2	–
MAX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	No. of texture image units accessible by a FS	11.1.3.5	–
MIN_PROGRAM_TEXTURE_GATHER_OFFSET	Z	**GetIntegerv**	-8	Min texel offset for `textureGather`	8.14.1	–
MAX_PROGRAM_TEXTURE_GATHER_OFFSET	Z^+	**GetIntegerv**	7	Max. texel offset for `textureGather`	8.14.1	–
MAX_TEXTURE_UNITS	Z^+	**GetIntegerv**	2	No. of fixed-function texture units	10	–
MAX_TEXTURE_COORDS	Z^+	**GetIntegerv**	8	No. of texture coordinate sets	10.2.2	–
MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	1	No. of atomic counter buffers accessed by a FS	7.7	–
MAX_FRAGMENT_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	8	No. of atomic counters accessed by a FS	11.1.3.6	–
MAX_FRAGMENT_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	8	No. of shader storage blocks accessed by a FS	7.8	–

Table 23.74: Implementation Dependent Compute Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_COMPUTE_WORK_GROUP_COUNT	$3 \times Z^+$	**GetIntegeri_v**	65535	Max. no. of workgroups (WG) that may be dispatched by a single dispatch command (per dimension)	19	–
MAX_COMPUTE_WORK_GROUP_SIZE	$3 \times Z^+$	**GetIntegeri_v**	1024 (x, y), 64 (z)	Max. local size of a compute WG (per dimension)	19	–
MAX_COMPUTE_WORK_GROUP_INVOCATIONS	Z^+	**GetIntegerv**	1024	Max. total compute shader (CS) invocations in a single local WG	19	–
MAX_COMPUTE_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	14∗	Max. no. of uniform blocks per compute program	7.6.2	–
MAX_COMPUTE_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Max. no. of texture image units accessible by a CS	11.1.3.5	–
MAX_COMPUTE_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	8	No. of atomic counter buffers accessed by a CS	7.7	–
MAX_COMPUTE_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	8	No. of atomic counters accessed by a CS	11.1.3.6	–
MAX_COMPUTE_SHARED_MEMORY_SIZE	Z^+	**GetIntegerv**	32768	Max. total storage size of all variables declared as *shared* in all CSs linked into a single program object	19.1	–
MAX_COMPUTE_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	No. of components for CS uniform variables	19.1	–
MAX_COMPUTE_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	8	No. of image variables in compute shaders	11.1.3	–
MAX_COMBINED_COMPUTE_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	*	No. of words for compute shader uniform variables in all uniform blocks, including the default	19.1	–
MAX_COMPUTE_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	8	No. of shader storage blocks accessed by a compute shader	7.8	–

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MIN_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	-8	Min texel offset allowed in lookup	11.1.3.5	–
MAX_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	7	Max. texel offset allowed in lookup	11.1.3.5	–
MAX_UNIFORM_BUFFER_BINDINGS	Z^+	**GetIntegerv**	84	Max. no. of uniform buffer binding points on the context	7.6.2	–
MAX_UNIFORM_BLOCK_SIZE	Z^+	**GetIntegerv**	16384	Max. size in basic machine units of a uniform block	7.6.2	–
UNIFORM_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	256†	Min. required alignment for uniform buffer sizes and offsets	7.6.2	–
MAX_COMBINED_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	70∗	Max. no. of uniform buffers per program	7.6.2	–
MAX_VARYING_COMPONENTS	Z^+	**GetIntegerv**	60	No. of components for output variables	11.1.2.1	–
MAX_VARYING_VECTORS	Z^+	**GetIntegerv**	15	No. of vectors for output variables	11.1.2.1	–
MAX_COMBINED_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	80	Total no. of texture units accessible by the GL	11.1.3.5	–
MAX_SUBROUTINES	Z^+	**GetIntegerv**	256	Max. no. of subroutines per shader stage	7.10	–
MAX_SUBROUTINE_UNIFORM_LOCATIONS	Z^+	**GetIntegerv**	1024	Max. no. of subroutine uniform locations per stage	7.10	–
MAX_UNIFORM_LOCATIONS	Z^+	**GetIntegerv**	1024	Max. no. of user-assignable uniform locations	7.6	–

Table 23.75: Implementation Dependent Aggregate Shader Limits
† The value of `UNIFORM_BUFFER_OFFSET_ALIGNMENT` is the maximum allowed, not the minimum.

Table 23.76: Implementation Dependent Aggregate Shader Limits (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_ATOMIC_COUNTER_BUFFER_BINDINGS	Z^+	**GetIntegerv**	1	Max. no. of atomic counter buffer bindings	6.8	–
MAX_ATOMIC_COUNTER_BUFFER_SIZE	Z^+	**GetIntegerv**	32	Max. size in basic machine units of an atomic counter buffer	7.7	–
MAX_COMBINED_ATOMIC_COUNTER_BUFFERS	Z^+	**GetIntegerv**	1	Max. no. of atomic counter buffers per program	7.7	–
MAX_COMBINED_ATOMIC_COUNTERS	Z^+	**GetIntegerv**	8	Max. no. of atomic counter uniforms per program	11.1.3.6	–
MAX_SHADER_STORAGE_BUFFER_BINDINGS	Z^+	**GetIntegerv**	8	Max. no. of shader storage buffer bindings in the context	7.8	–
MAX_SHADER_STORAGE_BLOCK_SIZE	Z^+	**GetInteger64v**	2^{27}	Max. size in basic machine units of a shader storage block	7.8	–
MAX_COMBINED_SHADER_STORAGE_BLOCKS	Z^+	**GetIntegerv**	8	No. of shader storage blocks accessed by a program	7.8	–
SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	256†	Min. required alignment for shader storage buffer binding offsets	7.8	–

† The value of `SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT` is the maximum allowed, not the minimum.

Table 23.77: Implementation Dependent Aggregate Shader Limits (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_IMAGE_UNITS	Z^+	**GetIntegerv**	8	No. of units for image load/store/atom	8.26	–
MAX_COMBINED_SHADER_OUTPUT_RESOURCES	Z^+	**GetIntegerv**	8	Limit on active image units + shader storage blocks + fragment outputs	8.26	–
MAX_IMAGE_SAMPLES	Z^+	**GetIntegerv**	0	Max. allowed samples for a texture level bound to an image unit	8.26	–
MAX_VERTEX_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	0	No. of image variables in vertex shaders	11.1.3.7	–
MAX_TESS_CONTROL_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	0	No. of image variables in tess. control shaders	11.1.3.7	–
MAX_TESS_EVALUATION_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	0	No. of image variables in tess. eval. shaders	11.1.3.7	–
MAX_GEOMETRY_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	0	No. of image variables in geometry shaders	11.1.3.7	–
MAX_FRAGMENT_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	8	No. of image variables in fragment shaders	11.1.3.7	–
MAX_COMBINED_IMAGE_UNIFORMS	Z^+	**GetIntegerv**	8	No. of image variables in all shaders	11.1.3.7	–

Table 23.78: Implementation Dependent Aggregate Shader Limits (cont.)
† The minimum value for each stage is MAX_*stage*_UNIFORM_BLOCKS × MAX_UNIFORM_BLOCK_SIZE / 4 + MAX_*stage*_UNIFORM_COMPONENTS

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for vertex shader uniform variables in all uniform blocks (including default)	7.6.2	–
MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for geometry shader uniform variables in all uniform blocks (including default)	7.6.2	–
MAX_COMBINED_TESS_CONTROL_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for TCS uniform variables in all uniform blocks (including default)	11.2.1.1	–
MAX_COMBINED_TESS_EVALUATION_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for TES uniform variables in all uniform blocks (including default)	11.2.3.1	–
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	No. of words for fragment shader uniform variables in all uniform blocks (including default)	7.6.2	–

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DEBUG_CALLBACK_FUNCTION	Y	**GetPointerv**	NULL	The current debug output callback function pointer	20.2	–
DEBUG_CALLBACK_USER_PARAM	Y	**GetPointerv**	NULL	The current debug output callback user parameter	20.2	–
DEBUG_LOGGED_MESSAGES	Z^+	**GetIntegerv**	0	The no. of messages currently in the debug message log	20.3	–
DEBUG_NEXT_LOGGED_MESSAGE_LENGTH	Z^+	**GetIntegerv**	0	The string length of the oldest debug message in the debug message log	20.3	–
DEBUG_OUTPUT_SYNCHRONOUS	B	**IsEnabled**	FALSE	The enabled state for synchronous debug message callbacks	20.8	–
DEBUG_GROUP_STACK_DEPTH	Z^+	**GetIntegerv**	1	Debug group stack pointer	20.6	–
DEBUG_OUTPUT	B	**IsEnabled**	Depends on the context†	The enabled state for debug output functionality	20	–

Table 23.79: Debug Output State
† The initial value of DEBUG_OUTPUT is TRUE in a debug context and FALSE in a non-debug context.

Table 23.80: Implementation Dependent Debug Output State

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_DEBUG_MESSAGE_LENGTH	Z^+	**GetIntegerv**	1	The max length of a debug message string, including its null terminator	20.1	–
MAX_DEBUG_LOGGED_MESSAGES	Z^+	**GetIntegerv**	1	The max no. of messages stored in the debug message log	20.3	–
MAX_DEBUG_GROUP_STACK_DEPTH	Z^+	**GetIntegerv**	64	Max. group stack depth	20.6	–
MAX_LABEL_LENGTH	Z^+	**GetIntegerv**	256	Max. length of a label string	20.7	–

Table 23.81: Implementation Dependent Values (cont.)

† These queries return the maximum no. of samples for all internal formats required to support multisampled rendering.

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_FRAMEBUFFER_WIDTH	Z^+	**GetIntegerv**	16384	Max. width for framebuffer object	9.2.1	–
MAX_FRAMEBUFFER_HEIGHT	Z^+	**GetIntegerv**	16384	Max. height for framebuffer object	9.2.1	–
MAX_FRAMEBUFFER_LAYERS	Z^+	**GetIntegerv**	2048	Max. layer count for layered framebuffer object	9.2.1	–
MAX_FRAMEBUFFER_SAMPLES	Z^+	**GetIntegerv**	4	Max. sample count for framebuffer object	9.2.1	–
MAX_SAMPLE_MASK_WORDS	Z^+	**GetIntegerv**	1	Max. no. of sample mask words	14.9.3	–
MAX_SAMPLES	Z^+	**GetIntegerv**	4	Max. no. of samples supported for multisampling†	9.2.4	–
MAX_COLOR_TEXTURE_SAMPLES	Z^+	**GetIntegerv**	1	Max. no. of samples supported for all color formats in a multisample texture†	22.3	–
MAX_DEPTH_TEXTURE_SAMPLES	Z^+	**GetIntegerv**	1	Max. no. of samples supported for all depth/stencil formats in a multisample texture†	22.3	–
MAX_INTEGER_SAMPLES	Z^+	**GetIntegerv**	1	Max. no. of samples supported for all integer format multisample buffers†	22.3	–
QUADS_FOLLOW_PROVOKING_VERTEX_CONVENTION	B	**GetBooleanv**	–	Whether quads follow provoking vertex convention	13.6	–
QUERY_COUNTER_BITS	$n \times Z^+$	**GetQueryiv**	See sec. 4.2.3	Asynchronous query counter bits	4.2.3	–
MAX_SERVER_WAIT_TIMEOUT	Z^+	**GetInteger64v**	0	Max. **WaitSync** timeout interval	4.1.1	–

Table 23.82: Implementation Dependent Values (cont.)
† "ULP" means "Units in the Last Place" or "Unit of Least Precision".

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MIN_FRAGMENT_INTERPOLATION_OFFSET	R	**GetFloatv**	-0.5	Furthest negative offset for `interpolateAtOffset`	15.1	–
MAX_FRAGMENT_INTERPOLATION_OFFSET	R	**GetFloatv**	+0.5 - 1 ULP†	Furthest positive offset for `interpolateAtOffset`	15.1	–
FRAGMENT_INTERPOLATION_OFFSET_BITS	Z^+	**GetIntegerv**	4	Subpixel bits for `interpolateAtOffset`	15.1	–
MAX_DRAW_BUFFERS	Z^+	**GetIntegerv**	8	Max. no. of active draw buffers	17.4.1	–
MAX_DUAL_SOURCE_DRAW_BUFFERS	Z^+	**GetIntegerv**	1	Max. no. of active draw buffers when using dual-source blending	17.3.6	–
MAX_COLOR_ATTACHMENTS	Z^+	**GetIntegerv**	8	Max. no. of FBO attachment points for color buffers	9.2.7	–

Table 23.83: Internal Format Dependent Values

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
SAMPLES	$0 * \times Z^+$	**GetInternalformativ**	†	Supported sample counts † See section 22.3	22.3	–
NUM_SAMPLE_COUNTS	Z^+	**GetInternalformativ**	1	No. of supported sample counts	22.3	–

Table 23.84: Implementation Dependent Transform Feedback Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS	Z^+	**GetIntegerv**	64	Max. no. of components to write to a single buffer in interleaved mode	13.3	–
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS	Z^+	**GetIntegerv**	4	Max. no. of separate attributes or outputs that can be captured in transform feedback	13.3	–
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS	Z^+	**GetIntegerv**	4	Max. no. of components per attribute or output in separate mode	13.3	–
MAX_TRANSFORM_FEEDBACK_BUFFERS	Z^+	**GetIntegerv**	4	Max. no. of buffer objs to write with transform feedback	13.3	–

Table 23.85: Framebuffer Dependent Values

† Unlike most framebuffer-dependent state, which is queried from the currently bound draw framebuffer, this state is queried from the currently bound read framebuffer. n is the value of SAMPLES.

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
DOUBLEBUFFER	B	**GetBooleanv**	–	True if front & back buffers exist	17.4.1	–
STEREO	B	**GetBooleanv**	–	True if left & right buffers exist	22	–
SAMPLE_BUFFERS	Z_2	**GetIntegerv**	0	No. of multisample buffers	14.3.1	–
SAMPLES	Z^+	**GetIntegerv**	0	Coverage mask size	14.3.1	–
SAMPLE_POSITION	$n \times 2 \times R^{[0,1]}$	**GetMultisamplefv**	–	Explicit sample positions	14.3.1	–
IMPLEMENTATION_COLOR_READ_FORMAT	E	**GetIntegerv**	†	Implementation preferred pixel *format*	18.2.2	–
IMPLEMENTATION_COLOR_READ_TYPE	E	**GetIntegerv**	†	Implementation preferred pixel *type*	18.2.2	–

Table 23.86: Framebuffer Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
AUX_BUFFERS	Z^+	**GetIntegerv**	0	No. of auxiliary buffers	17.4.1	–
RGBA_MODE	B	**GetBooleanv**	–	True if color buffers store RGBA	10.2	–
INDEX_MODE	B	**GetBooleanv**	–	True if color buffers store indexes	10.2	–
x_BITS	Z^+	**GetIntegerv**	-	No. of bits in x color buffer component. x is one of RED, GREEN, BLUE, ALPHA, or INDEX	9	–
DEPTH_BITS	Z^+	**GetIntegerv**	-	No. of depth buffer planes	9	–
STENCIL_BITS	Z^+	**GetIntegerv**	-	No. of stencil planes	9	–
ACCUM_x_BITS	Z^+	**GetIntegerv**	-	No. of bits in x accumulation buffer component (x is RED, GREEN, BLUE, or ALPHA	9	–

Table 23.87: Miscellaneous

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
LIST_BASE	Z^+	**GetIntegerv**	0	Setting of **ListBase**	21.4	list
LIST_INDEX	Z^+	**GetIntegerv**	0	No. of display list under construction; 0 if none	21.4	–
LIST_MODE	Z^+	**GetIntegerv**	0	Mode of display list under construction; undefined if none	21.4	–
–	$n \times S$	**GetObjectLabel**	empty	Debug label (per display list)	20.9	–
–	$16 * \times A$	–	empty	Server attribute stack	22	–
ATTRIB_STACK_DEPTH	Z^+	**GetIntegerv**	0	Server attribute stack pointer	22	–
–	$16 * \times A$	–	empty	Client attribute stack	22	–
CLIENT_ATTRIB_STACK_DEPTH	Z^+	**GetIntegerv**	0	Client attribute stack pointer	22	–
NAME_STACK_DEPTH	Z^+	**GetIntegerv**	0	Name stack depth	21.2	–
RENDER_MODE	E	**GetIntegerv**	RENDER	**RenderMode** setting	21.2	–
SELECTION_BUFFER_POINTER	Y	**GetPointerv**	0	Selection buffer pointer	21.2	select
SELECTION_BUFFER_SIZE	Z^+	**GetIntegerv**	0	Selection buffer size	21.2	select
FEEDBACK_BUFFER_POINTER	Y	**GetPointerv**	0	Feedback buffer pointer	21.3	feedback
FEEDBACK_BUFFER_SIZE	Z^+	**GetIntegerv**	0	Feedback buffer size	21.3	feedback
FEEDBACK_BUFFER_TYPE	E	**GetIntegerv**	2D	Feedback type	21.3	feedback
–	$n \times E$	**GetError**	0	Current error code(s)	2.3.1	–
–	$n \times B$	–	FALSE	True if there is a corresponding error	2.3.1	–
CURRENT_QUERY	$n \times Z^+$	**GetQueryiv**	0	Active query object names	4.2.3	–
QUERY_BUFFER_BINDING	Z^+	**GetIntegeriv**	0	Query result buffer binding	4.2.3	–
COPY_READ_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer "read" bind point	6.6	–
COPY_WRITE_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer "write" bind point	6.6	–
RESET_NOTIFICATION_STRATEGY	Z_2	**GetIntegerv**	See sec. 2.3.2	Reset notification behavior	2.3.2	–
TEXTURE_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to generic texture buffer bind point	8.1	texture
TEXTURE_CUBE_MAP_SEAMLESS	B	**IsEnabled**	FALSE	Seamless cube map filtering enable	8.13	–

Appendix A

Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer *state vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state. This repeatability requirement doesn't apply when using shaders containing side effects (image and buffer variable stores and atomic operations, and atomic counter operations), because these memory operations are not guaranteed to be processed in a defined order.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL.

A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

Rule 1 *For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 2 *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

Required:

- *Framebuffer contents (all bitplanes)*
- *The color buffers enabled for writing*
- *The values of matrices other than the top-of-stack matrices*

- *Scissor parameters (other than enable)*
- *Writemasks (color, index, depth, stencil)*
- *Clear values (color, index, depth, stencil, accumulation)*
  - *Current values (color, index, normal, texture coords, edgeflag)*
  - *Current raster color, index and texture coordinates*
  - *Material properties (ambient, diffuse, specular, emission, shininess)*

Strongly suggested:

- *Matrix mode*
- *Matrix stack depths*
- *Alpha test parameters (other than enable)*
- *Stencil parameters (other than enable)*
- *Depth test parameters (other than enable)*
- *Blend parameters (other than enable)*
- *Logical operation parameters (other than enable)*
- *Pixel storage and transfer state*
- *Evaluator state (except as it affects the vertex data generated by the evaluators)*
- *Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)*

Corollary 1 *Fragment generation is invariant with respect to the state values marked with • in Rule 2.*

Corollary 2 *The window coordinates (x, y, and z) of generated fragments are also invariant with respect to*

Required:

- *Current values (color, color index, normal, texture coords, edgeflag)*
- *Current raster color, color index, and texture coordinates*
- *Material properties (ambient, diffuse, specular, emission, shininess)*

Rule 3 *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it (the parameters that control the alpha test, for instance, are the alpha test enable, the alpha test function, and the alpha test reference value).*

Corollary 3 *Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.*

Rule 4 *The same vertex or fragment shader will produce the same result when run multiple times with the same input. The wording 'the same shader' means a program object that is populated with the same source strings, which are compiled and then linked, possibly multiple times, and which program object is then executed using the same GL state vector. Invariance is relaxed for shaders with side effects, such as accessing atomic counters (see section A.5).*

Rule 5 *All fragment shaders that either conditionally or unconditionally assign* `gl_FragCoord.z` *to* `gl_FragDepth` *are depth-invariant with respect to each other, for those fragments where the assignment to* `gl_FragDepth` *actually is done.*

If a sequence of GL commands specifies primitives to be rendered with shaders containing side effects (image and buffer variable stores and atomic operations, and atomic counter operations), invariance rules are relaxed. In particular, rule 1, corollary 3, and rule 4 do not apply in the presence of shader side effects.

The following weaker versions of rules 1 and 4 apply to GL commands involving shader side effects:

Rule 6 *For any given GL and framebuffer state vector, and for any given GL command, the contents of any framebuffer state not directly or indirectly affected by results of shader image or buffer variable stores or atomic operations, or atomic counter operations must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 7 *The same vertex or fragment shader will produce the same result when run multiple times with the same input as long as:*

- *shader invocations do not use image atomic operations or atomic counters;*
- *no framebuffer memory is written to more than once by image stores, unless all such stores write the same value; and*

- *no shader invocation, or other operation performed to process the sequence of commands, reads memory written to by an image store.*

When any sequence of GL commands triggers shader invocations that perform image stores, atomic operations, or atomic counter operations, and subsequent GL commands read the memory written by those shader invocations, these operations must be explicitly synchronized. For more details, see section 7.13.

A.4 Tessellation Invariance

When using a program containing tessellation evaluation shaders, the fixed-function tessellation primitive generator consumes the input patch specified by an application and emits a new set of primitives. The following invariance rules are intended to provide repeatability guarantees. Additionally, they are intended to allow an application with a carefully crafted tessellation evaluation shader to ensure that the sets of triangles generated for two adjacent patches have identical vertices along shared patch edges, avoiding "cracks" caused by minor differences in the positions of vertices along shared edges.

Rule 1 *When processing two patches with identical outer and inner tessellation levels, the tessellation primitive generator will emit an identical set of point, line, or triangle primitives as long as the active program used to process the patch primitives has tessellation evaluation shaders specifying the same tessellation mode, spacing, vertex order, and point mode input layout qualifiers. Two sets of primitives are considered identical if and only if they contain the same number and type of primitives and the generated tessellation coordinates for the vertex numbered* m *of the primitive numbered* n *are identical for all values of* m *and* n*.*

Rule 2 *The set of vertices generated along the outer edge of the subdivided primitive in triangle and quad tessellation, and the tessellation coordinates of each, depends only on the corresponding outer tessellation level and the spacing input layout qualifier in the tessellation evaluation shader of the active program.*

Rule 3 *The set of vertices generated when subdividing any outer primitive edge is always symmetric. For triangle tessellation, if the subdivision generates a vertex with tessellation coordinates of the form* $(0, x, 1-x)$*,* $(x, 0, 1-x)$*, or* $(x, 1-x, 0)$*, it will also generate a vertex with coordinates of exactly* $(0, 1-x, x)$*,* $(1-x, 0, x)$*, or* $(1-x, x, 0)$*, respectively. For quad tessellation, if the subdivision generates a vertex with coordinates of* $(x, 0)$ *or* $(0, x)$*, it will also generate a vertex with coordinates of exactly* $(1-x, 0)$ *or* $(0, 1-x)$*, respectively. For isoline tessellation,*

if it generates vertices at $(0, x)$ *and* $(1, x)$ *where* x *is not zero, it will also generate vertices at exactly* $(0, 1 - x)$ *and* $(1, 1 - x)$*, respectively.*

Rule 4 *The set of vertices generated when subdividing outer edges in triangular and quad tessellation must be independent of the specific edge subdivided, given identical outer tessellation levels and spacing. For example, if vertices at* $(x, 1 - x, 0)$ *and* $(1-x, x, 0)$ *are generated when subdividing the* $w = 0$ *edge in triangular tessellation, vertices must be generated at* $(x, 0, 1 - x)$ *and* $(1 - x, 0, x)$ *when subdividing an otherwise identical* $v = 0$ *edge. For quad tessellation, if vertices at* $(x, 0)$ *and* $(1 - x, 0)$ *are generated when subdividing the* $v = 0$ *edge, vertices must be generated at* $(0, x)$ *and* $(0, 1-x)$ *when subdividing an otherwise identical* $u = 0$ *edge.*

Rule 5 *When processing two patches that are identical in all respects enumerated in rule 1 except for vertex order, the set of triangles generated for triangle and quad tessellation must be identical except for vertex and triangle order. For each triangle* n_1 *produced by processing the first patch, there must be a triangle* n_2 *produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in* n_1*.*

Rule 6 *When processing two patches that are identical in all respects enumerated in rule 1 other than matching outer tessellation levels and/or vertex order, the set of interior triangles generated for triangle and quad tessellation must be identical in all respects except for vertex and triangle order. For each interior triangle* n_1 *produced by processing the first patch, there must be a triangle* n_2 *produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in* n_1*. A triangle produced by the tessellator is considered an interior triangle if none of its vertices lie on an outer edge of the subdivided primitive.*

Rule 7 *For quad and triangle tessellation, the set of triangles connecting an inner and outer edge depends only on the inner and outer tessellation levels corresponding to that edge and the spacing input layout qualifier.*

Rule 8 *The value of all defined components of* `gl_TessCoord` *will be in the range* $[0, 1]$*. Additionally, for any defined component* x *of* `gl_TessCoord`*, the results of computing* $1.0 - x$ *in a tessellation evaluation shader will be exact. Some floating-point values in the range* $[0, 1]$ *may fail to satisfy this property, but such values may never be used as tessellation coordinate components.*

A.5 Atomic Counter Invariance

When using a program containing atomic counters, the following invariance rules are intended to provide repeatability guarantees but within certain constraints.

Rule 1 *When a single shader type within a program accesses an atomic counter with only* `atomicCounterIncrement`*, any individual shader invocation is guaranteed to get a unique value returned.*

Corollary 1 *Also holds true with* `atomicCounterDecrement`.

Corollary 2 *This does not hold true for* `atomicCounter`.

Corollary 3 *Repeatability is relaxed. While a unique value is returned to the shader, even given the same initial state vector and buffer contents, it is not guaranteed that the* **same** *unique value will be returned for each individual invocation of a shader (for example, on any single vertex, or any single fragment). It is wholly the shader writer's responsibility to respect this constraint.*

Rule 2 *When two or more shader types within a program access an atomic counter with only* `atomicCounterIncrement`*, there is no repeatability of the ordering of operations between stages. For example, some number of vertices may be processed, then some number of fragments may be processed.*

Corollary 4 *This also holds true with* `atomicCounterDecrement` *and* `atomicCounter`.

A.6 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL implementations cannot apply hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command *always* is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating-point values may be represented using different formats in different renderers (hardware and software), many OpenGL state values may change subtly when renderers are swapped. This is the type of state value change that invariance rule 1 in section A.3 seeks to avoid.

Appendix B

Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

1. The `CURRENT_RASTER_TEXTURE_COORDS` must be maintained correctly at all times, including periods while texture mapping is not enabled, and when the GL is in color index mode.

2. When requested, texture coordinates returned in feedback mode are always valid, including periods while texture mapping is not enabled, and when the GL is in color index mode.

3. The error semantics of upward compatible OpenGL revisions may change, and features deprecated in a previous revision may be removed. Otherwise, only additions can be made to upward compatible revisions.

4. GL query commands are not required to satisfy the semantics of the **Flush** or the **Finish** commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.

5. Application specified point size and line width must be returned as specified when queried. Implementation-dependent clamping affects the values only while they are in use.

6. Bitmaps and pixel transfers do not cause selection hits.

7. The mask specified as the third argument to **StencilFunc** affects the operands of the stencil comparison function, but has no direct effect on the update of the stencil buffer. The mask specified by **StencilMask** has no effect on the

stencil comparison function; it limits the effect of the update of the stencil buffer.

8. Polygon shading is completed before the polygon mode is interpreted. If the shade model is `FLAT`, all of the points or lines generated by a single polygon will have the same color.

9. A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed. If the list is created in `COMPILE` mode, errors should not be generated while the list is being created.

10. **RasterPos** does not change the current raster index from its default value in an RGBA mode GL context. Likewise, **RasterPos** does not change the current raster color from its default value in a color index GL context. Both the current raster index and the current raster color can be queried, however, regardless of the color mode of the GL context.

11. A material property that is attached to the current color via **ColorMaterial** always takes the value of the current color. Attempts to change that material property via **Material** calls have no effect.

12. **Material** and **ColorMaterial** can be used to modify the RGBA material properties, even in a color index context. Likewise, **Material** can be used to modify the color index material properties, even in an RGBA context.

13. There is no atomicity requirement for OpenGL rendering commands, even at the fragment level.

14. Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized in `FILL` mode, and the fragments generated by the rasterization of "narrow" polygons may not form a continuous array.

15. OpenGL does not force left- or right-handedness on any of its coordinates systems. Consider, however, the following conditions: (1) the object coordinate system is right-handed; (2) the only commands used to manipulate the model-view matrix are **Scale** (with positive scaling values only), **Rotate**, and **Translate**; (3) exactly one of either **Frustum** or **Ortho** is used to set the projection matrix; (4) the near value is less than the far value for **DepthRange**. If these conditions are all satisfied, then the eye coordinate system is right-handed and the clip, normalized device, and window coordinate systems are left-handed.

16. ColorMaterial has no effect on color index lighting.

17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge. That is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon; the positions of vertex A and C are identical; and the positions of vertex B and D are identical. Vertex positions are identical for the fixed-function pipeline if they are specified with the same input values and the state of coordinate transformations is identical when the vertices are processed; otherwise they are identical if the `gl_Position` values output by the vertex (or if active, geometry) shader are identical. Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.

18. OpenGL state continues to be modified in `FEEDBACK` mode and in `SELECT` mode. The contents of the framebuffer are not modified.

19. The current raster position, the user defined clip planes, the spot directions and the light positions for `LIGHT`i, and the eye planes for texgen are transformed when they are specified. They are not transformed during a **PopAttrib**, or when copying a context.

20. Dithering algorithms may be different for different components. In particular, alpha may be dithered differently from red, green, or blue, and an implementation may choose to not dither alpha at all.

21. For any GL and framebuffer state, and for any group of GL commands and arguments, the resulting GL and framebuffer state is identical whether the GL commands and arguments are executed normally or from a display list. The only exception to this corollary is for built-in shader variables `gl_VertexID` and `gl_PrimitiveID`, which are not defined when drawing geometry within a display list.

Appendix C

The OpenGL SPIR-V Execution Environment

C.1 Required Versions and Formats

Implementations must support the 1.0 version of SPIR-V and the 1.0 version of the SPIR-V Extended Instructions for the OpenGL Shading Language (see section 1.3.4).

A SPIR-V module passed into **ShaderBinary** is interpreted as a series of 32-bit words in host endianness, with literal strings packed as described in section 2.2 of the SPIR-V Specification. The first few words of the SPIR-V module must be a magic number and a SPIR-V version number, as described in section 2.3 of the SPIR-V Specification.

C.2 Valid SPIR-V Built-In Variable Decorations

Implementations must support the built-in variable decorations described in table C.1:

C.3 Valid SPIR-V Capabilities

Implementations
must support the capability operands declared by `OpCapability`, described in table C.2.

C.4 Validation Rules

In addition to what is allowed and disallowed above:

- Every entry point must have no return value and accept no arguments.
- The `Logical` addressing model must be selected.
- `Scope` for execution must be limited to
  - `Workgroup`
  - `Subgroup`
- `Scope` for memory must be limited to
  - `Device`
  - `Workgroup`
  - `Invocation`
- `Storage Class` must be limited to
  - `UniformConstant`
  - `Input`
  - `Uniform`
  - `Output`
  - `Workgroup`
  - `Private`
  - `Function`
  - `AtomicCounter`
  - `Image`
- Images
  - `OpTypeImage` must declare a scalar 32-bit float or 32-bit integer type for the `Sampled Type`.
  - `OpSampledImage`, `OpImageQuerySizeLod`, and `OpImageQueryLevels` must only consume an `Image` operand whose type has its `Sampled` operand set to 1.
  - The `Depth` operand of `OpTypeImage` is ignored.

  - The `Image` format of `OpImageWrite` must not be `Unknown`, unless the `StorageImageWriteWithoutFormat` `OpCapability` was declared.

- `AtomicCounter` storage class
  - Unless otherwise specified, variables declared in the `AtomicCounter` storage class must be unsigned 32-bit integers.
  - When using the Atomic Instructions, the `Memory Semantics` operand must be `AtomicCounterMemory`, without inclusion of memory-ordering bits (implying `Relaxed` semantics).

- Decorations
  - The `Flat`, `NoPerspective`, `Sample`, and `Centroid` decorations must not be used on variables with storage class other than `Input` or on variables used in the interface of non-fragment shader entry points.
  - The `Patch` decoration must not be used on variables in the interface of a vertex, geometry, or fragment shader stage's entry point.
  - `OpTypeRuntimeArray` must only be used for the last member of an `OpTypeStruct` in the `Uniform` storage class, and be decorated as `BufferBlock`
  - If the `Xfb` Execution Mode is set, any output variable that is at least partially captured:
    * must be decorated with an `XfbBuffer`, declaring the capturing buffer
    * must have at least one captured output variable in the capturing buffer decorated with an `XfbStride` (and all such `XfbStride` values for the capturing buffer must be equal)
  - If the `Xfb` Execution Mode is set, any captured output:
    * must be a non-structure decorated with `Offset` or a member of a structure whose type member is decorated with `Offset` (not all members of such a struct need be captured)
    * must be a numerical type scalar, vector, matrix, or array of these
    * must have an `Offset` that is a multiple of its first component
    * must have an `Offset` that is a multiple of 8 if any captured output in the capturing buffer is a 64-bit type, in which case the corresponding `XfbStride` must also be a multiple of 8

    - must not overlap (i.e., alias in any capturing buffer) any other captured output
    - must not have an `Offset` that causes overflow of the `XfbStride`

- Compute Shaders
  - For each compute shader entry point, either a `LocalSize` execution mode or an object decorated with the `WorkgroupSize` decoration must be specified.
- Recursion
  - For all entry points, the static function-call graph rooted at that entry point must not contain cycles.
- User declared input and output variables
  - must not use `OpTypeBool`, either directly or as part of an aggregate.
- Atomic Access Rules
  - Atomic instructions must declare a scalar 32-bit integer type, or a scalar 64-bit integer type if the `Int64Atomics` capability is enabled, for the value pointed to by `Pointer`.

C.5 Precision and Operation of SPIR-V Instructions

The following rules apply to both single and double-precision floating point instructions:

- Positive and negative infinities and positive and negative zeros are generated as dictated by IEEE-754, but subject to the precisions allowed in the following table.
- Dividing a non-zero by a zero results in the appropriately signed IEEE-754 Inf.
- Any denormalized value input into a shader or potentially generated by any instruction in a shader may be flushed to 0.
- The rounding mode cannot be set and is undefined.
- NaNs are not required to be generated. Instructions that operate on a NaN are not required to return a NaN as the result.

- Support for signaling NaNs is optional and exceptions are never raised.

The precision of double-precision instructions is at least that of single precision. For single precision (32 bit) instructions, precisions are required to be at least as described in table C.3.

C.6 Precision of GLSL.std.450 Instructions

GLSL.std.450 extended instructions specifically defined in terms of the instructions in table C.4 inherit the precisions described in that table. GLSL.std.450 extended instructions not listed in, or defined in terms of those instructions have undefined precision. These include, for example, the trigonometric functions and determinant.

For the `OpSRem` and `OpSMod` instructions, if either operand is negative the result is undefined.

Built-in Variable Decoration
BaseInstance
BaseVertex
ClipDistance
CullDistance
DrawIndex
FragCoord
FragDepth
FrontFacing
GlobalInvocationId
HelperInvocation
InstanceId
InvocationId
Layer
LocalInvocationId
LocalInvocationIndex
NumWorkgroups
PatchVertices
PointCoord
PointSize
Position
PrimitiveId
SampleId
SampleMask
SamplePosition
TessCoord
TessLevelInner
TessLevelOuter
VertexId
ViewportIndex
WorkgroupId
WorkgroupSize

Table C.1: Built-in Variable Decorations

OpCapability Operand
AtomicStorageOps
AtomicStorage
ClipDistance
CullDistance
DerivativeControl
DrawParameters
Float64
GeometryPointSize
GeometryStreams
Geometry
Image1D
ImageBuffer
ImageCubeArray
ImageGatherExtended
ImageMSArray
ImageQuery
ImageRect
InterpolationFunction
Matrix
MultiViewport
SampleRateShading
Sampled1D
SampledBuffer
SampledCubeArray
SampledImageArrayDynamicIndexing
SampledRect
Shader
StorageBufferArrayDynamicIndexing
StorageImageArrayDynamicIndexing
StorageImageExtendedFormats
StorageImageMultisample
StorageImageWriteWithoutFormat
SubgroupVoteKHR
TessellationPointSize
Tessellation
TransformFeedback
UniformBufferArrayDynamicIndexing

Table C.2: Valid SPIR-V Capabilities

Instruction	Precision
`OpFAdd`	Correctly rounded
`OpFSub`	Correctly rounded
`OpFMul`	Correctly rounded
`OpFOrdEqual`, `OpFUnordEqual`	Correct result
`OpFOrdLessThan`, `OpFUnordLessThan`	Correct result
`OpFOrdGreaterThan`, `OpFUnordGreaterThan`	Correct result
`OpFOrdLessThanEqual`, `OpFUnordLessThanEqual`	Correct result
`OpFOrdGreaterThanEqual`, `OpFUnordGreaterThanEqual`	Correct result
`OpFDiv`	2.5 ULP for b in the range $[2^{-126}, 2^{126}]$
conversions between types	Correctly rounded

Table C.3: 32-Bit Precision

Instruction	Precision
`fma()`	Inherited from `OpFMul` followed by `OpFAdd`.
`exp(x)`, `exp2(x)`	$(3 + 2\|x\|)$ ULP
`log()`, `log2()`	3 ULP outside the range $[0.5, 2.0]$; absolute error $< 2^{-21}$ inside the range $[0.5, 2.0]$
`pow(x,y)`	Inherited from `exp2(y * log2(x))`
`sqrt()`	Inherited from 1.0 / `inversesqrt()`
`inversesqrt()`	2 ULP

Table C.4: GLSL.std.450 Precision

Appendix D

Compressed Texture Image Formats

The compressed texture formats used by OpenGL are described in the specifically identified sections of the *Khronos Data Format Specification*, version 1.1, available at URL

https://www.khronos.org/registry/dataformat/specs/1.1/dataformat.1.1.html

Unless otherwise described, the quantities encoded in these compressed formats are treated as normalized, unsigned values.

Those formats listed as sRGB-encoded have in-memory representations of R, G and B components which are nonlinearly-encoded as R', G', and B'; any alpha component is unchanged. As part of filtering, the nonlinear R', G', and B' values are converted to linear R, G, and B components; any alpha component is unchanged. The conversion between linear and nonlinear encoding is performed as described in the `KHR_DF_TRANSFER_SRGB` section of the Khronos Data Format Specification.

D.1 RGTC Compressed Texture Image Formats

RGTC formats are described in the "RGTC Compressed Texture Image Formats" chapter of the Khronos Data Format Specification. The mapping between OpenGL RGTC formats and that specification is shown in table D.1.

OpenGL format	Data Format Specification description
COMPRESSED_RED_RGTC1	BC4 unsigned
COMPRESSED_SIGNED_RED_RGTC1	BC4 signed
COMPRESSED_RG_RGTC2	BC5 unsigned
COMPRESSED_SIGNED_RG_RGTC2	BC5 signed

Table D.1: Mapping of OpenGL RGTC formats to descriptions.

OpenGL format	Data Format Specification description
COMPRESSED_RGBA_BPTC_UNORM	BC7, linear encoding
COMPRESSED_SRGB_ALPHA_BPTC_UNORM	BC7, sRGB encoding
COMPRESSED_RGB_BPTC_SIGNED_FLOAT	BC6H, signed format
COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT	BC6H, unsigned format

Table D.2: Mapping of OpenGL BPTC formats to descriptions.

D.2 BPTC Compressed Texture Image Formats

BPTC formats are described in the "BPTC2 Compressed Texture Image Formats" chapter of the Khronos Data Format Specification. The mapping between OpenGL BPTC formats and that specification is shown in table D.2.

D.3 ETC Compressed Texture Image Formats

ETC formats are described in the "ETC2 Compressed Texture Image Formats" chapter of the Khronos Data Format Specification. The mapping between OpenGL ETC formats and that specification is shown in table D.3.

OpenGL format	Data Format Specification description
COMPRESSED_R11_EAC	Unsigned R11 EAC
COMPRESSED_SIGNED_R11_EAC	Signed R11 EAC
COMPRESSED_RG11_EAC	Unsigned RG11 EAC
COMPRESSED_SIGNED_RG11_EAC	Signed RG11 EAC
COMPRESSED_RGB8_ETC2	RGB ETC2
COMPRESSED_SRGB8_ETC2	RGB ETC2 with sRGB encoding
COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2	RGB ETC2 with punchthrough alpha
COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2	RGB ETC2 with punchthrough alpha and sRGB encoding
COMPRESSED_RGBA8_ETC2_EAC	RGBA ETC2
COMPRESSED_SRGB8_ALPHA8_ETC2_EAC	RGBA ETC2 with sRGB encoding

Table D.3: Mapping of OpenGL ETC formats to descriptions.

Appendix E

Profiles and the Deprecation Model

OpenGL 3.0 introduced a deprecation model in which certain features are marked as *deprecated*. Deprecated features are expected to be completely removed from a future version of OpenGL. Deprecated features are summarized in section E.2.

To aid developers in writing applications which will run on such future versions, it is possible to create an OpenGL context which does not support deprecated features. Such a context is called a *forward compatible context*, while a context supporting all OpenGL features is called a *full context*. Forward compatible contexts cannot restore deprecated functionality through extensions, but they may support additional, non-deprecated functionality through extensions.

Profiles define subsets of OpenGL functionality targeted to specific application domains. Starting with OpenGL 3.2, two profiles are defined (see below). Future versions may define additional profiles addressing embedded systems or other domains. OpenGL implementations are not required to support all defined profiles, but must support the *core* profile described below.

To enable application control of deprecation and profiles, new *context creation APIs* have been defined as extensions to GLX, WGL and EGL. These APIs allow specifying a particular version, profile, and full or forward compatible status, and will either create a context compatible with the request, or fail (if, for example, requesting an OpenGL version or profile not supported by the implementation).

Only the ARB may define OpenGL profiles and deprecated features.

E.1 Core and Compatibility Profiles

The *core profile* of OpenGL defines essential functionality for the modern programmable shading model introduced in OpenGL 2.0, but does not include features marked as removed for that version of the Specification (see section E.2).

The *compatibility profile* does not remove any functionality.

It is not possible to implement both core and compatibility profiles in a single GL context, since the core profile mandates functional restrictions not present in the compatibility profile. Refer to the `WGL_ARB_create_context_profile` and `GLX_ARB_create_context_profile` extensions (see appendix K.3.3.66) for information on creating a context implementing a specific profile.

E.2 Deprecated and Removed Features

OpenGL 3.0 defined a set of *deprecated features*. OpenGL 3.1 removed most of the deprecated features and moved them into the optional `GL_ARB_compatibility` extension. The OpenGL 3.2 core profile removes the same features as OpenGL 3.1, while the optional compatibility profile supports all those features.

Deprecated and removed features are summarized below in two groups: features which are marked deprecated by the core profile, but have not yet been removed, and features actually removed from the core profile of the current version of OpenGL (no features have been removed from or deprecated in the compatibility profile).

Functions which have been removed will generate an `INVALID_OPERATION` error if called in the core profile or in a forward-compatible context. Functions which are partially removed (e.g. no longer accept some parameter values) will generate the errors appropriate for any other unrecognized value of that parameter when a removed parameter value is passed in the core profile or a forward-compatible context. Functions which are deprecated but have not yet been removed from the core profile continue to operate normally except in a forward-compatible context, where they are also removed.

E.2.1 Deprecated But Still Supported Features

The following features are deprecated, but still present in the core profile. They may be removed from a future version of OpenGL, and are removed in a forward-compatible context implementing the core profile.

- Wide lines - **LineWidth** values greater than 1.0 will generate an `INVALID_VALUE` error.

- Global component limit query - the implementation-dependent values MAX_VARYING_COMPONENTS and MAX_VARYING_FLOATS.
- The query targets NUM_COMPRESSED_TEXTURE_FORMATS and COMPRESSED_TEXTURE_FORMATS (see section 8.5).
- Bitmap pack/unpack state for bitmaps - the pixel pack parameters UNPACK_LSB_FIRST and PACK_LSB_FIRST.

E.2.2 Removed Features

- Application-generated object names - the names of all object types, such as buffer, query, and texture objects, must be generated using the corresponding **Gen*** commands. Trying to bind an object name not returned by a **Gen*** command will result in an INVALID_OPERATION error. This behavior is already the case for framebuffer, renderbuffer, and vertex array objects. Object types which have default objects (objects named zero), such as vertex array, framebuffer, and texture objects, may also bind the default object, even though it is not returned by **Gen***.
- Color index mode - no color index visuals are supplied by the window system-binding APIs such as GLX and WGL, so the default framebuffer is always in RGBA mode. All language and state related to color index mode vertex, rasterization, and fragment processing behavior is removed. COLOR_INDEX formats are also deprecated.
- OpenGL Shading Language versions 1.10 and 1.20. These versions of the shading language depend on many API features that have also been deprecated.
- **Begin / End** primitive specification - **Begin**, **End**, and **EdgeFlag*** (section 10.7); **Color***, **FogCoord***, **Index***, **Normal3***, **SecondaryColor3***, **TexCoord***, **Vertex*** (section 10.2); and all associated state in tables 23.2 and 23.3. Vertex arrays and array drawing commands must be used to draw primitives. However, **VertexAttrib*** and the current vertex attribute state are retained in order to provide default attribute values for disabled attribute arrays.
- Edge flags and fixed-function vertex processing - **ColorPointer**, **EdgeFlagPointer**, **FogCoordPointer**, **IndexPointer**, **NormalPointer**, **SecondaryColorPointer**, **TexCoordPointer**, **VertexPointer**, **EnableClientState**,

DisableClientState, and **InterleavedArrays**, **ClientActiveTexture** (section 10.3); **Frustum**, **LoadIdentity**, **LoadMatrix**, **LoadTransposeMatrix**, **MatrixMode**, **MultMatrix**, **MultTransposeMatrix**, **Ortho**, **PopMatrix**, **PushMatrix**, **Rotate**, **Scale**, and **Translate** (section 12.1.1; **Enable/Disable** targets RESCALE_NORMAL and NORMALIZE (section 12.1.2); **TexGen*** and **Enable/Disable** targets TEXTURE_GEN_* (section 12.1.3, **Material***, **Light***, **LightModel***, and **ColorMaterial**, **ShadeModel**, and **Enable/Disable** targets LIGHTING, VERTEX_PROGRAM_TWO_SIDE, LIGHT*i*, and COLOR_MATERIAL (sections 12.2.2 and 12.2.3; **ClipPlane**; and all associated fixed-function vertex array, multitexture, matrix and matrix stack, normal and texture coordinate, lighting, and clipping state. A vertex shader must be defined in order to draw primitives.

Language referring to edge flags in the current specification is modified as though all edge flags are TRUE.

Note that the **FrontFace** and **ClampColor** commands in section 12.2 are **not** deprecated, as they still affect other non-deprecated functionality; however, the **ClampColor** *target*s CLAMP_VERTEX_COLOR and CLAMP_FRAGMENT_COLOR are deprecated.

- Client vertex and index arrays - all vertex array attribute and element array index pointers must refer to buffer objects (section 10.3.9). The default vertex array object (the name zero) is also deprecated. Calling **VertexAttribPointer** when no buffer object or no vertex array object is bound will generate an INVALID_OPERATION error, as will calling any array drawing command when no vertex array object is bound.

- Rectangles - **Rect*** (section 10.8).

- Current raster position - **RasterPos*** and **WindowPos*** (section 14.7), and all associated state.

- Two-sided color selection (section 12.2.1.1) - **Enable** target VERTEX_PROGRAM_TWO_SIDE; OpenGL Shading Language built-ins gl_BackColor and gl_BackSecondaryColor; and all associated state.

- Non-sprite points (section 14.4) - **Enable/Disable** targets POINT_SMOOTH and POINT_SPRITE, and all associated state. Point rasterization is always performed as though POINT_SPRITE were enabled.

- Wide lines and line stipple - **LineWidth** is not deprecated, but values greater than 1.0 will generate an INVALID_VALUE error; **LineStipple** and **Enable/Disable** target LINE_STIPPLE (section 14.5.2, and all associated state.

- Quadrilateral and polygon primitives - vertex array drawing modes `POLYGON`, `QUADS`, and `QUAD_STRIP` (section 10.7, related descriptions of rasterization of non-triangle polygons in section 14.6, and all associated state.

- Separate polygon draw mode - **PolygonMode** *face* values of `FRONT` and `BACK`; polygons are always drawn in the same mode, no matter which face is being rasterized.

- Polygon Stipple - **PolygonStipple** and **Enable/Disable** target `POLYGON_STIPPLE` (section 14.6.2), and all associated state.

- Pixel transfer modes and operations - all pixel transfer modes, including pixel maps, shift and bias, color table lookup, color matrix, and convolution commands and state (sections 8.4.2, 8.4.3, and 8.4.5), and all associated state and commands defining that state.

- Pixel drawing - **DrawPixels** and **PixelZoom** (section 18.1). However, the language describing pixel rectangles in section 8.4 is retained as it is required for **TexImage*** and **ReadPixels**.

- Bitmaps - **Bitmap** (section 14.8) and the `BITMAP` external format.

- Legacy OpenGL 1.0 pixel formats - the values 1, 2, 3, and 4 are no longer accepted as internal formats by **TexImage*** or any other command taking an internal format argument. The initial internal format of a texel array is `RGBA` instead of 1 (see section 8.22). `TEXTURE_COMPONENTS` is deprecated; always use `TEXTURE_INTERNAL_FORMAT`.

- Legacy pixel formats - all `ALPHA`, `LUMINANCE`, `LUMINANCE_ALPHA`, and `INTENSITY` external and internal formats, including compressed, floating-point, and integer variants (see tables 8.8, 8.18, 8.20, 8.22, 16.1, and 8.26); all references to luminance and intensity formats elsewhere in the specification, including conversion to and from those formats; and all associated state, including state describing the allocation or format of luminance and intensity texture or framebuffer components.

- Depth texture mode - `DEPTH_TEXTURE_MODE`. Section 8.23.1 is to be changed so that r is returned to texture samplers directly, and the OpenGL Shading Language 1.30 Specification is to be changed so that $(r, 0, 0, 1)$ is always returned from depth texture samplers in this case.

- Texture wrap mode CLAMP - CLAMP is no longer accepted as a value of texture parameters TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R.

- Texture borders - the *border* value to **TexImage*** must always be zero, or an INVALID_VALUE error is generated (section 8.5); all language in section 8 referring to nonzero border widths during texture image specification and texture sampling; and all associated state.

- Automatic mipmap generation - **TexParameter*** *target* GENERATE_MIPMAP (section 8.14.5), and all associated state.

- Fixed-function fragment processing - **AreTexturesResident**, **PrioritizeTextures**, and **TexParameter** *target* TEXTURE_PRIORITY; **TexEnv** *target* TEXTURE_ENV, and all associated parameters; **TexEnv** *target* TEXTURE_FILTER_CONTROL, and parameter name TEXTURE_LOD_BIAS; **Enable** *targets* of all dimensionalities (TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, and TEXTURE_CUBE_MAP); **Enable** target COLOR_SUM; **Enable** *target* FOG, **Fog**, and all associated parameters; the implementation-dependent values MAX_TEXTURE_UNITS and MAX_TEXTURE_COORDS; and all associated state.

- Alpha test - **AlphaFunc** and **Enable/Disable** target ALPHA_TEST (section 17.3.2), and all associated state.

- Accumulation buffers - **ClearAccum**, and ACCUM_BUFFER_BIT is not valid as a bit in the argument to **Clear** (section 17.4.3); **Accum** (section 17.4.5); the ACCUM_*_BITS framebuffer state describing the size of accumulation buffer components (table 23.85); and all associated state.

  Window system-binding APIs such as GLX and WGL may choose to either not expose window configs containing accumulation buffers, or to ignore accumulation buffers when the default framebuffer bound to a GL context contains them.

- Pixel copying - **CopyPixels** (the comments also applying to **CopyTexImage** will be moved to section 8.6).

- Auxiliary color buffers, including AUXi targets of the default framebuffer.

- Context framebuffer size queries - RED_BITS, GREEN_BITS, BLUE_BITS, ALPHA_BITS, DEPTH_BITS, and STENCIL_BITS.

- Evaluators - **Map***, **EvalCoord***, **MapGrid***, **EvalMesh***, **EvalPoint***, and all evaluator map enables in table 21.1 (section 21.1, and all associated state.
- Selection and feedback modes - **RenderMode**, **InitNames**, **PopName**, **PushName**, **LoadName**, and **SelectBuffer** (section 21.2); **FeedbackBuffer** and **PassThrough** (section 21.3); and all associated state.
- Display lists - **NewList**, **EndList**, **CallList**, **CallLists**, **ListBase**, **GenLists**, **IsList**, and **DeleteLists** (section 21.4); all references to display lists and behavior when compiling commands into display lists elsewhere in the specification; and all associated state.
- Hints - the `PERSPECTIVE_CORRECTION_HINT`, `POINT_SMOOTH_HINT`, `FOG_HINT`, and `GENERATE_MIPMAP_HINT` targets to **Hint** (section 21.5).
- Attribute stacks - **PushAttrib**, **PushClientAttrib**, **PopAttrib**, **PopClientAttrib**, the `MAX_ATTRIB_STACK_DEPTH`, `MAX_CLIENT_ATTRIB_STACK_DEPTH`, `ATTRIB_STACK_DEPTH`, and `CLIENT_ATTRIB_STACK_DEPTH` state, the client and server attribute stacks, and the values `ALL_ATTRIB_BITS` and `CLIENT_ALL_ATTRIB_BITS` (section 21.6).
- Unified extension string - `EXTENSIONS` target to **GetString** (section 22.2).
- Token names and queries - all token names and queries not otherwise mentioned above for deprecated state, as well as all query entry points where all valid targets of that query are deprecated state (chapter 22 and the state tables).

Appendix F

Version 4.2

OpenGL version 4.2, released on August 8, 2011, is the fourteenth revision since the original version 1.0.

Separate versions of the OpenGL 4.2 Specification exist for the *core* and *compatibility* profiles described in appendix E, respectively subtitled the "Core Profile" and the "Compatibility Profile". This document describes the Compatibility Profile. An OpenGL 4.2 implementation *must* be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 4.2 compatibility and core profiles are upward compatible with the OpenGL 4.1 compatibility and core profiles, respectively.

Following are brief descriptions of changes and additions to OpenGL 4.2.

F.1 New Features

New features in OpenGL 4.2, including the extension or extensions if any on which they were based, include:

- Support for BPTC compressed textures (`ARB_texture_compression_bptc`).
- Allow pixel storage parameters to affect packing and unpacking of compressed textures (`ARB_compressed_texture_pixel_storage`).
- Shader atomic counters (`ARB_shader_atomic_counters`).
- Immutable texture images (`ARB_texture_storage`).

- Instanced transformed feedback drawing (ARB_transform_feedback_instanced).
- Allow the offset within buffer objects used for instanced rendering to be specified (ARB_base_instance).
- OpenGL Shading Language built-in functions allowing loads from and stores to texture images from any shader stage, and application control over the ordering of image load/store operations relative to other OpenGL pipeline operations accessing the same memory (ARB_shader_image_load_store).
- New OpenGL Shading Language features with no OpenGL API impact (ARB_conservative_depth and ARB_shading_language_420pack - see the OpenGL Shading Language Specification for details).
- Queries for sample counts available for a given internal format and usage (ARB_internalformat_query).
- More restrictive alignment constraints for mapped buffers (ARB_map_buffer_alignment).

F.2 Deprecation Model

The following features are newly deprecated by the OpenGL 4.2 core profile:

- The query targets NUM_COMPRESSED_TEXTURE_FORMATS and COMPRESSED_TEXTURE_FORMATS (see section 8.5).

Features deprecated by OpenGL 4.1 remain deprecated, but have not yet been removed.

F.3 Changed Tokens

New token names are introduced to be used in place of old, less general names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table F.1. Note that COPY_READ_BUFFER and COPY_WRITE_BUFFER continue to be used as buffer *target*s for e.g. **BindBuffer**; the _BINDING forms are used only when querying the buffer object bound to those targets.

New Token Name	Old Token Name
COPY_READ_BUFFER_BINDING	COPY_READ_BUFFER
COPY_WRITE_BUFFER_BINDING	COPY_WRITE_BUFFER
TRANSFORM_FEEDBACK_ACTIVE	TRANSFORM_FEEDBACK_BUFFER_ACTIVE
TRANSFORM_FEEDBACK_PAUSED	TRANSFORM_FEEDBACK_BUFFER_PAUSED

Table F.1: New token names and the old names they replace.

F.4 Change Log for Released Specifications

Changes in the specification update of January 19, 2012:

- Corrections to figure 3.1 (Bug 7997).
- Minor bugfixes and typos in sections 3, 10.3, 11.1, 11.1.1, 11.1.3.5, 4.2, 13.3.3, 14.5.2 (restored description of non-antialiased wide line rendering to the core profile since they are deprecated, but not yet removed), 8.2 (fixed prototypes for **SamplerParameter** commands), 15.2.1, 17.3.10 (specify that multisample buffer is only resolved at this time if the default framebuffer is bound), 9.2.8 (correct limits on *layer* for different types of attached textures), 9.4.2, 8.11 (remove redundant description by **IsTexture** that unbound object names created by **GenTextures** are not the names of texture objects), 23 (add **GetInteger64v** as a supported state query), chapter 5, and tables 23.42, 23.43, 23.67, and 23.84 (Bug 7895).
- Add missing automatic unbinding of previously bound buffer objects for **BindBufferRange** and **BindBufferBase** in section 6.1.1 (Bug 8196).
- More clearly specify interface matching rules for shader inputs and outputs in section 7.4.1, for cases where both sides of an interface are found in the same program and where they are in different programs (Bug 7030).
- Clarify in section 11.1.1 that `dvec3` and `dvec4` vertex shader inputs consume only a single attribute location for the purpose of matching inputs to generic vertex attributes, but may consume two vectors for the purposes of determining if too many attribute vectors are used (Bug 7809). Also, add missing language describing the set of attributes consumed by matrix vertex attributes, with fixes to explicitly address `dmat*` types.
- Remove dangling references to nonexistent `gl_VerticesOut` in section 11.2.1.2.3 (Bug 8357).

- Fix names of cube map sampler type tokens in table 7.3 (Bug 8303).
- Fix behavior of **DeleteTransformFeedbacks** in section 13.3.1 to generate an error if any of the objects being deleted has transform feedback active (Bug 8323).
- Remove ambiguity in the order of operations and which vertices are appended by transform feedback when it is resumed in section 13.3.2 (Bug 8202).
- Updated description of errors resulting from specifying texture images of level 1 or greater which exceed implementation-dependent limits, in sections 8.5 and 8.17.3 (Bug 8210).
- Remove clamping of D_t and D_{ref} prior to depth texture comparison in section 8.23.1, since it doesn't reflect hardware reality (Bug 7975).
- Update description of texture access from shadow samplers in section 15.2.1 to interact with texture swizzle (Bug 7962) and clarify that swizzling is not performed on the results of incomplete texture lookups (Bug 7917).
- Add buffer clearing to the list of operations affected by scissor rectangle zero in section 14.9.2 (Bug 8368).
- Clarify that the initial state of `SAMPLE_MASK_VALUE` is for all bits to be set in table 23.16 (Bug 8441).
- Add missing `PROGRAM_SEPARABLE` state to table 23.43 (Bug 8442).
- Add missing `CONVENTION` to `QUADS_FOLLOW_PROVOKING_VERTEX_CONVENTION` token in section 13.6 and table 23.81 (Bug 8432).
- Numerous minor fixes to state table type fields and formatting (Bugs 8430, 8431).
- Clarified that automatic unbinding of deleted objects, as described in section 5.1.2, does not affect attachments to unbound container objects the deleted objects are themselves attached to (Bug 8233).
- Add version in which several extensions were introduced to core GL in section K.3 (Bug 8418).

Changes in the specification update of August 22, 2011:

- More clearly specify interface matching rules for shader inputs and outputs in section 7.4.1, for cases where both sides of an interface are found in the same program and where they are in different programs (Bug 7030).
- Clarify in section 11.1.1 that `dvec3` and `dvec4` vertex shader inputs consume only a single attribute location for the purpose of matching inputs to generic vertex attributes, but may consume two vectors for the purposes of determining if too many attribute vectors are used (Bug 7809). Also, add missing language describing the set of attributes consumed by matrix vertex attributes, with fixes to explicitly address `dmat*` types.

Changes in the released specification of August 8, 2011:

- Update name of `MIN_MAP_BUFFER_ALIGNMENT` to follow GL conventions in section 6.3 and table 23.67 (Bug 7825).
- Change query object state description in section 4.2 so the initial state of the query result available flag agrees with the state table (Bug 7823).
- Minor cleanups to atomic counter language in section 7.6 and to atomic counter token names in tables 23.69, 23.70, 23.72, and 23.73 (Bug 7834).
- Clarify that completeness affects texture lookup and fetch operations in all shader stages in section 8.17 (Bug 7856).
- Change **BindImageTexture** parameter name from *index* to *unit* and fix minor language issues in section 8.26 (Bugs 7744, 7850, 7851).
- Fix typos in section 22.3 (Bug 7843).
- Fix minimum maximums for `MAX_FRAGMENT_IMAGE_UNIFORMS` and `MAX_COMBINED_IMAGE_UNIFORMS` in table 23.77 (Bug 7805).
- Change minimum maximum for `MAX_ATOMIC_COUNTER_BUFFER_SIZE` to 32 in table 23.76 (Bug 7855).

F.5 Credits and Acknowledgements

OpenGL 4.2 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 4.2, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of

new ARB extensions together with OpenGL 4.2. In addition, many people participated in developing earlier vendor and `EXT` extensions on which the OpenGL 4.2 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Acorn Pooley, NVIDIA
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benji Bowman, Imagination Technologies
Bill Licea-Kane (Chair, ARB OpenGL Shading Language TSG, `ARB_shader_atomic_counters`)
Bruce Merry, ARM (Detailed specification review, `ARB_texture_storage`)
Chris Dodd, NVIDIA
Christophe Riccio, Imagination Technologies
Daniel Koch (`ARB_internalformat_query`)
Eric Werness, NVIDIA (`ARB_texture_compression_bptc`)
Graham Sellers, AMD (`ARB_base_instance`, `ARB_conservative_depth`, `ARB_transform_feedback_instanced`)
Greg Roth, NVIDIA
Ian Romanick, Intel (`ARB_texture_storage`)
Jacob Ström, Ericsson AB
Jan-Harald Fredriksen (`ARB_internalformat_query`)
Jeannot Breton, NVIDIA
Jeff Bolz, NVIDIA Corporation (`ARB_shader_image_load_store`)
Jeremy Sandmel, Apple
John Kessenich, Independent (OpenGL Shading Language Specification Editor, `ARB_shading_language_420pack`)
Jon Leech, Independent (OpenGL API Specification Editor)
Lingjun (Frank) Chen, Qualcomm
Mark Callow, HI Corporation
Maurice Ribble, Qualcomm
Nick Haemel, AMD
Pat Brown, NVIDIA Corporation (`ARB_shader_image_load_store`, `ARB_shading_language_packing`)
Patrick Doane, Blizzard
Pierre Boudier, AMD
Piers Daniell, NVIDIA Corporation (`ARB_compressed_texture_pixel_storage`, `ARB_map_buffer_alignment`)
Robert Simpson, Qualcomm
Tom Olson, ARM (Chair, Khronos OpenGL ES Working Group)

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

Appendix G

Version 4.3

OpenGL version 4.3, released on August 6, 2012, is the fifteenth revision since the original version 1.0.

Separate versions of the OpenGL 4.3 Specification exist for the *core profile* and *compatibility profile* described in appendix E, respectively subtitled the "Core Profile" and the "Compatibility Profile". This document describes the Compatibility Profile. An OpenGL 4.3 implementation *must* be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 4.3 compatibility and core profiles are upward compatible with the OpenGL 4.2 compatibility and core profiles, respectively (see appendix F).

Following are brief descriptions of changes and additions to OpenGL 4.3. Descriptions of changes and additions in versions of OpenGL prior to 4.2 are omitted in this Specification, but may be found in the OpenGL 3.0 Specification (for features in versions 1.0 - 3.0, inclusive) and the OpenGL 4.2 Specification (for features in versions 3.1 - 4.1, inclusive). These Specifications are available in the OpenGL Registry.

G.1 Restructuring

The Specification has been substantially restructured to introduce high-level concepts and describe objects before their use, and more cleanly split descriptions of programmable and fixed-function processing. Chapter and section numbering has been aligned between the two profile Specifications so that a section number will always refer to the same concept in both profiles (although that section may be

empty in the core profile).

G.2 New Features

New features in OpenGL 4.3, including the extension or extensions if any on which they were based, include:

- `ARB_arrays_of_arrays` (OpenGL Shading Language only)
- `ARB_ES3_compatibility`
- `ARB_clear_buffer_object`
- `ARB_compute_shader`
- `ARB_copy_image`
- `ARB_debug_output`
- `ARB_explicit_uniform_location`
- `ARB_fragment_layer_viewport` (OpenGL Shading Language only)
- `ARB_framebuffer_no_attachments`
- `ARB_internalformat_query2`
- `ARB_invalidate_subdata`
- `ARB_multi_draw_indirect`
- `ARB_program_interface_query`
- `ARB_robust_buffer_access_behavior`
- `ARB_shader_image_size` (OpenGL Shading Language only)
- `ARB_shader_storage_buffer_object`
- `ARB_stencil_texturing`
- `ARB_texture_buffer_range`
- `ARB_texture_query_levels`
- `ARB_texture_storage_multisample`

- ARB_texture_view
- ARB_vertex_attrib_binding
- KHR_debug
- Add VERTEX_ATTRIB_ARRAY_LONG query for whether a vertex attribute is stored as an unconverted double (Bug 8272).
- Add queries for #version strings of all OpenGL Shading Language versions supported by the GL (Bug 7811).
- Increase required number of uniform blocks per program stage from 12 to 14 (Bug 8891).

G.3 Deprecation Model

The following features are deprecated by the OpenGL 4.3 core profile.

- Bitmap pack/unpack state for bitmaps - the pixel pack parameters UNPACK_LSB_FIRST PACK_LSB_FIRST and (see sections 8.4.1 and 18.2).

The following features which were previously deprecated have been reintroduced to the OpenGL 4.3 core profile:

- The **GetPointerv** command (see section 22.2) and the STACK_OVERFLOW and STACK_UNDERFLOW errors (see table 2.3). These features are used by the debug functionality in chapter 20.

Other features deprecated by OpenGL 4.2 remain deprecated, but have not yet been removed.

G.4 Changed Tokens

New token names are introduced to be used in place of old, less general names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table G.1.

New Token Name	Old Token Name
MAX_COMBINED_SHADER_OUTPUT_RESOURCES	MAX_COMBINED_IMAGE_UNITS_AND_FRAGMENT_OUTPUTS

Table G.1: New token names and the old names they replace.

G.5 Change Log for Released Specifications

Changes in the specification update of February 14, 2013:

- Do not perform validity checks on the **BindBufferRange** *size* and *offset* arguments when a zero *buffer* is specified to unbind a buffer, in section 6.1.1 (Bug 9765).
- Clean up descriptions of **BindBufferBase** in section 6.1.1 so it is described without reference to **BindBufferRange**, and note in section 6.7.1 that a zero size query result for a buffer binding is a sentinel indicating the entire buffer is bound (Bug 9513).
- Fix typo in error descriptions in section 6.7 (Bug 9720).
- Update section 6.8 to reference the tables of buffer binding state of different types, and move uniform buffer binding state from table 23.46 to new table 23.60 to match (Bug 9566).
- Clarify that **Uniform*d** cannot be used to load uniforms with `boolean` types in section 7.6.1 (Bug 9345).
- Added double-precision matrix types to the description of uniform buffer object storage layouts in section 7.6.2.1, and cleaned up description of the matrix stride and how to query it (Bug 9375).
- Correct off-by-one error for valid range of sampler values in introduction to section 7.11 (Bug 8905).
- Clarify in section 7.15 that table 23.54 is not part of program object state, and update the table caption to match (Bug 9781).
- Clarify description of the *data* argument to **TexSubImage*** in section 8.6 so that it may not be `NULL`, unlike **TexImage*** (Bug 9750).
- Fix typo in description of **TexParameter*** in section 8.10 (Bug 9625).

- Add a color-renderable column to table 8.19 and modify section 9.4 to define color-renderable formats with respect to the table, rather than with respect to base formats. This results in the `RGB9_E5` format no longer being color-renderable, which was an error (Bug 9338).
- Allow vector forms of **TexParameter*** to be used to set scalar parameters in section 8.10, reversing an old spec change made in error (vector parameters, however, still cannot be set with the scalar calls) (Bug 7346).
- Restore missing clamp for D_t and D_{ref} (depth texture comparison mode parameters) in section 8.23.1 when using a fixed-point texture (Bug 7975).
- Correct exp_{shared} to exp_s terminology and include missing N term when describing shared exponent texture color conversion and final conversion in sections 8.25 and 18.2.8 (Bug 9486).
- Specify that `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER` for **GetFramebufferParameteriv** in section 9.2.3 (Bug 9344).
- Added `STENCIL_INDEX8` as a required stencil-only renderbuffer format in sections 9.2.5 and 9.4.3, for compatibility with OpenGL ES 3.0 (Bug 9418).
- Fixes to description of isoline tessellation in section 11.2.2.3 to describe use of outer tessellation levels in the correct order (Bug 9607).
- Clamp values at specification time for **DepthRange*** (section 13.8.1) and **ClearDepth** (section 17.4.3), to avoid subtle issues when using floating-point depth buffers. However, this change does not reintroduce use of the `clampf` and `clampd` types eliminated in OpenGL 4.2 (Bug 9517).
- Change **DrawBuffer** error for `COLOR_ATTACHMENT`m out of range from `INVALID_VALUE` to `INVALID_OPERATION` in section 17.4.1, to match **DrawBuffers** and OpenGL ES 3.0 (Bug 8568).
- Modify language describing buffer writes in section 17.4.1 so that fragment colors are not written only to draw buffers with no color attachment, or with `NONE` as the draw buffer, allowing writes to other draw buffers to succeed. Specify that when only some output variables are written, only the fragment colors corresponding to unwritten variables are undefined (Bug 9494).
- Allow attachment parameters to **InvalidateSubFramebuffer** in section 17.4.4 to include `DEPTH_STENCIL_ATTACHMENT` (Bug 9480).

- Specify that the **BlitFramebuffer** *mask* may be zero in section 18.3.2 (Bug 9748).
- Cleaned up language describing parameters to **DebugMessageControl** in section 20.4 to avoid triple negatives (Bug 9392).
- Increase minimum value for `MAX_UNIFORM_BUFFER_BINDINGS` to 84 in table 23.75 to account for correct number of bindings/stage (14) (Bug 9424).

Changes in the released Specification of August 6, 2012:

- Restructured as described in section G.1.
- Added new features as described in section G.2.
- Add title image page using the "pipeline metro" diagram.
- Miscellaneous minor typos and fixes to better match OpenGL ES 3.0 spec language (Bugs 7885, 7904, 7919).
- Changed "rectangular texture" to "rectangle texture" throughout the spec for consistency (Bug 9262). Other consistency changes including using "equivalent to" consistently for pseudocode samples defining the operation of a command.
- Many cleanups and additions to error language throughout the spec to add previously implicit errors explicitly (however, this is still a work in progress). In particular, added explicit errors for all commands taking *program* or *shader* arguments as described at the start of section 7.1, and for commands taking *shadertype* arguments (Bug 9145); and added explicit `INVALID_VALUE` errors for negative values of `sizei` and `sizeiptr` arguments (Bug 9320).
- Cleaned up description of function prototypes from the old `T` notation to `T *` or `const T *` as appropriate for the actual C binding of the corresponding command.
- Added `NUM_SHADING_LANGUAGE_VERSIONS` and `SHADING_LANGUAGE_VERSION` queries for supported GLSL `#version` strings in sections 1.3.1, 1.3.3 and 22.2, and in table 23.68 (see Bug 7811). Still need enum assignments for these.
- Remove assertion that draw and read framebuffers must be of the same class in section 2.1 (Bug 9134).

- Clarify in the caption to table 2.2 that `sync` is defined as a pointer type in the C binding (Bug 9140).
- Reintroduced `STACK_OVERFLOW` and `STACK_UNDERFLOW` errors to the core profile in table 2.3, since they are used by the debug group APIs (Bug 9158).
- Describe new, more complete error summary and typesetting style in section 2.3.1. Convert (most) error summaries beginning with section 4.1, adding implicit error conditions that have not been described with the commands they apply to before. This is a work in progress.
- Clean up query objects in section 4.2 to clarify that `TIME_ELAPSED` and `TIMESTAMP` queries are different type of queries, and remove an inapplicable error condition for `TIMESTAMP` queries (Bug 9268).
- Add language to **DeleteBuffers** in section 6 and **BufferData** in section 6.2 specifying that these commands cause any existing mappings of a buffer being operated on in any context to be unmapped, per a rather offhand reference in section 6.3.1 (Bug 9323).
- Restore `COPY_READ_BUFFER` and `COPY_WRITE_BUFFER` as buffer *target* names in sections 6.1 and 6.6. The `_BINDING` aliases are used only when querying those binding points (Bugs 8475, 9115).
- Bring compute shader language in sync with changes to the extension spec. In particular, add `DISPATCH_INDIRECT_BUFFER` binding section 6.1, describe it in section 10.3.11, update **DispatchComputeIndirect** to use it in section 19, add new table 23.63, and update aggregate shader limits in section 23.75 (Bug 9130).
- Add create-on-bind behavior for **BindBufferRange** and **BindBufferBase** in section 6.1.1, mirroring **BindBuffer** (Bug 9216).
- Clarify that offset and alignment constraints for **ClearBufferSubData** in section 6.2.1 are based on the total size of a texel of type *internalformat* (size of base type times no. of components) (Bug 9211).
- Update errors for **ClearBuffer*Data** and mention them and **InvalidateBuffer*Data** among the commands that can modify buffer object storage in sections 6.2.1, 7.6, 7.8, and 7.13 (Bug 9154).
- Clarify that buffer mappings are not affected by whether or not a context is current in section 6.3.1 (Bug 9323).

- Add language in section 6.3.2 specifying that commands which write to (as well as read from) mapped buffers are also supposed to generate errors (Bug 9115).
- Make **InvalidateBuffer*Data** generate errors for invalid object handles in section 6.5 (Bug 9341).
- Merge description of different types of indexed array buffer bindings into section 6.7.1, and move description of *target*-specific **BindBufferRange** errors into section 6.1.1 with reference to section 6.7.1 (Bug 9115 and general cleanup).
- Extend **ShaderBinary** in section 7.2 to allow support for shader binary formats including all shader types, not just vertex and fragment shaders (Bug 9282).
- Add description of "top-level arrays" to active shader storage block discussion in section 7.3.1 (Bug 9115). This probably needs to migrate back to the extension as well, along with a few other language changes in this section which Pat suggested in his PDF review but hasn't put into the extension yet.
- Clarify error descriptions for **UseProgramStages** and **ActiveShaderProgram** (section 7.4), **UseProgram** (section 7.6.1), and **ProgramUniform*** (section 7.6.1) to generate an `INVALID_OPERATION` error "if *program* has not been linked, or was last linked unsucessfully" rather than "if *program* has not been successfully linked" (Bug 8640, tracking similar changes to other commands previously).
- Merge similar descriptions of uniform variable component limits for each separate shader stage into section 7.6.
- Fix nonexistent token `ATOMIC_COUNTER_ARRAY_STRIDE` to `UNIFORM_ARRAY_STRIDE` in section 7.7.1 (Bug 9346).
- Removed redundant definition of **GetSubroutineUniformLocation** from the beginning of section 7.10.
- Added `INVALID_VALUE` error in section 7.11 if **Uniform1i{v}** is used to set a sampler to a value less than zero or greater than or equal to the value of `MAX_COMBINED_TEXTURE_IMAGE_UNITS`, matching the similar error for setting image uniforms.
- Add `VERTEX_ATTRIB_ARRAY_LONG` state in section 7.14 and table 23.6 (Bug 8272).

- Add all specific compressed texture formats to the required format list in section 8.5.1. Include EAC and ETC2 format in specific format language in table 8.22 and sections 8.6, 8.7, and 8.24 (Bug 9156).

- Add additional `INVALID_OPERATION` errors depending on odd combinations of read buffer and FBO state for **CopyTexImage*** and **CopyTexSubImage*** in section 8.6 (Bug 8559).

- Disallow **CopyTexImage*** between sRGB and linear formats in section 8.6, and define **BlitFramebuffer** to linearize sRGB data from the read buffer in section 18.3.2 (Bug 8560).

- Allow multisample texture targets as arguments to **TexParameter*** in section 8.10, with additional error conditions when attempting to set a disallowed min filter or base level parameter value.

- Replace listings of all six cube map face selection targets with references to tables 8.27 or 9.3, in several places throughout the spec.

- Fix error generated for invalid texture handle passed to **TextureView** in section 8.18 (Bug 9337).

- Tweaked descriptions of transferring vertices in sections 10.3.4 and 10.4 to more closely match OpenGL ES 3.0 (Bug 8686).

- Restore missing description of **DrawElementsInstanced** in section 10.4.

- Renamed the formal parameter *primcount* to *instancecount* for **DrawArraysInstancedBaseInstance**, **DrawArraysInstanced**, **DrawElementsInstancedBaseInstance**, **DrawElementsInstanced**, **DrawElementsInstancedBaseVertex**, and **DrawElementsInstancedBaseVertexBaseInstance** (section 10.4), and for **DrawTransformFeedbackInstanced** and **DrawTransformFeedbackStreamInstanced** (section 13.3.3). Used equivalent terminology in the pseudocode descriptions of **DrawElementsIndirect** and **DrawArraysIndirect** (section 10.4). Renamed the formal parameter *primcount* to *drawcount* for **MultiDrawArrays**, **MultiDrawArraysIndirect**, **MultiDrawElements**, **MultiDrawElementsIndirect**, **MultiDrawElementsBaseVertex** (section 10.4) (Bug 9230).

- Moved description of **GetVertexAttrib*** into section 10.5 (Bug 9115).

- Specify in section 11.1.1 that special built-in inputs and outputs such as `gl_VertexID` should be enumerated in the `PROGRAM_INPUT` and `PROGRAM_OUTPUT` interfaces, as well as the legacy function **GetActiveAttrib**. Add spec language counting the built-ins `gl_VertexID` and `gl_InstanceID` against the active attribute limit (Bug 9201).

- Swap order of tessellation levels in describing isoline tessellation in section 11.2.2.3, to match actual hardware (Bug 9195).

- Remove language about deferred deletion for **DeleteTransformFeedbacks** in section 13.3.1 (Bug 8948).

- Add transform feedback-related error for **ProgramBinary**, matching existing error for **LinkProgram** in section 13.3.2 when *program* is the name of a program being used by one or more transform feedback objects (Bug 7928).

- Add description of `MAX_COMPUTE_SHARED_MEMORY_SIZE` in section 19.1, lifted from GLSL spec (Bug 9069).

- Add description of the type of the debug callback function, including platform-dependent calling conventions, in section 20.2.

- Cleaned up section 21.1 to use a naming scheme for parameters that matches the names of evaluator command parameters in `gl.h`.

- Remove inaccurate description of GLSL version string sort order in section 22.2. Instead, ensure that the most recent GLSL version corresponding to the context profile is returned first, and other entries have no defined ordering (Bug 7811).

- Change Z_n terminology used in state tables to describe enumerated state with n possible values to E throughout, since maintaining the n was always tricky as features were added and the possible values are fully described in the spec body. This affects hundreds of state table entries as well as adding a description of E in table 23.1.

- Move `IMPLEMENTATION_COLOR_READ_FORMAT` and `IMPLEMENTATION_COLOR_READ_TYPE` from table 23.85 to table 23.64 since they are not framebuffer-dependent values, unlike OpenGL ES (Bug 8561).

- Increased minimum values for `MAX_VERTEX_UNIFORM_BLOCKS`, `MAX_TESS_CONTROL_UNIFORM_BLOCKS`, `MAX_TESS_EVALUATION_`-

UNIFORM_BLOCKS, MAX_GEOMETRY_UNIFORM_BLOCKS, MAX_FRAGMENT_UNIFORM_BLOCKS, and MAX_COMPUTE_UNIFORM_BLOCKS to 14 in tables 23.69, 23.70, 23.71, 23.72, 23.73, and 23.74 respectively, and of MAX_COMBINED_UNIFORM_BLOCKS to 70 in table 23.75 (Bug 8891).

- Added UNPACK_LSB_FIRST and PACK_LSB_FIRST state to the deprecated features list in section G.3 (Bug 7865).

G.6 Credits

OpenGL 4.3 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 4.3, including the company that they represented at the time of their contributions, follow.

Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 4.3. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 4.3 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Registry.

Aaron Plattner, NVIDIA
Acorn Pooley, NVIDIA
Ahmet Oguz Akyuz, AMD
Alex Eddy, Apple Inc
Anton Staaf, Google
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benj Lipchak, Apple
Benjamin Morris, NVIDIA
Bill Licea-Kane (Chair, ARB OpenGL Shading Language TSG)
Brent Wilson, NVIDIA
Bruce Merry, Independent
Chris Marrin, Apple
Chris Niederauer, Apple Inc
Christophe Riccio, AMD (ARB_debug_group, ARB_debug_label, ARB_shader_image_size, ARB_texture_query_levels, KHR_debug_output)
Dan Omachi, Apple Inc
Daniel Koch, TransGaming Inc. (ARB_internalformat_query2)
Daniel Rakos, AMD

Eric Werness, NVIDIA
Georg Kolling, Imagination Technologies
Graham Sellers, AMD (ARB_multi_draw_indirect, ARB_clear_buffer_object, ARB_compute_shader, ARB_copy_image, ARB_texture_buffer_range, ARB_texture_storage_multisample)
Greg Roth, NVIDIA
Henri Verbeet, CodeWeavers
Jaakko Konttinen, AMD (ARB_debug_output)
James Jones, NVIDIA
Jan-Harald Fredriksen, ARM
Jason Green, TransGaming
Jean-François Roy, Apple
Jeff Bolz, NVIDIA (ARB_invalidate_subdata, ARB_texture_view, ARB_vertex_attrib_binding)
Joe Kain, NVIDIA
John Kessenich, Independent (OpenGL Shading Language Specification Editor, ARB_arrays_of_arrays)
Jon Leech, Independent (OpenGL API Specification Editor)
Kenneth Russell, Google (ARB_robustness_isolation)
Kent Miller, Apple
Lingjun (Frank) Chen, Qualcomm
Mark Callow, HI Corporation
Mark Kilgard, NVIDIA (ARB_robustness)
Mark Young, AMD
Mathias Schott, NVIDIA
Matt Collins, Apple
Maurice Ribble, Qualcomm
Michael Gold, NVIDIA (ARB_copy_image)
Michael Morrison, NVIDIA
Pat Brown, NVIDIA (ARB_framebuffer_no_attachments, ARB_program_interface_query, ARB_shader_storage_buffer_object)
Pierre Boudier, AMD
Piers Daniell, NVIDIA (ARB_ES3_compatibility, ARB_debug_output2, ARB_explicit_uniform_location, ARB_fragment_layer_viewport, ARB_robust_buffer_access_behavior, ARB_stencil_texturing)
Richard Schreyer, Apple
Seth Sowerby, Apple
Thomas Volk, NVIDIA

Tim Johansson, Opera
Vladimir Vukicevic, Mozilla
Yaki Tebeka, Graphic Remedy
Yuan Wang, IMG

G.7 Acknowledgements

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

The “pipeline metro” cover image was created by Dominic Agoro-Ombaka of Gold Standard Group.

Appendix H

Version 4.4

OpenGL version 4.4, released on July 22, 2013, is the sixteenth revision since the original version 1.0.

Separate versions of the OpenGL 4.4 Specification exist for the *core profile* and *compatibility profile* described in appendix E, respectively subtitled the "Core Profile" and the "Compatibility Profile". This document describes the Compatibility Profile. An OpenGL 4.4 implementation *must* be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 4.4 compatibility and core profiles are upward compatible with the OpenGL 4.3 compatibility and core profiles, respectively (see appendix G).

Following are brief descriptions of changes and additions to OpenGL 4.4. Descriptions of changes and additions in versions of OpenGL prior to 4.2 are omitted in this Specification, but may be found in the OpenGL 3.0 Specification (for features in versions 1.0 - 3.0, inclusive) and the OpenGL 4.2 Specification (for features in versions 3.1 - 4.1, inclusive). These Specifications are available in the OpenGL Registry.

H.1 New Features

New features in OpenGL 4.4, including the extension or extensions if any on which they were based, include:

- `ARB_buffer_storage`
- `ARB_clear_texture`

- `ARB_enhanced_layouts`
- `ARB_multi_bind`
- `ARB_query_buffer_object`
- `ARB_texture_mirror_clamp_to_edge`
- `ARB_texture_stencil8`
- `ARB_vertex_type_10f_11f_11f_rev`
- New implementation-dependent state `MAX_VERTEX_ATTRIB_STRIDE`, which constrains the maximum value of *stride* parameters to vertex array pointer-setting commands.

H.2 Deprecation Model

No new features are deprecated, and no previously deprecated features are re-introduced by the OpenGL 4.4 core profile.

Features deprecated by OpenGL 4.3 remain deprecated, but have not yet been removed.

H.3 Change Log for Released Specifications

Changes in the released Specification update of March 19, 2014:

- Fix typos and apply minor non-semantic language changes in several places to more closely match the OpenGL ES 3.1 Specification.
- Tweak **Enable / Disable** language for different *target*s throughout the spec for more consistency.
- Modify language in section 5.1.2 so that binding-related state is restored to default values after automatic unbinds (Bugs 10076, 11630).
- Add samplers to the list of object types in section 5.1.3 for which deletion is delayed until the object is no longer in use, and remove transform feedback objects since deletion is defined to generate an error while they are active (Bugs 11374, 10079).
- Add atomic counter binding, offset and stride assignments to the values reset by **ProgramBinary** in section 7.5 (bugfix from ES spec).

- Move **Uniform*** errors to section 7.6.1 where the **Uniform*** commands are defined, from sections 7.11 and 7.12
- Correct typo `MAX_ATOMIC_COUNTER_BUFFERS` to `MAX_COMBINED_ATOMIC_COUNTER_BUFFERS` in section 7.7, and specify new program interface query for atomic counter buffer data size query in section 7.7.2
- Restrict error for **UniformSubroutinesuiv** in section 7.10 to the case where no program stage is active for the shader stage identified by *shadertype* (Bug 11306).
- Update **DeleteSamplers** language in section 8.2 to allow for the case where a sampler is bound to multiple texture units (bugfix from ES spec).
- Fix conversion equation reference for **SamplerParameteriv** in section 8.2. Reorganize descriptions of exceptions to default data conversion rules here and in section 8.10 for **TexParameter*** (Bug 11185).
- Add error in section 8.2 for calling scalar **SamplerParameter{if}** entry points with non-scalar *param* tokens, matching **TexParameter*** (Bug 11186).
- Add error for **TexImage3D** in section 8.5 when specifying invalid *border* or *target* values with specific compressed texture formats, matching **CompressedTexImage3D** (Bug 11239).
- Update table 8.19 in section 8.5.2 to make all remaining sized color formats texture-renderable that were not already, since the component size promotion rules mean there are already required texture formats with the same format and as least as many bits/component (Bug 11097).

  We discussed but did not also make these formats required renderbuffer formats, because doing this might imply a format change when a format supported at exactly the required component size for textures is only supported at a larger component size for renderbuffers (e.g. create texture at `RGB4`, use as a texture, then use as a renderbuffer and find it has to be converted to `RGB565`).
- Change maximum allowed texture size for **TexImage3DMultisample** in section 8.8 to the value of `MAX_ARRAY_TEXTURE_LAYERS` (Bug 11135).
- Remove explicit `OUT_OF_MEMORY` errors from **TexImage*Multisample** in section 8.8 and **RenderbufferStorageMultisample** in section 9.2.4. (public Bug 952).

- Add subsection headings and Errors sections for **GetTexParameter*** and **GetTexLevelParameter*** in section 8.11.
- Change formal parameter names *value* and *data* to *pname* and *params*, following the headers and man pages, for **GetTexParameter*** and **GetTexLevelParameter*** (see section 8.11). Also make this change for **GetTexGen*** (see section 12.1.3.1), **GetLight*** and **GetMaterial*** (see section 12.2.4), and **GetTexEnv*** (see section 16.1.1). (Bug 11523).
- Added per-*target* maximum level-of-detail values to definition of **GetTexLevelParameter*** in section 8.11.3 (Bug 11136)
- Add error for **GetTexLevelParameter*** in section 8.11 when querying multisample-specific parameters for non-multisample textures (this was imported from the OpenGL ES spec and it's not certain it should be here).
- Define behavior of **GetTexLevelParameter*** in section 8.11.3 for queries of multisample state from non-multisampled textures (Bug 11814).
- Change rounding mode for layer numbers of array textures in section 8.14.2 to prefer round-to-nearest-even, while still allowing old spec behavior (Bug 11399).
- Add cube map array texel arrays to the enumerated state in section 8.22, and remove redundant description of the bound buffer texture object name.
- Add description of `DEPTH_STENCIL_TEXTURE_MODE` in section 8.22, and correct its type in table 23.20 (Bug 11770).
- Add section 8.26.1 summarizing image unit binding state (Bug 10076).
- Restructure error condition for **FramebufferParameteri** in section 9.2.1 to avoid ambiguity (Bug 11831).
- Define **GetFramebufferAttachmentParameteriv** in section 9.2.3 to return `NONE` when querying the object type of depth or stencil attachments, the default framebuffer is bound, and the corresponding buffer of the default framebuffer has zero bits (Bug 10908).
- Clean up language describing the **GetFramebufferAttachmentParameteriv** query for `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` in section 9.2.3 (Bug 11102).

- Remove bogus error condition for **FramebufferRenderbuffer**, and add errors for invalid *layer* depending on texture type in section 9.2.7 (bugfix from ES spec).
- Set the vertex attribute array pointer state explicitly in the pseudocode for **VertexAttrib*Pointer** in section 10.3.2, and define queries of `VERTEX_ATTRIB_ARRAY_BUFFER_BINDING` and `VERTEX_ATTRIB_ARRAY_DIVISOR` to look up the corresponding state via the vertex attribute binding in section 10.5 (Bug 11789).
- Rewrite section 10.3.11 to merge language for indirect drawing and indirect compute dispatch commands (Bug 11601).
- Fix texel fetch limit on layer for array textures in section 11.1.3.2 (Bug 11770).
- Fix typo in section 11.1.3.7 (Bug 11553).
- Relax restrictions on required shader stages in sections 11.2 and 11.3 so that separable program objects containing tessellation and geometry shaders are not also required to contain a vertex shader (Bug 11508).
- Modify tessellation primitive generation language in section 11.2.2 to discard patches only when **relevant** outer tessellation levels are NaNs, but not for non-relevant levels (Bug 11484).
- Add missing error for invalid *pname* to **GetMultisamplefv** in section 14.3.1 (Bug 11134).
- Change error for invalid `mode*` parameters to **BlendEquation*** in section 17.3.6.1 to `INVALID_ENUM` (bug 11354).
- Fix error for invalid blending function arguments in section 17.3.6.2 to `INVALID_ENUM` (Bug 11770).
- Add errors for **ClearBuffer*** in section 17.4.3.1 when the wrong type of buffer is passed to different forms of the command (Bug 11139).
- Fix table reference in description of **ClearBuffer*** in section 17.4.3.1 from 17.5 to 17.4 so it's clear that `DRAW_BUFFER`*i* can contain any legal value and the corresponding buffers will be cleared (Bug 11463).
- Modify caption for table 17.4 in section 17.4.1 so it's clear table entries refer only to selected buffers for the draw buffer, and remove redundant description of initial draw buffers state (Bugs 11462, 11463).

- Moved language allowing undefined behavior for overlapping copies to the beginning of section 18.3, extended it to the case of overlapping copies via texture views, applied it to all pixel copy operations in the section, and restricted the effects to undefined pixel values rather than general undefined behavior (Bug 11355).

- Replace the "Get Value" of LABEL with – for debug labels of objects in tables 23.6, 23.9, 23.20, 23.23, 23.30, 23.33, 23.41, 23.42, 23.43, 23.55, 23.59, 23.61, and 23.87 (Bug 11131).

- Clarified description of sourcing generic vertex attributes from client memory in section 10.3.9. Rewrote descriptions of the vertex binding-specific state in table 23.7, added missing VERTEX_BINDING_DIVISOR and VERTEX_BINDING_BUFFER state, and marked all such state as part of the vertex-array attribute group (in the compatibility profile only) (Bug 10736).

- Add missing state for framebuffers with no attachment to tables 23.30 and 23.81 (Bug 11187).

- Add missing COMPUTE_SHADER state to program pipeline objects in table 23.42 (Bug 11539).

Changes in the released Specification update of October 18, 2013:

- Add footnote to section 2.3.1 noting that OUT_OF_MEMORY errors are not explicitly shown in command-specific Error sections because all GL commands can potentially generate them. Remove explicit OUT_OF_MEMORY errors from **BufferStorage**, **BufferData**, and **MapBufferRange** in section 6, and the **TexStorage*** commands in section 8.19 (public Bug 952).

- Add new section 2.4 defining the term *rendering commands*, and modify language using this term in sections 7.11, 9.4.4, 10.3.9, 10.9, and 11.1.3.12 to use the new definition, sometimes narrowed to drawing commands, or to rendering commands which invoke shaders (Bug 10403).

- Add footnote to section 2.6.9 specifying that undefined behavior results when mixing non-shared core API framebuffer objects and shared EXT extension framebuffer objects (Bug 10738).

- Change description of **BufferStorage** MAP_COHERENT_BIT in section 6.2 to say that changes are "visible to any subsequently issued GL commands" rather than "immediately visible" (Public Bug 935).

- Remove error for **BufferStorage** in section 6.2 which did not allow *flags* to contain `MAP_WRITE_BIT` while not also containing `DYNAMIC_STORAGE_BIT` (Bug 10561, public Bug 925).
- Rewrite section 6.3.2 describing the effects of mapped buffers on other commands and possible resulting errors or undefined behavior (Bug 10684).
- Update **GetProgramResourceiv** in section 7.3.1 to return one for `ARRAY_SIZE` queries of non-arrays (for compatibility with **GetActiveUniform**) and zero for explicitly unsized arrays, and describe circumstances in which queries of `ARRAY_SIZE` and `TOP_LEVEL_ARRAY_SIZE` for unknown array sizes may return zero (Bugs 10641, 10647).
- Allow **ActiveShaderProgram** and **UseProgramStages** to accept zero *program* values in section 7.4, to reset the corresponding program pipeline object state to its initial value (Public Bug 871).
- Add additional `const` qualifier to parameter type of *uniformNames* for **GetUniformIndices** in section 7.6 (Bug 10703).
- Fix typo from `SHADER_STORAGE_BLOCK` to `SHADER_STORAGE_BUFFER` for **ShaderStorageBlockBinding** in section 7.8 (Bug 10795).
- Fix list of supported texture targets for stencil, depth, and depth+stencil formats in section 8.5 to include multisample targets (Bug 10558).
- Move descriptions of required texture and renderbuffer/texture formats into tables 8.19 and 8.21, and update language in sections 8.5.1, 9.2.5, 9.4, and 9.4.3 to refer to those tables and to describe required framebuffer formats as color-renderable. Also, merged the no-longer-existent table 9.1 of renderbuffer-only stencil formats into table 8.21, and modified the definition of stencil-renderable in section 9.4 accordingly (Bug 9338).
- Tag type of stencil fields in table 8.21 as *ui* (Bug 10748).
- Add error for **TexImage*DMultisample** in section 8.8 when *internalformat* is not a valid format (Bug 11018).
- Change type of *internalformat* argument to **TexImage2DMultisample** and **TexImage3DMultisample** in section 8.8 to `enum`, since the legacy internal formats are never accepted by these commands (Bug 10496).
- Note that during computation of scale factors in section 8.14.1, implementations have chosen to perform clamping of intermediate terms in the sum of level-of-detail biases differently (Bug 9779).

- Specify how border color values are clamped for compressed texture image formats in table 8.22 and section 8.14.2 (Bug 9476).
- Specify behavior of **GetFramebufferAttachmentParameteriv** in section 9.2.3 when querying texture attachments which have not yet specified a texture image or which do not yet have an allocated image store, and fix the error generated when querying a combined depth+stencil attachment to apply to the component type query, rather than the color encoding query (Bugs 9170, 10357).
- Clean up errors for **FramebufferTexture*D** in section 9.2.8 (Bug 10674).
- Merge rows of table 10.3 for **VertexAttrib*Format** and **VertexAttrib*Pointer** commands which are otherwise identical (Bug 10692).
- Rewrite description of **BindVertexBuffer** in section 10.3.2 to clarify that it may create buffers in the same fashion as **BindBuffer** (Bug 10693).
- Explicitly specify in section 10.3.2 that the **VertexAttrib*Pointer** commands may generate any of the errors defined by **VertexAttrib*Format** and **VertexAttribBinding**, since the pseudocode invokes those commands (Bug 10631). This was implicit in the previous spec language and has simply been called out more explicitly for clarity without replicating the errors.
- Typo fix in error summary for **ClientActiveTexture** in section 10.3.3 (Bug 10673).
- Add `PRIMITIVE_RESTART_FOR_PATCHES_SUPPORTED` to section 10.3.6 and table 23.64 to allow querying primitive restart support for patch primitives (Bug 10364). Note that some implementations already could not support this, but had no way to indicate this.
- Drop bogus references to "disabled attributes" in section 10.3.7 (Bug 10695).
- Generate `INVALID_VALUE` errors consistently for misaligned *indirect* parameters to **DrawArraysIndirect** and **DrawElementsIndirect** in section 10.4, and **DispatchComputeIndirect** in section 19 (Bug 10385).
- Use GL types instead of type name tokens in caption for table 10.9 (Bug 10636).
- Clarify in sections 11.1.3.5 and 15.2.1 that swizzling during texture lookups of textures with depth component data is always performed, whether depth comparision is disabled or enabled (Bug 10702, Public Bug 749).

- Restrict description in section 11.1.3.12 of checks required to be performed by **ValidateProgram** to only the errors described in that section (Bug 10650).
- Remove a sentence fragment accidentally left in the description of **ValidateProgramPipeline** in section 11.1.3.12.
- Specify in section 14.3.1 that the value of `SAMPLE_BUFFERS` is framebuffer-dependent, like `SAMPLES` (Bug 10688).
- Remove redundant language describing depth texture lookups in section 15.2.1 and replace with a link to section 11.1.3.5, which also has some additional bugfixes that were not present here (Bug 10997).
- Move a paragraph in section 15.2.2 to near the end of the section for better flow (Bug 10687).
- Restrict language disallowing writing to multiple "classes" of fragment shader outputs in section 15.2.3 to compatibility profile only (Bug 10126).
- Restore `TEXTURE_RECTANGLE` target to section 16.2 and table 23.17 (Bug 10671).
- Improve language describing undefined behavior when different color values are written to the same multiply-attached color buffer, and move it from section 17.3.9 to section 17.4.1 (Bug 10983).
- Specify in section 18.3.2 that linearization of sRGB formats during reads performed by blending and blitting operations is controlled by the `FRAMEBUFFER_SRGB` enable (Bug 8560).
- Change the type of `FRAMEBUFFER_ATTACHMENT_LAYERED` in table 23.31 from $n \times B$ to B.
- Restore missing `TEXTURE_BUFFER_BINDING` query in table 23.87 for the buffer object bound to the corresponding bind point (Bug 4353, public Bug 844).

Changes in the released Specification of July 22, 2013:

- Added new features as described in section H.1.
- Changed references throughout the spec to the value of `SAMPLE_BUFFERS` being "greater than zero" to "one", since it can only take on values of zero and one.

- Added introductory subsection 1.4 describing how to file bug reports against the GL and GLSL Specifications in the Khronos public Bugzilla (public bug 379).
- Added a note that querying `QUERY_RESULT_AVAILABLE` for a query object will eventually succed in section 4.2.3 (Bug 9766).
- Add views of an object's data store to the list of conditions in sections 5 and 5.1.3 under which an object is considered in use for purposes of determining object lifetimes (Bug 10511).
- Minor clarifications in sections 5.3.1, 8.21, 7.6.3, 7.7.2, 7.8, 13.1, and 15.2.1 (Bug 10346).
- Added `INVALID_VALUE` error for **BindBufferRange** in section 6.1.1 if *offset* is negative (Bug 9873).
- Add **BufferSubData** error for immutable storage without dynamic draw flag in section 6.2 (Bug 10326).
- Modified prototypes for vector forms of the commands **GetConvolutionParameter** (section 8.4.3.8), **GetHistogramParameter** (section 8.4.3.13), **GetMinmaxParameter** (section 8.4.3.16), **MultiTexCoord*v** (section 10.2.2), **GetTexParameter*** and **GetTexLevelParameter*** (section 8.11.1), and **VertexAttrib4Nub** (section 10.2.1), so that they are passed pointers rather than scalars as suggested by the 'T' notation (public bug 273).
- Clarify that no specific compressed one-dimensional texture formats are supported by the API, but may be by extensions, in section 8.7 (Bug 10388).
- Restore errors for setting invalid rectangle texture parameters in section 8.10 (Bug 10208).
- Move language constraining $level_{base}$ and $level_{max}$ for immutable textures from section 8.10 to section 8.14.3 (Bug 9342).
- Added alternate mipmap level selection computation in equation 8.15 (Bug 10119).
- Add forward references to pixel packing and `PACK_*` pixel storage modes from descriptions of texture queries and **PixelStore** in sections 8.11, 8.4.1, and 18.4 (Bug 10380).

- Describe wrap mode application for MIRROR_CLAMP_TO_EDGE consistently with other modes in table 8.28, and use mod math operator instead of $fmod()$ function since the inputs are described in non-scaled integer coordinates (Bug 10432).
- Use actual VIEW_CLASS_* compatibility class names in table 8.30 instead of abstract class names, and refer to these tokens from the description of VIEW_COMPATIBILITY_CLASS in section 22.3.2 instead of enumerating them again (Bug 10518).
- Change completeness condition for stencil index textures and stencil texturing from depth+stencil textures to match other integer textures in section 8.17 (Bug 10372).
- Add number of samples to proxy texture state in section 8.22 and specify that its value is not checked for multisample textures (Bug 10171).
- Add error for querying component type of a combined depth+stencil buffer with **GetFramebufferAttachmentParameteriv** in section 9.2.3, and note that the color encoding for non-color buffers is returned as LINEAR (Bug 9170).
- Add conditions in section 9.4.1 for texture image attachments, making framebuffer attachment completeness dependent on valid mipmap level and mipmap completness of the image and including the case where texture image attachment is part of a cubemap texture, which must be mipmap cube complete (Bug 9689).
- Remove redundant paragraph from section 10.2 (Bug 10311).
- Limit *stride* parameters to **BindVertexBuffer*** and the generic and fixed-function vertex array specification commands in section 10.3 to the value of the new implementation-dependent state MAX_VERTEX_ATTRIB_STRIDE in table 23.67 (Bug 10229).
- Move non-local error descriptions for ***Indirect*** rendering and dispatch commands to live with the commands themselves in sections 10.3.11, 10.4, and 19 (Bug 10385).
- Only allow generated *buffer* object names in the core profile for **BindVertexBuffer** in section 10.3.2 (Bug 10486).
- Added **BlitFramebuffer** to commands affected by conditional rendering in section 10.9 (Bug 9562).

- Restore error for **BindAttribLocation** attribute variable names starting with the reserved "gl_" prefix to core profile in section 11.1.1 (Bug 10203).
- Clarify that **GetTransformFeedbackVarying** may be used to query any transform feedback varying variable in section 11.1.2.1 (Bug 10472).
- Add a non-local error in section 13.3.2 for GL commands that attempt to read or write to an active and unpaused transform feedback buffer (Bug 10193).
- Changed type of **GetPolygonStipple** argument to ubyte in section 14.6.2 to match shipping header files and be typesafe (Bug 10110).
- Clean up description of **GetFragData*** in section 15.2.3 (Bug 10127).
- Made default value for ALPHA_TEST_REF floating-point in section 17.3.2, and changed query in table 23.26 to **GetFloatv** (Bug 10128).
- Strike redundant language describing default/FBO bindings under **DrawBuffer** (see section 17.4.1) and **ReadBuffer** (section 18.2.1). Bring language for the two sections in sync, and restore errors for **ReadBuffer** corresponding to **DrawBuffer** (Bug 10172).
- Fix error specified for **ColorMaski** in section 17.4.2 to specify the correct formal parameter name, *buf*, and be restricted to that command (public bug 256).
- Minimize use of generic **ClearBuffer*** terminology, since **ClearBuffer*Data** behave differently than the similarly named commands in section 17.4.3 with respect to conditional rendering and rasterizer discard (Bug 10312).
- Clarify that **ClearBuffer*** can clear any valid draw buffers in section 17.4.3.1, and that the commands have no effect when the selected draw buffer has the value NONE (Bug 10537).
- Relax **BlitFramebuffer** in section 18.3.2 so that format conversion can take place during multisample blits, since drivers already allow this and some apps depend on it. Simplified references to SAMPLE_BUFFERS by using the term "multisampled". Cleaned up the Errors section and moved errors from spec body language into it (Bugs 9692, 10219).
- Add const qualifier to *userParam* argument of **DebugMessageCallback** and DEBUGPROC in section 20.2. Specify that unrecognized message IDs in **DebugMessageControl** *ids* array are ignored in section 20.4. Specify

in section 20.9 that **GetDebugMessageLog** *messageLog* parameter must be NULL when *bufSize* is less than zero, to allow an early out. (Bug 10083).

- Reorganize section 22.3 to group internal format query *pname*s by the type of query and sort them, and add CLEAR_TEXTURE *pname* to match extension spec.
- Add missing PRIMITIVE_RESTART_FIXED_INDEX to table 23.8 (Bug 10250).
- Clean up state table entries to indicate enumerated types in tables 23.8, 23.11, 23.19, 23.23, 23.24, 23.31, and 23.33 (Bug 10251).
- Fixed types of CLIP_DISTANCEi and DEPTH_RANGE in table 23.10 (Bugs 10106, 10107).
- Add missing UNIFORM_BLOCK_NAME_LENGTH state to table 23.47 (Bug 8136).
- Correct section references from table 23.65 for MAX_ATTRIB_STACK_DEPTH and MAX_CLIENT_ATTRIB_STACK_DEPTH (Bug 10572).
- Change minimum number of compressed texture formats to 18 in table 23.67, comprising the required specific formats in table 8.22 (Bug 7235).

H.4 Credits

OpenGL 4.4 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 4.4, including the company that they represented at the time of their contributions, follow.

Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 4.4. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 4.4 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Registry.

Alon Or-bach, Samsung
Aras Pranckevicius, Unity3D
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benj Lipchak, Apple

Benji Bowman, Imagination Technologies
Bill Licea-Kane, AMD
Brent Insko, Intel
Bruce Merry, Independent
Christoph Kubisch, NVIDIA
Christophe Riccio, AMD
Daniel Koch, NVIDIA (`GL_ARB_clear_texture`, `GL_ARB_texture_mirror_clamp_to_edge`, `GL_ARB_vertex_type_10f_11f_11f_rev`)
Daniel Rakos, AMD (`GL_ARB_query_buffer_object`)
Graham Connor, Imagination Technologies
Graham Sellers, AMD (`GL_ARB_buffer_storage`)
Ian Romanick, Intel
James Helferty, NVIDIA
James Jones, NVIDIA
Jeff Bolz, NVIDIA (`GL_ARB_texture_stencil8`)
John Kessenich, Independent (OpenGL Shading Language Specification Editor, `GL_ARB_enhanced_layouts`)
Jon Leech, Independent (OpenGL API Specification Editor)
Jonathan Putsman, Imagination Technologies
Karol Gasinski, Intel
Larry Seiler, Intel
Lingjun (Frank) Chen, Qualcomm
Mark Callow, Artspark
Nick Penwarden, Epic Games
Pat Brown, NVIDIA (`GL_ARB_multi_bind`)
Pierre Boudier, AMD
Piers Daniell, NVIDIA
Rob Barris, NVIDIA
Robert Simpson, Qualcomm
Slawomir Grajewski, Intel
Tim Foley, Intel
Tom Olson, ARM

H.5 Acknowledgements

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

The “pipeline metro” cover image was created by Dominic Agoro-Ombaka of Gold Standard Group.

Appendix I

Version 4.5

OpenGL version 4.5, released on August 11, 2014, is the seventeenth revision since the original version 1.0.

Separate versions of the OpenGL 4.5 Specification exist for the *core profile* and *compatibility profile* described in appendix E, respectively subtitled the "Core Profile" and the "Compatibility Profile". This document describes the Compatibility Profile. An OpenGL 4.5 implementation *must* be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 4.5 compatibility and core profiles are upward compatible with the OpenGL 4.4 compatibility and core profiles, respectively (see appendix H).

Following are brief descriptions of changes and additions to OpenGL 4.5. Descriptions of changes and additions in versions of OpenGL prior to 4.2 are omitted in this Specification, but may be found in the OpenGL 3.0 Specification (for features in versions 1.0 - 3.0, inclusive) and the OpenGL 4.2 Specification (for features in versions 3.1 - 4.1, inclusive). These Specifications are available in the OpenGL Registry.

I.1 New Features

New features in OpenGL 4.5, including the extension or extensions if any on which they were based, include:

- `GL_ARB_clip_control`
- `GL_ARB_cull_distance`

- GL_ARB_ES3_1_compatibility
- GL_ARB_conditional_render_inverted
- GL_KHR_context_flush_control
- GL_ARB_derivative_control (OpenGL Shading Language Only)
- GL_ARB_direct_state_access
- GL_ARB_get_texture_sub_image
- GL_KHR_robustness
- GL_ARB_shader_texture_image_samples (OpenGL Shading Language Only)
- GL_ARB_texture_barrier

I.2 Deprecation Model

No new features are deprecated, and no previously deprecated features are reintroduced by the OpenGL 4.5 core profile.

Features deprecated by OpenGL 4.4 remain deprecated, but have not yet been removed.

I.3 Change Log for Released Specifications

Changes in the released Specification update of June 29, 2017

- Updated document copyright to new standard Khronos specification copyright.
- Specify in section 2.3.1 that detection of errors based on the value of parameters is always performed prior to any state-based validation (Bug 11046).
- Clarify that program object state queries in section 7.14 return the state presently in effect, which may be different than most recently set state (Bug 9702).
- Clarify in section 7.14 that there is always an info log for shader, program, and program pipeline objects, even if it is empty; and that the strings returned by **GetShaderInfoLog**, **GetProgramInfoLog** and **GetProgramPipelineInfoLog** are always null-terminated (Bug 16110).

- Note that the implementation-dependent maximum texture dimensions are independent in each image dimension, and that implementations may fail to create textures of the maximum sizes due to resource exhaustion, in section 8.5.3 (Bug 12249).
- Adjust the level-of-detail computation of $f(x, y)$ in section 8.14.1 to consider other vectors when computing scale factors (Bug 16179).
- Clean up level-of-detail computation in equation 8.15 section 8.14.3 to relax computation to match the Vulkan specification (Bug 16233).
- Specify in section 8.14.4 that **GenerateMipmap** has no effect when the base mipmap level is undefined (Bug 11147).
- Add a description in section 9.2.3 of the case when querying **Get*FramebufferAttachmentParameteriv** for `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` returns `NONE`, and the default framebuffer attachment is not allocated (Bug 16189).
- Add an error to **FramebufferTexture*D** and ***FramebufferTextureLayer** in section 9.2.8 when attempting to attach a non-existent level of an immutable texture, and make the language in these sections closer to OpenGL ES (Bugs 15946,16214).
- Specify that generic vertex attribute zero has current values in the description of **VertexAttrib*** in section 10.2.1, and remove the error in section 10.5 for **GetVertexAttrib*** queries of `CURRENT_VERTEX_ATTRIB` *index* zero, for increased interoperability with core profile and OpenGL ES (Bug 9606).
- Added a missing Errors section for **TexEnv*** in section 16.1 of the compatibility profile, limited fixed-function texture environment state to the value of **MAX_TEXTURE_IMAGE_UNITS** by generating an error in **TexEnv*** as well as the fixed-function texture enables described in section 16.2, and fixed the back-reference from **GetTexEnv*** in section 16.1.1 to the **TexEnv*** Errors section to account for the inconsistent name of the *target* / *env* parameter (Bug 10336).
- Add clarification that one of the error conditions for ***DrawBuffers** in section 17.4.1 applies only to the default framebuffer (Bug 16153).
- Change the initial `QUERY_RESULT_AVAILABLE` state for query objects to `TRUE` in table 23.55, consistent with other changes made in Public Bug 1213 (Bug 16089).

- Update the extension summaries in appendix K.3.3 to reflect all currently published ARB extensions.

Changes in the released Specification update of October 24, 2016

- Add language to **GetProgramResourceiv** in section 7.3.1.1 specifying the returned block index when the interface block is declared as an array of block instances, and clarify the definition of active uniform blocks in section 7.6 (Bug 11938).
- Remove required clamping of texture border colors in section 8.14.2 and make out-of-range values undefined, since not all hardware supports this (Bug 14442).
- Add missing error for **VertexAttrib*Pointer** in section 10.3.2 of the core profile when no vertex array object is bound (Bug 10694).
- Document in section 10.7.4 that double-precision vertex attributes transferred with **ArrayElement** use either **VertexAttrib** or **VertexAttribL**, depending on how the array pointer was specified (Bug 6516).
- Add validation errors in section 11.1.3.12 for multiple program objects exceeding any of the `COMBINED` resource limits, not just shader storage blocks (Bug 8834).
- Attempt to rotate landscape-format state table pages when being displayed - this is dependent on the PDF reader (Bug 11976).

Changes in the released Specification update of July 7, 2016

- Specify that queries returning unsigned integers will clamp negative state values in section 2.2.1 (Bug 14444).
- Clean up error language around reuse of query objects in section 4.2 to make clear that occlusion query objects created with *target*s `ANY_SAMPLES_PASSED` or `ANY_SAMPLES_PASSED_CONSERVATIVE` may be specified and reused with either of those query targets (Bug 13342).
- Use consistent phrasing of "has/has been linked successfully" to describe the link status of programs where relevant in the descriptions of conditions for **UseProgram** in section 7.3, **GetProgramResourceLocation*** in section 7.3.1.1, **UseProgramStages** and **ActiveShaderProgram** in section 7.4, **GetProgramBinary** in section 7.5, **ProgramUniform*** in section 7.6.1,

GetActiveAttrib in section 11.1.1, and **GetFragDataLocation** and **GetFragDataIndex** in section 15.2.3.

This replaces a variety of previous usages including the rather wordy "has not been linked, or was linked unsuccessfully" (Bug 8640).

- Clarify shader interface matching rules in section 7.4.1 so that there is no match in the case where name, type and qualification match, but one variable has a `location` qualifier and the other does not (Bug 13613).
- Fixed name of formal parameter of **SamplerParameter*v** in section 8.2 to *params* (Bug 14158).
- Relax error that prevented **CompressedTex*SubImage3D** from accepting `TEXTURE_CUBE_MAP` and `TEXTURE_CUBE_MAP_ARRAY` in section 8.7 (Bug 14366).
- Simplify error language for **TexParameter*** and **TextureParameter*** in section 8.10 (Bug 15457).
- Fix name of formal parameter *target* to **GetnTexImage** in section 8.11.4.
- Specify that the `COLOR_ATTACHMENT`m tokens are defined for m in the range $[0, 31]$ in section 9.2, and specify that `INVALID_OPERATION` errors are generated for valid `COLOR_ATTACHMENT`m tokens where m is outside the range of valid color attachments for the commands **Get*FramebufferAttachmentParameteriv** in section 9.2.3, ***FramebufferRenderbuffer** in section 9.2.7, ***FramebufferTexture*** in section 9.2.8, ***DrawBuffers** in section 17.4.1, and **InvalidateSubFramebuffer** and **InvalidateNamedFramebufferSubData** in section 17.4.4 (Bug 13858).
- Add missing validation error for the *target* argument of **BindRenderbuffer** in section 9.2.4 (Bug 14283).
- Add missing constraint for two-dimensional multisample array textures on the *level* argument of ***FramebufferTextureLayer** in section 9.2.8 (Bug 14189).
- Rewrite *layer* validation errors for ***FramebufferTextureLayer** in section 9.2.8 to break out validation by texture dimensionality and specify the correct constraint for cube map array textures, matching the OpenGL ES specification.
- Remove description of `component layout` qualifiers affecting values assigned to attribute variables of matrix type in section 11.1.1 (Bug 15326).

- Remove description of OpenGL Shading Language as not supporting multi-dimensional arrays from sections 11.2.1.2 and 11.2.3.3 (Bug 13824).
- Specify in section 11.3.1 that the *mode* parameter used for validating geometry shaders depends on whether or not a tessellation evaluation shader is active (Bug 14141).
- Remove clause from section 13.3.2 specifying an error for **ResumeTransformFeedback** if the active program object has been relinked since transform feedback became active for the current transform feedback object (Bug 15414).
- Cleanup the description of fragment coordinate w_f in section 15.2.2 (Bug 5434).
- Update the description of `gl_SampleMaskIn` in section 15.2.2 and `gl_SampleMask` in section 15.2.3 to refer to the maximum number of samples supported for any renderable internal format, and make corresponding changes to the definitions of the `MAX_*SAMPLES` queries in table 23.81 (Bug 15792, based on ES bug 15122).
- Fix work group size limit error for **DispatchCompute** in section 19 (Bug 15069).
- Cleanup introduction to section 23, fix type of `SYNC_FLAGS` in table 23.61, and add `CONTEXT_PROFILE_MASK` in table 23.68 (Bug 5581).
- Rewrite appendix D to refer to the Khronos Data Format Specification for the definition of formats (Bug 15255).

Changes in the released Specification update of May 28, 2015

- Fix minor typos found during OpenGL ES spec updates.
- Add language to section 4.1.2 tightening the behavior of `SYNC_FLUSH_COMMANDS_BIT` to only guarantee that commands are flushed up to and including the **FenceSync** command (Bug 11525).
- Change description of `MAP_COHERENT_BIT` for buffer storage in section 6.2 so that barriers with `CLIENT_MAPPED_BUFFER_BARRIER_BIT` does not need to make CPU writes visible to the GPU in this case without an explicit flush (Bug 13578).

- Add language for arrays of arrays to the generation of active resource lists in section 7.3.1 (Bug 13004).
- Add language in sections 7.3.1, 7.3.1.1, and 11.1.2.1 allowing program interface queries to distinguish between entire arrays and array elements for transform feedback resources (Bug 12787).
- Specify that `shared` and `std140` layout uniform blocks and their members are always active in section 7.6 (Bug 10182).
- Clarify the interaction of multisample write masks, shader side effects, and early per-fragment tests. Update the description of rasterization at the start of chapter 14 and in figure 14.1, and add language in sections 7.13.1 and 14.9. Move pixel ownership, scissor, and multisample fragment operations from section 17.3 into sections 14.9.1 and 14.9.2, note that stencil test, depth test, and occlusion query operations may be performed prior to fragment shading if requeseted by the shader, and update figure 17.1 to match. Move description of the application of `gl_SampleMask` from section 14.9.3 to section 15.2.3. Split alpha to coverage from multisample fragment operations, leaving the former in section 17.3.1. Update section references in table 23.16 (Bug 12936).
- Strike `INVALID_ENUM` error for invalid packed depth-stencil *format*/*type* combinations in section 8.4.4; it is covered by the generic special interpretation language in section 8.4.4.2. Also change the error for invalid integer-float *format*/*type* mismatches to `INVALID_OPERATION` in section 8.4.4 (Bug 11167).
- Add missing error conditions in section 8.5 when specifying texture images with specific compressed internal formats and there's a texture dimensionality and/or *target* mismatch with the format (Bug 11239).
- Added a definition of the effective internal format corresponding to base formats in section 8.5 (without going into detail on how it is determined, unlike the OpenGL ES Specification), and modified the descriptions of format matching for **TextureView** in section 8.18 and **CopyImageSubData** in section 18.3.3 to use the effective internal format where appropriate (Bug 13111).
- Fix error for invalid *target* to **GetTexSubImage*** in section 8.6 to generate `INVALID_ENUM` instead of `INVALID_VALUE` (Bug 13563).

- Add errors to ***Tex*SubImage3D** (section 8.6 and to **GetTexture*Image** and **GetCompressedTextureImage** (section 8.11.4) in the cases where the *target* or effective target of the *texture* is `TEXTURE_CUBE_MAP` or `TEXTURE_CUBE_MAP_ARRAY`, and the texture is not cube complete or cube array complete, respectively (Bug 13223).

- Define initial state for sampler objects min filter and wrap modes in table 23.23, instead of deferring it to inapplicable language describing these parameters for texture objects as dependent on the texture type. Add a footnote noting this for **CreateSamplers** in section 8.2 (Bug 13499).

- Add missing language from `ARB_gpu_program5` describing *comp* argument to `textureGather*`, and `textureGatherOffsets` commands in section 8.14.2. This language should have been included in the OpenGL 4.0 Specification, but was left out by mistake (Bug 5910).

- Specify that the values returned for RGBA texture sampling and fetches from incomplete textures are in floating-point format, in sections 8.14.2 and 11.1.3.5 (Bug 13525).

- Drop redundant bullet point in the list of texture incompleteness conditions at the end of section 8.17 (Bug 12791).

- Fix typo for `DEPTH_STENCIL_TEXTURE_MODE` (was improperly shown as `DEPTH_TEXTURE_STENCIL_MODE` in section 8.22) (Bug 13590).

- Restore the initial value of internal format for buffer textures in the compatibility profile to `LUMINANCE8` in section 8.22 and table 23.21, fixing an error introduced in earlier GL specs (Bug 10185).

- Restore cube map arrays, which were accidentally removed, to the list of framebuffer attachment completeness conditions in section 9.4.1 (Bug 11201).

- Rewrite the beginning of section 11.1.3.2 to clarify which operations are and are not performed during `texelFetch` operations (Bug 13833).

- Allow errors resulting from user-defined tessellation control inputs and outputs, and tessellation evaluation inputs that are not specified to be arrays exactly `gl_MaxPatchVertices` long to be detected at compile time, as well as link time in sections 11.2.1.2, 11.2.1.2.3 and 11.2.3.3 (Bug 12185).

- Clarify in section 14.3.1 that querying SAMPLE_POSITION returns a shading sample location, not a rasterization or coverage sample location (Bug 13484).
- Align language with GL_OES_sample_shading in section 14.3.1.1.
- Rearrange description of gl_SampleMaskIn in section 15.2.2 following variables whose use can kick off per-sample shading, and mention that (by symmetry with GL_OES_sample_variables).
- Add explicit errors list for **GetPointerv** and **GetString** in section 22.2.
- Fix description of MAX_TESS_CONTROL_TOTAL_OUTPUT_COMPONENTS in table 23.70 to reflect that it is the maximum for all outputs, not per-patch outputs (Bug 13765).

Changes in the released Specification update of February 2, 2015

- Remove reference to nonexistent ARRAY_SIZE language in the discussion of array trimming for active variables in section 7.3.1 (Bug 13445).
- Remove unimplementable TEXTURE_BINDING query from description of **BindTexture** in section 8.1, section 22.5, and table 23.18 (Bug 13278). Also removed in GL_ARB_direct_state_access specification and the specfile and headers in the Registry.

Changes in the released Specification update of October 30, 2014

- Restore language describing non-sequentiality of resource locations for consecutive active array elements in section 7.3.1 (Bug 12318).
- Restore description of MAX_UNIFORM_BLOCK_SIZE in section 7.6.2, which was lost in the restructuring and program interface query language introduced in OpenGL 4.3, but change behavior so that exceeding the limit **will** cause link failure, compared to **may** cause link failure in the 4.2 language (Bug 12897).
- Add NEAREST_MIPMAP_NEAREST to the allowed filter modes for STENCIL_INDEX textures in section 8.17 (Bug 12791).
- Modify behavior of primitive restart in section 10.3.6 to only apply to ***DrawElements*** commands for both forms of primitive restart, matching shipping drivers (Bug 12893).

- Rearrange descriptions of **DrawArraysOneInstance**, **DrawElementsOneInstance**, and the actual **DrawElements*** commands in section 10.4 to use the term "vertex ID" when referring to the actual element index of an element transferred to the GL, and make clear that the vertex ID does include the *basevertex* value passed to the **DrawElements*BaseVertex** commands (Bug 12756).
- Add description of conditions for which multisample texel fetch operations are undefined in section 11.1.3.3 (Bug 12255).
- Modify the end of section 13.7.1 to remove language requiring that integer outputs be qualified as `flat`, since fragment shader inputs now control as described in the OpenGL Shading Language Specification (Bug 12623).
- In the description of the source of the derived point size when program point size mode is enabled in section 14.4, remove the unreachable case where a tessellation evaluation shader is active, but no tessellation control shader or geometry shader is active (Bug 12865).
- Add missing error for invalid *target* argument to **InvalidateSubFramebuffer** in section 17.4.4 (Bug 12727).
- Restore description of `TEXTURE_BUFFER_OFFSET_ALIGNMENT` in table 23.67, `UNIFORM_BUFFER_OFFSET_ALIGNMENT` in table 23.75, and `SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT` in table 23.76 to refer to them as minimum require alignments, while the footnote in the caption continues to note that the numeric limits are the maximum allowed values (Bug 11962).
- Increase the minimum values for `MAX_GEOMETRY_UNIFORM_COMPONENTS` and `MAX_COMPUTE_UNIFORM_COMPONENTS` from 512 to 1024 in tables 23.72 and 23.74, respectively. This makes the minimums consistent across all shader stages (Bug 12731).

Changes in the released Specification update of September 19, 2014

- Clarify description of the initial state of query objects for **CreateQueries** and **BeginQueryIndexed** in section 4.2 (Public Bug 1213).
- Add DSA-style **GetQueryBufferObject*** queries along with **GetQueryObject*** in section 4.2.3 (Public Bug 1214).
- Add missing **MapNamedBufferRange** to description of `MAP_COHERENT_BIT` in section 6.2 (Public Bug 1208).

- Modify description of active resource list enumeration in section 7.3.1 to treat only arrays of aggregate types as top-level arrays, and clarify how this applies to **GetProgramResourceiv** queries `TOP_LEVEL_ARRAY_SIZE` and `TOP_LEVEL_ARRAY_STRIDE` (Bug 11753).
- Clarify behavior of rendering to multiple framebuffer object attachments of different sizes in section 9.2 (Bug 10403).
- Add `FRONT` and `BACK` as valid *attachment* names for **Get*FramebufferAttachmentParameteriv** in section 9.2.3 (Bug 12695).

Changes in the released Specification of August 11, 2014:

- Added new features as described in section I.1.
- Cleanup many bugs and typos (public Bug 1186).
- Clean up reference to texture borders in the core profile, where borders must always be zero width, simplifying equations (Bug 10507).
- Replace all uses of "texel array" and "image array" with "texture image" or "image" as appropriate, to avoid confusion with array texture types.
- Use "max.", "min.", and "no." consistently throughout state tables instead of "maximum", "minimum", and "number".
- Make description of WebGL in section 1.3.5 consistent with the OpenGL ES 3.1 Specification.
- Change required size of `boolean` from "1 or more" bits to exactly 8 bits in table 2.2, matching OpenGL ES 3.1 (Bug 11847).
- Clarify in equations 2.3 and 2.4 that either truncation or rounding are acceptable when converting from floating-point to normalized fixed-point (Bug 9976).
- Simplify and generalize language about object and name lifetimes in section 5.1.3 to make clear that an active query object behaves like an object attachment (Bug 12161).
- Add missing validation error for invalid *target* parameters to **MapBufferRange** in section 6.3.
- Simplify language in section 6.3.2 to remove confusing reference to "invalid reads, writes, or state changes" to mapped buffer objects (Bug 12300).

- Modify error condition for invalid buffer binding target arguments to **CopyBufferSubData** in section 6.6 to `INVALID_ENUM` instead of `INVALID_VALUE`.
- Make **LinkProgram** fail with an empty program object (one with no shaders attached) in the core profile only, in section 7.3 (Bug 12215).
- Fix error condition for **UseProgram** in section 7.3 (Bug 12281).
- Fix description of `LINK_STATUS` for **GetProgramiv** in section 7.14 (Bug 9698).
- Change the variable representing compressed texture block size to $blocksize$ in section 8.7, to avoid confusion with the texture border width b_s used in many other places.
- Clarify that filter state is ignored for multisample texture access in sections 8.8 and 11.1.3.3 (Bug 12171).
- Include `MIRROR_CLAMP_TO_EDGE` in the wrap modes not allowed for rectangle texture targets in the description of **TexParameter*** in section 8.10 (public Bug 1186).
- Make the minification vs. magnification switch-over point always zero in sections 8.14.3 and 8.15 (Bug 9997).
- Strike irrelevant clause from description of error condition for *numlayers* argument to **TextureView** in section 8.18 (Bug 11891).
- Add an error for **TextureView** in section 8.18 if the computed number of levels or layers for the new texture view is not positive (Bug 12256).
- Update errors for **TexStorage*DMultisample** in section 8.19 to include an appropriate subset of the generic errors for **Tex*Storage*D** commands defined in section 8.19, and remove redundant errors from the latter commands (Bug 11937).
- Allow **GetFramebufferParameteriv** and **GetNamedFramebufferParameteriv** in section 9.2.3 to be used to query the framebuffer-dependent state in table 23.85, including queries for the default framebuffers (Bug 12360).
- Add missing error for invalid *attachment* arguments of **GetFramebufferAttachmentParameteriv** and **GetNamedFramebufferAttachmentParameteriv** in section 9.2.3.

- Fix error condition for **GetFramebufferAttachmentParameteriv** in section 9.2.3 (Bug 12180).
- Moved description of `SAMPLE_BUFFERS` and `SAMPLE_BUFFERS` from section 9.4.2 to new section 9.2.3.1, and add a comment about the *effective* value of these parameters for framebuffer objects other than the currently bound draw framebuffer. Change references to these parameters accordingly in sections 8.6, 14.3.1, 18.2.2, 18.4, and 18.3.2 (Bug 12360).
- Add missing cube map array textures to list of layered texture types for **FramebufferTexture** in section 9.2.8 (Bug 12336).
- Restore missing error condition for **CheckFramebufferStatus** *target* argument in section 9.4.2.
- Clarify in section 10.4 that **DrawArrays** transfers no elements when *count* is zero (Bug 10015).
- Specify the values of `gl_VertexID` in the descriptions of drawing pseudocommands **DrawArraysOneInstance** and **DrawElementsOneInstance** in section 10.4 (Bug 12202).
- Remove *count* from the list of parameters validated in the pseudocode for **MultiDrawElements** in section 10.4, since it's an array, not a value (Bug 7004).
- Clarify that **MultiDrawArraysIndirect** and **MultiDrawElementsIndirect** in section 10.4 share the same errors as the comparable non-**MultiDraw*** commands, and throughout the spec, clarify relevant pseudocode examples with "(assuming no errors are generated)" (Bug 12351).
- Add description of `ELEMENT_ARRAY_BUFFER_BINDING`. to section 10.5 (Bug 11042).
- Fix list of required state in section 10.6 and table 23.7 (Bug 10283).
- Clarify description of **BindAttribLocation** in section 11.1.1 (Bug 12186).
- Make validation fail in section 11.1.3.12 when an empty program pipeline object (one with no code for any shader stage) is current (Bug 12176).
- Add language for robust buffer access in section 11.1.3.13 expanding the definition of what can be returned from out-of-bounds shader reads within a buffer object (Bugs 10826, 12104).

- Modify descriptions of tessellation in sections 11.2.2, 11.2.2.1, and 11.2.2.2 to clarify that tessellation may produce multiple vertices with the same `gl_TessCoord` values under some conditions (Bug 11979).
- Fixed **DebugMessageInsert** in section 20.5 so error for invalid *severity* arguments is `INVALID_ENUM`, matching `ARB_debug_output`, instead of `INVALID_VALUE`.
- Change title of section 22.2 to more accurately describe the contents (Bug 12352).
- Change query commands for buffer storage state in table 23.9 to **GetBufferParameteriv** (Bug 12307).
- Change the description column for `TEXTURE_BUFFER_OFFSET_ALIGNMENT` in table 23.67, `UNIFORM_BUFFER_OFFSET_ALIGNMENT` in table 23.75, and `SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT` in table 23.76 to refer to them as maximum, not minimum, values, consistently with the footnote in the caption (Bug 11962).
- Clean up caption for table 23.85 to accurately describe which FBO binding is used for different queries (Bug 12360).

I.4 Credits

OpenGL 4.5 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 4.5, including the company that they represented at the time of their contributions, follow.

Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 4.5. In addition, many people participated in developing earlier vendor and `EXT` extensions on which the OpenGL 4.5 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Registry.

Acorn Pooley, NVIDIA
Alex Corscadden, VMware
Anton Staaf, Google
Aras Pranckevicius, Unity
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)

Bill Licea-Kane, Qualcomm
Brian Paul, VMware Inc. (`GL_ARB_conditional_render_inverted`, `GL_ARB_cull_distance`, `GL_ARB_get_texture_sub_image`)
Bruce Merry, ARM
Chris Marrin, Apple
Christoph Kubisch, NVIDIA
Christophe Riccio, Unity (`GL_ARB_direct_state_access`)
Daniel Koch, NVIDIA
Daniel Rakos, AMD
Dominik Witczak, Samsung Electronics (reference pages)
Evan Hart, NVIDIA
Graham Sellers, AMD (`GL_ARB_direct_state_access`)
Greg Roth, NVIDIA
Ian Romanick, Intel
Jaroslaw Lipowski (extensive proofreading in public bug 1186)
Jason Mitchell, Valve
Jeannot Breton, NVIDIA
Jeff Bolz, NVIDIA (`GL_ARB_texture_barrier`)
Jesse Hall, Google
John Kessenich, Independent (OpenGL Shading Language Specification Editor, `GL_ARB_derivative_control`)
John McDonald, NVIDIA
Jon Leech, Independent (OpenGL API Specification Editor, reference pages)
Kenneth Russell, Google
Kent Miller, Apple
Lingjun (Frank) Chen, Qualcomm
Mark Callow, ArtSpark
Mark Kilgard, NVIDIA (`GL_ARB_clip_control`, `GL_ARB_robustness`)
Mathias Schott, NVIDIA (`GL_ARB_shader_texture_image_samples`)
Nick Penwarden, Epic Games
Pat Brown, NVIDIA
Patrick Doane, Blizzard
Piers Daniell, NVIDIA (`GL_ARB_shader_texture_image_samples`, `GL_ARB_ES3_1_compatibility`)
Robert Simpson, Qualcomm
Simon Bennett, VMware
Stefan Dösinger, CodeWeavers
Tim Foley, Intel
Timo Suoranta, Broadcom
Timothy Lottes, Epic Games

Tom Olson, ARM
Tristan Lorach, NVIDIA
Vladimir Vukicevic, Mozilla

I.5 Acknowledgements

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

The “pipeline metro” cover image was created by Dominic Agoro-Ombaka of Gold Standard Group.

Appendix J

Version 4.6

OpenGL version 4.6, released on July 31, 2017, is the eighteenth revision since the original version 1.0.

Separate versions of the OpenGL 4.6 Specification exist for the *core profile* and *compatibility profile* described in appendix E, respectively subtitled the "Core Profile" and the "Compatibility Profile". This document describes the Compatibility Profile. An OpenGL 4.6 implementation *must* be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 4.6 compatibility and core profiles are upward compatible with the OpenGL 4.5 compatibility and core profiles, respectively (see appendix I).

Following are brief descriptions of changes and additions to OpenGL 4.6. Descriptions of changes and additions in versions of OpenGL prior to 4.2 are omitted in this Specification, but may be found in the OpenGL 3.0 Specification (for features in versions 1.0 - 3.0, inclusive) and the OpenGL 4.2 Specification (for features in versions 3.1 - 4.1, inclusive). These Specifications are available in the OpenGL Registry.

J.1 New Features

New features in OpenGL 4.6, including the extension or extensions if any on which they were based, include:

- `GL_ARB_indirect_parameters`
- `GL_ARB_pipeline_statistics_query`

- `GL_ARB_polygon_offset_clamp`
- `GL_KHR_no_error`
- `GL_ARB_shader_atomic_counter_ops` (OpenGL Shading Language Only)
- `GL_ARB_shader_draw_parameters`
- `GL_ARB_shader_group_vote` (OpenGL Shading Language Only)
- `GL_ARB_gl_spirv`
- `GL_ARB_spirv_extensions`
- `GL_ARB_texture_filter_anisotropic`
- `GL_ARB_transform_feedback_overflow_query`

J.2 Deprecation Model

No new features are deprecated, and no previously deprecated features are reintroduced by the OpenGL 4.6 core profile.

Features deprecated by OpenGL 4.5 remain deprecated, but have not yet been removed.

J.3 Change Log Descriptions

In the change logs below, a comment like *(github #16)* refers to a public Github issue on the https://github.com/KhronosGroup/OpenGL-API/ repository; *(Public Bug 1432)* refers to a public Bugzilla issue at https://www.khronos.org/bugzilla; *(gitlab #40)* refers to a private issue on Khronos' internal server; and *(Bug 16315)* refers to a private Bugzilla issue on Khronos' internal server.

J.4 Change Log for Released Specifications

Changes in the released Specification of May 5, 2022

- Require shader interface matching to include OpenGL ES precision qualifiers where relevant in section 7.4.1 (gitlab #157).

- Change texture image error for mismatching *format* and base internal format in section 8.5.3 so that if either is `STENCIL_INDEX`, both must be (github #86).
- Disambiguate computation of number of rows to skip when fetching successive compressed image blocks from memory in section 8.7 (github #62).
- Update definitions of texture completeness in section 8.17 and of framebuffer completeness in section 9.4.1) to account for layered rendering to a cubemap and make the definitions more internally consistent (gitlab #141).
- Remove redundant error for **TexStorage3D** in section 8.19 when *target* is `TEXTURE_CUBE_MAP_ARRAY` and *depth* is not a multiple of 6 (gitlab #135).
- Add alternate order for triangles generated by triangle strips with adjacency in table 10.2 (gitlab #150).
- Remove reference to "state for vertex coordinates or generic attribute zero" from compatibility profile language in section 10.2.4 (gitlab #123).
- Add **CreateVertexArrays** to commands allowed to create an *array* name for **BindVertexArray** in section 10.3.1 (github #68).
- Fix typo in pseudocode equivalent of **DrawElementsInstanced** in section 10.4 by removing extra 0 argument (github #69).
- Update computation of element array buffer access for **DrawElementsOneInstance** in section 10.4 to offset in units of the size of the *type* being accessed (github #85).
- Add cube maps to allowed types of *texture* names for **FramebufferTextureLayer** in section 9.2.8 (gitlab #134).
- Change the minimum value of `MAX_FRAGMENT_INTERPOLATION_OFFSET` to $+0.5 - 1ULP$ in table 23.82 (gitlab #149).
- Add SPIR-V environment validation rule for atomics in section C.4, matching a change in `ARB_gl_spirv` (gitlab #132).

Changes in the released Specification of October 22, 2019

- Change the name of formal parameter *bufSize* to *count* for **GetSynciv**, **GetActiveSubroutineUniformName**, **GetActiveSubroutineName**, **GetInternalformativ**, **GetInternalformati64v**, and **GetProgramResourceiv** (github #52 / #285).

- Remove query objects from the list of shareable container objects in section 5 (github #60).
- Specify that linking OpenGL and OpenGL ES shader objects is allowed in the description of **LinkProgram** in section 7.3 (gitlab #98).
- Fix typo in section 7.3.1.1 (github #55).
- Reorganize description of state restored by **ProgramBinary** in section 7.5 (gitlab #120).
- Specify in section 7.5 that the state of program parameter `PROGRAM_SEPARABLE` is restored when **ProgramBinary** is called successfully (gitlab #114).
- Specify that shader storage block size is restricted to the fixed size portion reported by `BUFFER_DATA_SIZE` in section 7.8 (github #36).
- Add atomic operations on atomic counter variables to the description of ways for shaders to access buffer object memory in section 7.13 (github #14).
- Clarify memory barrier requirements in section 7.13.1 (gitlab #95).
- Add a "Copyable" column to table 8.22 specifying if formats may be used with non-**Compressed** texture image commands, and add errors to the **TexImage*D** and **CopyTexImage*D** commands (and thus to the ***SubImage*** commands which have the same constraints) referring to this table (github #11).
- Clarify completeness requirements and interactions with multisampling for integer format textures, and specify that multisample textures never require mipmaps in section 8.17 (github #45).
- Add `VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`, `VERTEX_BINDING_STRIDE`, `VERTEX_BINDING_DIVISOR`, and `VERTEX_BINDING_BUFFER` to the state queriable with **GetVertexArrayIndexediv** in section 10.5 (github #42 / gitlab #108).
- Fix typo in section 11.2.2.2 (gitlab #113).
- Update the behavior of point clipping in section 13.7 to allow pop-free clipping, for consistency with OpenGL ES (github #51).

- Loosen the line interpolation requirements in section 14.5.2.2 to properly handle `gl_FragCoord` and allow diagonal interpolation of input attributes, bring the specification in line with Vulkan and shipping hardware (gitlab #116).
- Change default values of `VERTEX_BINDING_DIVISOR` and `VERTEX_BINDING_BUFFER` to zero in table 23.7 (github #44).
- Specify that the image format of `OpImageWrite` must not be unknown unless the `StorageImageWriteWithoutFormat` capability was declared, in section C.4 (gitlab #112).
- Specify that user interface variables must not contain a value of type `OpTypeBool` in section C.4 (gitlab #111).

Changes in the released Specification of February 2, 2019

- Fix minor a/an grammar issues (gitlab #101).
- Fix "builtin" → "built-in" in a few places.
- Use the term "workgroup" consistently instead of "work group" (internal issue #87 - just a typo, so not marked everywhere the change was done).
- Use explicit errors for negative `sizei` parameters everywhere for consistency, even though there's a generic error described in section 2.3.1 (gitlab #52).
- Modify description of conversion behavior for normalized floating-point values in sections 2.3.5.1 and 8.5 so that they are not clamped on upload (Bug 13802).
- Clarify enumeration of block members for arrays of uniform and shader storage buffer objects in section 7.3.1.1 (gitlab #94).
- Clarify shader interface matching in section 7.4.1 so that the built-in interface blocks between vertex shaders and downstream shaders match (Bug 16124).
- Generate an `INVALID_OPERATION` error for **GetProgramBinary** in section 7.5 if there are no defined binary formats (Bug 16155).
- Clean up definition of which uniform blocks count against combined limits in section 7.6.2 (gitlab #69).

- Make clarifications on uniform and shader storage buffer object layout requirements in section 7.6.2.3 (gitlab #92).
- Add `TEXTURE_CUBE_MAP` or `TEXTURE_CUBE_MAP_ARRAY` as valid *target*s for **CompressedTexImage3D** in section 8.7 (Bug 14713)
- Update the texture level-of-detail equation to better approximate ellipse major and minor axes for anisotropic sampling in section 8.14, and use the same terminology as the isotropic sampling case (gitlab #5).
- Fix equation for $w'(x, y)$ in section 8.14.2 to correctly clamp to d_t instead of h_t (gitlab #97). Clarify language in section 9.2.3 about valid **GetFramebufferAttachmentParameteriv** queries when the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `NONE` (Bug 16198).
- Specify in section 11.1.3.4 that `textureSize` returns undefined values when the texture is incomplete (Bug 11750).
- Add a footnote to section 11.1.3.12 clarifying that unlike OpenGL ES, OpenGL does not require validation to detect mismatched interfaces between shader stages (Bug 15331).
- Modified **BeginTransformFeedback** language about simultaneous use of buffers for transform feedback and other operations in section 13.3.2 to ensure this behavior is undefined (gitlab #83).
- Specify `std430` as a base layout for compute shader uniform storage requirements in section 19.1 (github #40).
- Add the commands **DispatchComputeIndirect**, ***DrawArraysIndirect**, and ***DrawElementsIndirect** to those generating `INVALID_OPERATION` errors when called during display list compilation, in section 21.4.1 (gitlab #84).

Changes in the released Specification of May 14, 2018

- Clarify undefined GL behavior with no current context in section 2.1 (Bug 11522).
- Add a footnote to the description of **Flush** in section 2.3.3 explaining historical reasons for its inconsistent behavior (Bug 13916).

- Note in section 2.6.6 that texture objects contain a data store. Change the feedback loop language in section 9.3 to refer to the data store rather than the texture object, and include texture views as ways to create feedback loops (Bug 11934).
- Drop now-irrelevant language in section 6.1 saying that the GL may make different choices about storage location and layout based on the initial binding (github #23).
- Clarify for **Clear*BufferSubData** in section 6.2.1 that there are no compatibility restrictions between the format and type of the specified clear color, and the internal format to be stored in the buffer (Bug 16232).
- Clarify error for **glSpecializeShader** in section 7.2.1, and add a new error if shader types mismatch (github #16).
- Add descriptions of `COMPATIBLE_SUBROUTINES` and `NUM_COMPATIBLE_SUBROUTINES` to the **GetProgramResourceiv** queries described in section 7.3.1.1 (gitlab #40).
- Specify that only a single integer is written for several resources already described as queriable with **GetProgramResourceiv** in section 7.3.1.1 (github #35).
- Clarify in section 7.4.1 that output blocks that match in all ways except their name are not guaranteed to match (Bug 15338).
- Require `Binding` for uniform and shader storage blocks in section 7.4.2 and add whitelist for accepted storage classes in section C.4 (github #5, gitlab #55)
- Specify in section 7.7.1 that the value of `UNIFORM_ARRAY_STRIDE`) for `atomic_uint` arrays is always 4 (gitlab #31).
- Added description of storage block activeness in section 7.8, based on that of uniform blocks (Bug 13745, gitlab #63).
- Add new section 7.9 defining invocation groups, as used in the OpenGL Shading Language Specification (github #15).
- Add **MultiDraw*IndirectCount** and `PARAMETER_BUFFER` to the list of commands whose buffer bindings can be affected by `COMMAND_BARRIER_BIT` in the discussion of **MemoryBarrier** in section 7.13.2 (github #17).

- Use the term "work group size" consistently in section 7.14 and chapter 19, instead of a mix of similar terms usually starting with "local" (Bug 11723).
- Remove error in section 8.6 for the ***TexSubImage3D** commands when *target* is `TEXTURE_CUBE_MAP_ARRAY`, and the target image is not cube array complete. This was inappropriately added and should only exist for the ***TextureSubImage3D** commands (Bug 13223).
- Fix typo (missing comma) in prototype for **TextureParameterI*** in section 8.10 (Public Bug 1432).
- Drop reference in section 8.14.1 to the no-longer-used minification switchover value c and replace with 0.0 (github #34).
- Specify in section 8.14.4 that mimap generation has no effect when any $level_{base}$ dimension is zero (gitlab #72).
- Add a requirement to the **Generate*Mipmap** commands in section 8.14.4 that any synchronization required before performing mipmap reduction will be done within the commands themselves (Public Bug 1429).
- Add error for **Tex*Storage3D** in section 8.19 when *target* is `TEXTURE_CUBE_MAP_ARRAY` and *depth* is not a multiple of 6 (Public Bug 1439).
- Clarify meaning of `FULL_SUPPORT` for `FRAMEBUFFER_RENDERABLE` and `FRAMEBUFFER_RENDERABLE_LAYERED` in section 9.4.3 (Public Bug 689).
- Modify the description of indirect commands in buffer objects in section 10.3.11 to say that indirect parameters *are* sourced from the indirect buffer in the core profile, and *may be* sourced from it in the compatibility profile (Public Bug 1082).
- Remove assertions that OpenGL Shading Language does not support multidimensional arrays in sections 11.2.1.2.3 and 11.3.4.4 (gitlab #62).
- Specify that interpolation of vertex outputs is controlled by fragment shader input qualifiers at the beginning of section 11.1.3.10 (github #10).
- Move definition of *upstream shader* for geometry shaders from section 11.3.4.4 to section 11.3.3 (Bug 14158).
- Use new terminology for the "last active vertex processing stage" and "when a vertex processing stage is active", rather than referring to "vertex shader outputs" and "when a vertex shader is active", respectively. This allows the

same language to apply to outputs from the shading pipeline coming from any of vertex, tessellation evaluation, or geometry shaders. These changes are made in a new section 13.2 and in sections 13.3, 13.3.2, 13.6, 13.7, 13.7.1, 14.4, 14.5.1, 14.6.1, 14.6.6, and 16.3 (github #9).

- Add a footnote to the description of **BindTransformFeedback** in section 13.3.1 explaining that the generic bind point should not be affected by this command, and move `TRANSFORM_FEEDBACK_BUFFER_BINDING` generic binding state from the transform feedback object state table 23.59 to the context transformation state table 23.10 (gitlab #66).

- Clarify near the end of section 13.3.2 that a buffer object must be bound for transform feedback only if the buffer is active (gitlab #38).

- Remove duplicate definition of "fragment" near the beginning of section 14 (Public Bug 1388).

- Ensure discussion of polygon vertex lighting in section 14.6.4 appears in the compatibility profile (github #18).

- Simplify description of per-sample shading in section 15.2.2 and clarify the requirements on the interaction between `gl_SampleMaskIn` and **MinSampleShading** (gitlab #45).

- Clarify occlusion query description in section 17.3.5 to refer to fragments that are not discarded by earlier stages, rather than fragments that pass the depth test (github #19).

- Don't require that textures passed to **CopyImageSubData** in section 18.3.3 be mipmap complete, by ignoring format-based completeness rules (Bug 16224).

- Fix error for **DispatchComputeIndirect** in section 19 to refer to `DISPATCH_INDIRECT_BUFFER` instead of `DRAW_INDIRECT_BUFFER` (gitlab #41).

- Clarify comparison of internal format queries to **GetTexLevelParameter*** in section 22.3.2, so that internal formats not supported by that command are still valid queries (gitlab #65).

- Specify
  in section C.4 that `OpImageQuerySizeLod` and `OpImageQueryLevels` only work with `Sampled` images (gitlab #73).

- Fix typo in table D.2 (Public Bug 1920).

Changes in the released Specification of July 30, 2017:

- Added new features as described in section J.1.
- Update cover image to OpenGL 4.6 (gitlab #16).
- Numerous minor typographical cleanups (gitlab #25), and use "level-of-detail" consistently, replacing occasional usage of "level of detail". These are not marked in color.
- Direct bug reports to API and GLSL Github repositories, instead of Bugzilla, in section 1.4 (gitlab #2).
- Add SPIR-V to the Related APIs in section 1.3.4, and refer to this section from the SPIR-V Execution Environment appendix in section C (gitlab #2).
- Added clarifying language to section 2.3.1 specifying when `INVALID_OPERATION` errors are generated vs. `INVALID_ENUM` (gitlab #7).
- Change reporting of `OUT_OF_MEMORY` errors in section 2.3.1.1 (gitlab #14).
- Add the pipeline statistics query targets to those for which the *index* parameter of **BeginQueryIndexed** and **EndQueryIndexed** in section 4.2.2, and of **GetQueryIndexediv** in section 4.2.3, must be zero (gitlab #26).
- Add new subsections 4.2.1 and 4.2.2, with existing content. Refer to the table of query object targets in section 4.2.1 for error purposes in **BeginQueryIndexed** and related commands in the descriptions of **CreateQueries**, **BeginQueryIndexed**, **EndQueryIndexed**, and **GetQueryIndexediv** (gitlab #18).
- Fix reference to the `BuiltIn` decoration (not variable) in section 7.4.2 (gitlab #32).
- Minor cleanup of SPIR-V interface language in section 7.4.2 and table C.6, and clarify `OpTypeRuntimeArray` restriction in section C.4 (gitlab #20).
- Clarify `RECTANGLE_TEXTURE` and compressed texture interactions in errors for ***Tex*SubImage2D** in section 8.6, **CompressedTexImage2D** in section 8.7, and **Tex*Storage2D** in section 8.19 (gitlab #10).

- Change errors for invalid effective *target*s of **TextureBufferRange** in section 8.9, **TextureParameter*** in section 8.10, **GetTextureParameter*** in section 8.11.2, **GetTextureLevelParameter*** in section 8.11.3, and **GetTextureImage** in section 8.11.4 from `INVALID_VALUE` to `INVALID_OPERATION` (gitlab #22).

- Change some errors for **GetTextureSubImage** in section 8.26 and **InvalidateTexSubImage** in section 8.20 from `INVALID_VALUE` to `INVALID_OPERATION` (gitlab #17).

- Tweak language for **TextureStorage1D** errors in section 8.19 to update generated error for the wrong effective *target* (gitlab #8).

- Change *level* limit for ***FramebufferTexture*D** and ***FramebufferTextureLayer** in section 9.2.8 from the value of `TEXTURE_IMMUTABLE_LEVELS` to that of `TEXTURE_VIEW_NUM_LEVELS`, to account for texture views (gitlab #19).

- Remove incorrectly-applied language for `gl_DrawID` that had been applied to **DrawArraysInstancedBaseInstance** in section 10.4 (gitlab #21).

- Changed the name of the first formal parameter of **GetTexEnv*** in section 16.1.1 from *env* to *target*, to match **TexEnv*** (gitlab #9).

- Add `TEXTURE_MAX_ANISOTROPY` parameter to sampler state in table 23.23, as well as texture object state (gitlab #15).

- Change required state for `QUERY_COUNTER_BITS` in table 23.81 and `CURRENT_QUERY` in table 23.87 from $18 \times E$ to $n \times E$ to avoid having to track the total number of otherwise unrelated pieces of queryable state (gitlab #11).

- Refer to the tables in section C instead of saying "above" or "below", since these tables are floats that may shift their position in the document as other material is added and removed, and clean up related language in section C.6 (gitlab #2).

J.5 Credits

OpenGL 4.6 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of

OpenGL 4.6, including the company that they represented at the time of their contributions, follow.

Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 4.6. In addition, many people participated in developing earlier vendor and `EXT` extensions on which the OpenGL 4.6 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Registry.

Brian Paul, VMware (`ARB_pipeline_statistics_query`, `ARB_transform_feedback_overflow_query`)
Christophe Riccio, OpenGL Garden / Unity
Daniel Kartch, NVIDIA
Daniel Koch, NVIDIA (`ARB_gl_spirv`, `ARB_spirv_extensions`)
Daniel Rakos, AMD (`ARB_shader_atomic_counter_ops`)
David Blythe, SGI
Dominik Witczak, AMD
Eric Lengyel, Terathon Software
Graham Sellers, AMD (`ARB_indirect_parameters`, `ARB_shader_draw_parameters`)
James Jones, NVIDIA
Jan-Harald Fredriksen, ARM
John Kessenich, Google (`ARB_gl_spirv`)
Jon Leech, Independent (OpenGL API Specification Editor)
Mark Kilgard, NVIDIA (`EXT_texture_filter_anisotropic` on which the ARB functionality was based)
Maurice Ribble, Qualcomm (`KHR_no_error`)
Pat Brown, NVIDIA (`ARB_shader_group_vote`)
Piers Daniell, NVIDIA (Chair, Khronos OpenGL Working Group; `ARB_polygon_offset_clamp`, `ARB_texture_filter_anisotropic`)
Steve Hill, Broadcom
Tobias Hector, Imagination Technologies
Walt Donovan, NVIDIA

J.6 Acknowledgements

The ARB gratefully acknowledges technical support by James Riordon, webmaster of Khronos.org and OpenGL.org.

The “pipeline metro” cover image was created by Dominic Agoro-Ombaka.

Appendix K

OpenGL Registry, Header Files, and ARB Extensions

K.1 OpenGL Registry

Many extensions to the OpenGL API have been defined by vendors, groups of vendors, and the OpenGL ARB. In order not to compromise the readability of the OpenGL Specification, such extensions are not integrated into the core language; instead, they are made available online in the *OpenGL Registry*, together with extensions to window system binding APIs, such as GLX and WGL, and with specifications for OpenGL, GLX, and related APIs.

Extensions are documented as changes to a particular version of the Specification. The Registry is available on the World Wide Web at URL

http://www.opengl.org/registry/

K.2 Header Files

Historically, C and C++ source code calling OpenGL was to `#include` a single header file, `<GL/gl.h>`. In addition to the core OpenGL API, the APIs for all extensions provided by an implementation were defined in this header.

When platforms became common where the OpenGL SDK (library and header files) were not necessarily obtained from the same source as the OpenGL driver, such as Microsoft Windows and Linux, `<GL/gl.h>` could not always be kept in sync with new core API versions and extensions supported by drivers. At this time the OpenGL ARB defined a new header, `<GL/glext.h>`, which could be obtained directly from the OpenGL Registry (see section K.1). The combination

of <GL/gl.h> and <GL/glext.h> always defines all APIs for all profiles of the latest OpenGL version, as well as for all extensions defined in the Registry.

<GL/glcorearb.h> defines APIs for the core profile of OpenGL, together with ARB extensions compatible with the core profile. It does not include APIs for features only in the compatibility profile or for other extensions.

There is currently no Khronos-supported mechanism for using vendor extensions together with <GL/glcorearb.h>, due to lack of demand and lack of knowledge on which vendor extensions are compatible with the core profile. In the future, this may be addressed by a hypothetical header <GL/glcoreext.h> which would define APIs for additional EXT and vendor extensions compatible with the core profile, but not defined in <GL/glcorearb.h>. Most older extensions are not compatible with the core profile.

Applications using the compatibility profile (see appendix E) should #include the traditional <GL/gl.h> and <GL/glext.h> headers.

Applications using the core profile, and which do not need to use vendor extensions, may instead #include the <GL/glcorearb.h> header.

By using <GL/glcorearb.h>, instead of the legacy <GL/gl.h> and <GL/glext.h>, newly developed applications are given increased protection against accidentally using a legacy feature that has been removed from the core profile, and against using a less portable EXT or vendor extension. This can assist in developing applications on a GL implementation that supports the compatibility profile when the application is also intended to run on other platforms supporting only the core profile.

Developers should always be able to download <GL/glcorearb.h> from the Registry, with this headers replacing, or being used in place of older versions that may be provided by a platform SDK.

K.3 ARB and Khronos Extensions

OpenGL extensions that have been approved by the Khronos OpenGL Architectural Review Board Working Group (ARB), or jointly approved by the ARB and the Khronos OpenGL ES Working Group (KHR), are summarized in this section. ARB and KHR extensions are not required to be supported by a conformant OpenGL implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future revision of the specification.

K.3.1 Naming Conventions

To distinguish ARB and KHR extensions from core OpenGL features and from vendor-specific extensions, the following naming conventions are used:

- A unique *name string* of the form "`GL_ARB_`*name*" or "`GL_KHR_`*name*" is associated with each extension. If the extension is supported by an implementation, this string will be present in the `EXTENSIONS` string returned by **GetString**, and will be among the `EXTENSIONS` strings returned by **GetStringi**, as described in section 22.2.
- All functions defined by the extension will have names of the form ***Function*ARB** or ***Function*KHR**, respectively.
- All enumerants defined by the extension will have names of the form *NAME*`_ARB`. or *NAME*`_KHR`, respectively.
- In addition to OpenGL extensions, there are also ARB extensions to the related GLX and WGL APIs. Such extensions have name strings prefixed by `"GLX_"` and `"WGL_"` respectively. Not all GLX and WGL ARB extensions are described here, but all such extensions are included in the registry.

K.3.2 Promoting Extensions to Core Features

Extensions can be *promoted* to required core features in later revisions of OpenGL. When this occurs, the extension specifications are merged into the core specification. Functions and enumerants that are part of such promoted extensions will have the **ARB**, **KHR**, **EXT**, or vendor affix removed.

Implementations of such later revisions should continue to export the name strings of promoted extensions in the `EXTENSIONS` string and continue to support the affixed versions of functions and enumerants as a transition aid.

For descriptions of extensions promoted to core features in OpenGL 1.3 and beyond, see the corresponding version of the OpenGL specification, or the descriptions of that version in version-specific appendices to later versions of the specification.

K.3.3 Extension Summaries

This section describes each ARB and KHR extension briefly. In most cases, the functionality of these extensions also was added to a version of the OpenGL Specification, and in these cases only the extension string is described, together with the corresponding OpenGL version.

K.3.3.1 Multitexture

The name string for multitexture is `GL_ARB_multitexture`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.2 Transpose Matrix

The name string for transpose matrix is `GL_ARB_transpose_matrix`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.3 Multisample

The name string for multisample is `GL_ARB_multisample`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.4 Texture Add Environment Mode

The name string for texture add mode is `GL_ARB_texture_env_add`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.5 Cube Map Textures

The name string for cube mapping is `GL_ARB_texture_cube_map`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.6 Compressed Textures

The name string for compressed textures is `GL_ARB_texture_compression`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.7 Texture Border Clamp

The name string for texture border clamp is `GL_ARB_texture_border_clamp`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.8 Point Parameters

The name string for point parameters is `GL_ARB_point_parameters`. It was promoted to a core features in OpenGL 1.4.

K.3.3.9 Vertex Blend

Vertex blending replaces the single model-view transformation with multiple vertex units. Each unit has its own transform matrix and an associated current weight. Vertices are transformed by all the enabled units, scaled by their respective weights, and summed to create the eye-space vertex. Normals are similarly transformed by the inverse transpose of the model-view matrices.

The name string for vertex blend is `GL_ARB_vertex_blend`.

K.3.3.10 Matrix Palette

Matrix palette extends vertex blending to include a palette of model-view matrices. Each vertex may be transformed by a different set of matrices chosen from the palette.

The name string for matrix palette is `GL_ARB_matrix_palette`.

K.3.3.11 Texture Combine Environment Mode

The name string for texture combine mode is `GL_ARB_texture_env_combine`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.12 Texture Crossbar Environment Mode

The name string for texture crossbar is `GL_ARB_texture_env_crossbar`. It was promoted to a core features in OpenGL 1.4.

K.3.3.13 Texture Dot3 Environment Mode

The name string for DOT3 is `GL_ARB_texture_env_dot3`. It was promoted to a core feature in OpenGL 1.3.

K.3.3.14 Texture Mirrored Repeat

The name string for texture mirrored repeat is `GL_ARB_texture_mirrored_repeat`. It was promoted to a core feature in OpenGL 1.4.

K.3.3.15 Depth Texture

The name string for depth texture is `GL_ARB_depth_texture`. It was promoted to a core feature in OpenGL 1.4.

K.3.3.16 Shadow

The name string for shadow is `GL_ARB_shadow`. It was promoted to a core feature in OpenGL 1.4.

K.3.3.17 Shadow Ambient

Shadow ambient extends the basic image-based shadow functionality by allowing a texture value specified by the `TEXTURE_COMPARE_FAIL_VALUE_ARB` texture parameter to be returned when the texture comparison fails. This may be used for ambient lighting of shadowed fragments and other advanced lighting effects.

The name string for shadow ambient is `GL_ARB_shadow_ambient`.

K.3.3.18 Window Raster Position

The name string for window raster position is `GL_ARB_window_pos`. It was promoted to a core feature in OpenGL 1.4.

K.3.3.19 Low-Level Vertex Programming

Application-defined *vertex programs* may be specified in a new low-level programming language, replacing the standard fixed-function vertex, transformation, lighting, and texture coordinate generation pipeline. Vertex programs enable many new effects and are an important first step towards future graphics pipelines that will be fully programmable in an unrestricted, high-level shading language.

The name string for low-level vertex programming is `GL_ARB_vertex_program`.

K.3.3.20 Low-Level Fragment Programming

Application-defined *fragment programs* may be specified in the same low-level language as `GL_ARB_vertex_program`, replacing the standard fixed-function vertex texturing, fog, and color sum operations.

The name string for low-level fragment programming is `GL_ARB_fragment_program`.

K.3.3.21 Buffer Objects

The name string for buffer objects is `GL_ARB_vertex_buffer_object`. It was promoted to a core feature in OpenGL 1.5.

K.3.3.22 Occlusion Queries

The name string for occlusion queries is `GL_ARB_occlusion_query`. It was promoted to a core feature in OpenGL 1.5.

K.3.3.23 Shader Objects

The name string for shader objects is `GL_ARB_shader_objects`. It was promoted to a core feature in OpenGL 2.0.

K.3.3.24 High-Level Vertex Programming

The name string for high-level vertex programming is `GL_ARB_vertex_shader`. It was promoted to a core feature in OpenGL 2.0.

K.3.3.25 High-Level Fragment Programming

The name string for high-level fragment programming is `GL_ARB_fragment_shader`. It was promoted to a core feature in OpenGL 2.0.

K.3.3.26 OpenGL Shading Language

The name string for the OpenGL Shading Language is `GL_ARB_shading_language_100`. The presence of this extension string indicates that programs written in version 1 of the Shading Language are accepted by OpenGL. It was promoted to a core feature in OpenGL 2.0.

K.3.3.27 Non-Power-Of-Two Textures

The name string for non-power-of-two textures is `GL_ARB_texture_non_power_of_two`. It was promoted to a core feature in OpenGL 2.0.

K.3.3.28 Point Sprites

The name string for point sprites is `GL_ARB_point_sprite`. It was promoted to a core feature in OpenGL 2.0.

K.3.3.29 Fragment Program Shadow

Fragment program shadow extends low-level fragment programs defined with `GL_ARB_fragment_program` to add shadow 1D, 2D, and 3D texture targets, and remove the interaction with `GL_ARB_shadow`.

The name string for fragment program shadow is `GL_ARB_fragment_program_shadow`.

K.3.3.30 Multiple Render Targets

The name string for multiple render targets is `GL_ARB_draw_buffers`. It was promoted to a core feature in OpenGL 2.0.

K.3.3.31 Rectangle Textures

Rectangle textures define a new texture target `TEXTURE_RECTANGLE_ARB` that supports 2D textures without requiring power-of-two dimensions. Rectangle textures are useful for storing video images that do not have power-of-two sizes (POTS). Resampling artifacts are avoided and less texture memory may be required. They are also useful for shadow maps and window-space texturing. These textures are accessed by dimension-dependent (aka non-normalized) texture coordinates.

Rectangle textures are a restricted version of non-power-of-two textures. The differences are that rectangle textures are supported only for 2D; they require a new texture target; and the new target uses non-normalized texture coordinates.

The name string for texture rectangles is `GL_ARB_texture_rectangle`. It was promoted to a core feature in OpenGL 3.1.

K.3.3.32 Floating-Point Color Buffers

Floating-point color buffers can represent values outside the normal $[0, 1]$ range of colors in the fixed-function OpenGL pipeline. This group of related extensions enables controlling clamping of vertex colors, fragment colors throughout the pipeline, and pixel data read back to client memory, and also includes WGL and GLX extensions for creating framebuffers with floating-point color components (referred to in GLX as *framebuffer configurations*, and in WGL as *pixel formats*).

The name strings for floating-point color buffers are `GL_ARB_color_buffer_float`, `GLX_ARB_fbconfig_float`, and `WGL_ARB_pixel_format_float`. `GL_ARB_color_buffer_float` was promoted to a core feature in OpenGL 3.0.

K.3.3.33 Half-Precision Floating-Point

This extension defines the representation of a 16-bit floating-point data format, and a corresponding $type$ argument which may be used to specify and read back pixel and texture images stored in this format in client memory. Half-precision floats are

smaller than full precision floats, but provide a larger dynamic range than similarly sized (`short`) data types.

The name string for half-precision floating-point is `GL_ARB_half_float_pixel`. It was promoted to a core feature in OpenGL 3.0.

K.3.3.34 Floating-Point Textures

Floating-point textures stored in both 32- and 16-bit formats may be defined using new *internalformat* arguments to commands which specify and read back texture images.

The name string for floating-point textures is `GL_ARB_texture_float`. It was promoted to a core feature in OpenGL 3.0.

K.3.3.35 Pixel Buffer Objects

The buffer object interface is expanded by adding two new binding targets for buffer objects, the pixel pack and unpack buffers. This permits buffer objects to be used to store pixel data as well as vertex array data. Pixel-drawing and -reading commands using data in pixel buffer objects may operate at greatly improved performance compared to data in client memory.

The name string for pixel buffer objects is `GL_ARB_pixel_buffer_object`. It was promoted to a core feature in OpenGL 2.1.

K.3.3.36 Floating-Point Depth Buffers

The name string for floating-point depth buffers is `GL_ARB_depth_buffer_float`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_NV_depth_buffer_float` extension, and is provided to enable this functionality in older drivers.

K.3.3.37 Instanced Rendering

The name string for instanced rendering is `GL_ARB_draw_instanced`. It was promoted to a core feature in OpenGL 3.1.

K.3.3.38 Framebuffer Objects

The name string for framebuffer objects is `GL_ARB_framebuffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based

on the earlier GL_EXT_framebuffer_object, GL_EXT_framebuffer_multisample, and GL_EXT_framebuffer_blit extensions, and is provided to enable this functionality in older drivers.

K.3.3.39 sRGB Framebuffers

The name string for sRGB framebuffers is GL_ARB_framebuffer_sRGB. It was promoted to a core feature in OpenGL 3.0.

To create sRGB format surfaces for use on display devices, an additional pixel format (config) attribute is required in the window system integration layer. The name strings for the GLX and WGL sRGB pixel format interfaces are GLX_ARB_framebuffer_sRGB and WGL_ARB_framebuffer_sRGB respectively.

K.3.3.40 Geometry Shaders

This extension defines a new shader type called a *geometry shader*. Geometry shaders are run after vertices are transformed, but prior to the remaining fixed-function vertex processing, and may generate new vertices for, or remove vertices from the primitive assembly process.

The name string for geometry shaders is GL_ARB_geometry_shader4. It was promoted to a core feature in OpenGL 3.2.

K.3.3.41 Half-Precision Vertex Data

The name string for half-precision vertex data is GL_ARB_half_float_vertex. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL_NV_half_float extension, and is provided to enable this functionality in older drivers.

K.3.3.42 Instanced Rendering

This instanced rendering interface is a less-capable form of GL_ARB_draw_instanced which can be supported on older hardware.

The name string for instanced rendering is GL_ARB_instanced_arrays. It was promoted to a core feature in OpenGL 3.3.

K.3.3.43 Flexible Buffer Mapping

The name string for flexible buffer mapping is GL_ARB_map_buffer_range. This extension is equivalent to new core functionality introduced in OpenGL 3.0,

based on the earlier GL_APPLE_flush_buffer_range extension, and is provided to enable this functionality in older drivers.

K.3.3.44 Texture Buffer Objects

The name string for texture buffer objects is GL_ARB_texture_buffer_object. It was promoted to a core feature in OpenGL 3.1.

K.3.3.45 RGTC Texture Compression Formats

The name string for RGTC texture compression formats is GL_ARB_texture_compression_rgtc. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL_EXT_texture_compression_rgtc extension, and is provided to enable this functionality in older drivers.

It was promoted to a core feature in OpenGL 3.0.

K.3.3.46 One- and Two-Component Texture Formats

The name string for one- and two-component texture formats is GL_ARB_texture_rg. It was promoted to a core feature in OpenGL 3.0. This extension is equivalent to new core functionality introduced in OpenGL 3.0, and is provided to enable this functionality in older drivers.

K.3.3.47 Vertex Array Objects

The name string for vertex array objects is GL_ARB_vertex_array_object. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL_APPLE_vertex_array_object extension, and is provided to enable this functionality in older drivers.

It was promoted to a core feature in OpenGL 3.0.

K.3.3.48 Versioned Context Creation

Starting with OpenGL 3.0, a new context creation interface is required in the window system integration layer. This interface specifies the context version required as well as other attributes of the context.

The name strings for the GLX and WGL context creation interfaces are GLX_ARB_create_context and WGL_ARB_create_context respectively.

K.3.3.49 Uniform Buffer Objects

The name string for uniform buffer objects is `GL_ARB_uniform_buffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

K.3.3.50 Restoration of features removed from OpenGL 3.0

OpenGL 3.1 removes a large number of features that were marked deprecated in OpenGL 3.0. GL implementations needing to maintain these features to support existing applications may do so, following the deprecation model, by exporting an extension string indicating those features are present. Applications written for OpenGL 3.1 should not depend on any of the features corresponding to this extension, since they will not be available on all platforms with 3.1 implementations.

The name string for restoration of features deprecated by OpenGL 3.0 is `GL_ARB_compatibility`.

The profile terminology introduced with OpenGL 3.2 eliminates the necessity for evolving this extension. Instead, interactions between features removed by OpenGL 3.1 and new features introduced in later OpenGL versions are defined by the compatibility profile corresponding to those versions.

K.3.3.51 Fast Buffer-to-Buffer Copies

The name string for fast buffer-to-buffer copies is `GL_ARB_copy_buffer`. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

K.3.3.52 Shader Texture Level-of-Detail Control

The name string for shader texture level-of-detail control is `GL_ARB_shader_texture_lod`. This extension is equivalent to new core functions introduced in OpenGL Shading Language 1.30 and is provided to enable this functionality in older versions of the shading language.

K.3.3.53 Depth Clamp Control

The name string for depth clamp control is `GL_ARB_depth_clamp`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

K.3.3.54 Base Vertex Offset Drawing Commands

The name string for base vertex offset drawing commands is `GL_ARB_draw_elements_base_vertex`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

K.3.3.55 Fragment Coordinate Convention Control

The name string for fragment coordinate convention control is `GL_ARB_fragment_coord_conventions`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

K.3.3.56 Provoking Vertex Control

The name string for provoking vertex control is `GL_ARB_provoking_vertex`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

K.3.3.57 Seamless Cube Maps

The name string for seamless cube maps is `GL_ARB_seamless_cube_map`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

K.3.3.58 Fence Sync Objects

The name string for fence sync objects is `GL_ARB_sync`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

K.3.3.59 Multisample Textures

The name string for multisample textures is `GL_ARB_texture_multisample`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

K.3.3.60 BGRA Attribute Component Ordering

The name string for BGRA attribute component ordering is `GL_ARB_vertex_array_bgra`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

K.3.3.61 Per-Buffer Blend Control

The blending interface is extended to specify blend equation and blend function on a per-draw-buffer basis.

The name string for per-buffer blend control is `GL_ARB_draw_buffers_blend`. It was promoted to a core feature in OpenGL 4.0.

K.3.3.62 Sample Shading Control

Sample shading control adds the ability to request that an implementation use a minimum number of unique sets of fragment computation inputs when multisampling a pixel.

The name string for sample shading control is `GL_ARB_sample_shading`. It was promoted to a core feature in OpenGL 4.0.

K.3.3.63 Cube Map Array Textures

A cube map array texture is a two-dimensional array texture that may contain many cube map layers. Each cube map layer is a unique cube map image set.

The name string for cube map array textures is `GL_ARB_texture_cube_map_array`. It was promoted to a core feature in OpenGL 4.0.

K.3.3.64 Texture Gather

Texture gather adds a new set of texture functions (`textureGather`) to the OpenGL Shading Language that determine the 2×2 footprint used for linear filtering in a texture lookup, and return a vector consisting of the first component from each of the four texels in the footprint.

The name string for texture gather is `GL_ARB_texture_gather`. It was promoted to a core feature in OpenGL 4.0.

K.3.3.65 Texture Level-Of-Detail Queries

Texture level-of-detail queries adds a new set of fragment shader texture functions (`textureLOD`) to the OpenGL Shading Language that return the results of au-

tomatic level-of-detail computations that would be performed if a texture lookup were to be done.

The name string for texture level-of-detail queries is `GL_ARB_texture_query_lod`.

K.3.3.66 Profiled Context Creation

Starting with OpenGL 3.2, API profiles are defined. Profiled context creation extends the versioned context creation interface to specify a profile which must be implemented by the context.

The name strings for the GLX and WGL profiled context creation interfaces are `GLX_ARB_create_context_profile` and `WGL_ARB_create_context_profile` respectively.

K.3.3.67 Shading Language Include

Shading language include adds support for `#include` directives to shaders, and a named string API for defining the text corresponding to `#include` pathnames.

The name string for shading language include is `GL_ARB_shading_language_include`.

K.3.3.68 BPTC texture compression

BPTC texture compression provides new block compressed specific texture formats which can improve quality in images with sharp edges and strong chrominance transitions, and support high dynamic range floating-point formats.

The name string for BPTC texture compression is `GL_ARB_texture_compression_bptc`.

K.3.3.69 Extended Blend Functions

The name string for extended blend functions is `GL_ARB_blend_func_extended`. This extension is equivalent to new core functionality introduced in OpenGL 3.3, and is provided to enable this functionality in older drivers.

K.3.3.70 Explicit Attribute Location

The name string for explicit attribute location is `GL_ARB_explicit_attrib_location`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

K.3.3.71 Boolean Occlusion Queries

The name string for boolean occlusion queries is `GL_ARB_occlusion_query2`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

K.3.3.72 Sampler Objects

The name string for sampler objects is `GL_ARB_sampler_objects`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

K.3.3.73 Shader Bit Encoding

The name string for shader bit encoding is `GL_ARB_shader_bit_encoding`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

K.3.3.74 RGB10A2 Integer Textures

The name string for RGB10A2 integer textures is `GL_ARB_texture_rgb10_a2ui`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

K.3.3.75 Texture Swizzle

The name string for texture swizzle is `GL_ARB_texture_swizzle`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

K.3.3.76 Timer Queries

The name string for timer queries is `GL_ARB_timer_query`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

K.3.3.77 Packed 2.10.10.10 Vertex Formats

The name string for packed 2.10.10.10 vertex formats is `GL_ARB_vertex_type_2_10_10_10_rev`. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

K.3.3.78 Draw Indirect

The name string for draw indirect is `GL_ARB_draw_indirect`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

K.3.3.79 GPU Shader5 Miscellaneous Functionality

The name string for GPU shader5 miscellaneous functionality is `GL_ARB_gpu_shader5`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

K.3.3.80 Double-Precision Floating-Point Shader Support

The name string for double-precision floating-point shader support is `GL_ARB_gpu_shader_fp64`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

K.3.3.81 Shader Subroutines

The name string for shader subroutines is `GL_ARB_shader_subroutine`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

K.3.3.82 Tessellation Shaders

The name string for tessellation shaders is `GL_ARB_tessellation_shader`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

K.3.3.83 RGB32 Texture Buffer Objects

The name string for RGB32 texture buffer objects is `GL_ARB_texture_buffer_object_rgb32`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

K.3.3.84 Transform Feedback 2

The name string for transform feedback 2 is `GL_ARB_transform_feedback2`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

K.3.3.85 Transform Feedback 3

The name string for transform feedback 3 is `GL_ARB_transform_feedback3`. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

K.3.3.86 OpenGL ES 2.0 Compatibility

The name string for OpenGL ES 2.0 compatibility is `GL_ARB_ES2_compatibility`. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

K.3.3.87 Program Binary Support

The name string for program binary support is `GL_ARB_get_program_binary`. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

K.3.3.88 Separate Shader Objects

The name string for separate shader objects is `GL_ARB_separate_shader_objects`. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

K.3.3.89 Shader Precision Restrictions

The name string for shader precision restritions is `GL_ARB_shader_precision`. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

K.3.3.90 Double Precision Vertex Shader Inputs

The name string for double precision vertex shader inputs is `GL_ARB_vertex_attrib_64bit`. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

K.3.3.91 Viewport Arrays

The name string for viewport arrays is `GL_ARB_viewport_array`. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

K.3.3.92 Robust Context Creation

Robust context creation allows creating an OpenGL context supporting robust buffer access behavior and a specified graphics reset notification behavior exposed through the `GL_ARB_robustness` extension (see section K.3.3.95).

The name strings for GLX and WGL robust context creation are `GLX_ARB_create_context_robustness` and `WGL_ARB_create_context_robustness`, respectively.

K.3.3.93 OpenCL Event Sharing

OpenCL event sharing allows creating OpenGL sync objects linked to OpenCL event objects, potentially improving efficiency of sharing images and buffers between the two APIs.

The name string for OpenCL event sharing is `GL_ARB_cl_event`.

K.3.3.94 Debug Output Notification

Debug output notification enables GL to inform the application when various events occur that may be useful during development and debugging.

The name string for debug output notification is `GL_ARB_debug_output`.

K.3.3.95 Context Robustness

Context robustness provides "safe" APIs that limit data written to application memory to a specified length, provides a mechanism to learn about graphics resets affecting the context, and defines guarantee that out-of-bounds buffer object accesses will have deterministic behavior precluding instability or termination. Some of these behaviors are controlled at context creation time via the companion `GLX_ARB_create_context_robustness` or `WGL_ARB_create_context_robustness` extensions (see section K.3.3.92).

The name string for context robustness is `GL_ARB_robustness`.

K.3.3.96 Shader Stencil Export

Sharder stencil export enables shaders to generate a stencil reference value, allowing stencil testing to be performed against per-shader-invocation values.

The name string for shader stencil export is `GL_ARB_shader_stencil_export`.

K.3.3.97 Base Instanced Rendering

The name string for base instanced rendering is `GL_ARB_base_instance`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.98 OpenGL Shading Language 4.20 Feature Pack

The name string for the OpenGL Shading Language 4.20 feature pack is `GL_ARB_shading_language_420pack`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.99 Instanced Transform Feedback

The name string for instanced transform feedback is `GL_ARB_transform_feedback_instanced`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.100 Compressed Texture Pixel Storage

The name string for compressed texture pixel storage is `GL_ARB_compressed_texture_pixel_storage`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.101 Conservative Depth

The name string for conservative depth is `GL_ARB_conservative_depth`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.102 Internal Format Query

The name string for internal format query is `GL_ARB_internalformat_query`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.103 Map Buffer Alignment

The name string for map buffer alignment is `GL_ARB_map_buffer_alignment`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.104 Shader Atomic Counters

The name string for shader atomic counters is `GL_ARB_shader_atomic_counters`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.105 Shader Image Load/Store

The name string for shader image load/store is `GL_ARB_shader_image_load_store`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.106 Shading Language Packing

The name string for shading language packing is `GL_ARB_shading_language_packing`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.107 Texture Storage

The name string for texture storage is `GL_ARB_texture_storage`. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

K.3.3.108 ASTC Texture Compression

The name string for ASTC texture compression is `GL_KHR_texture_compression_astc_ldr`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.109 Debug Contexts

This KHR extension defines debugging features and combines the functionality of `GL_ARB_debug_output`, `GL_ARB_debug_output2`, `GL_ARB_debug_group`,

and GL_ARB_debug_label. It is intended primarily to bring this debug functionality to OpenGL ES implementations.

The name string for debug contexts is GL_KHR_debug.

K.3.3.110 Shader Array of Arrays

The name string for shader array of arrays is GL_ARB_arrays_of_arrays. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.111 Clear Buffer Object

The name string for clear buffer object is GL_ARB_clear_buffer_object. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.112 Compute Shaders

The name string for compute shaders is GL_ARB_compute_shader. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.113 Copy Image

The name string for copy image is GL_ARB_copy_image. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.114 Texture Views

The name string for texture views is GL_ARB_texture_view. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.115 Vertex Attribute Binding

The name string for vertex attribute binding is GL_ARB_vertex_attrib_binding. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.116 Robustness Isolation

The name string for robustness isolation is `GL_ARB_robustness_isolation`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.117 OpenGL ES 3.0 Compatibility

The name string for OpenGL ES 3.0 compatibility is `GL_ARB_ES3_compatibility`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.118 Shader Explicit Uniform Location

The name string for shader explicit uniform location is `GL_ARB_explicit_uniform_location`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.119 Fragment Layer Viewport

The name string for fragment layer viewport is `GL_ARB_fragment_layer_viewport`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.120 Binding a Framebuffer Without Attachments

The name string for binding a framebuffer without attachments is `GL_ARB_framebuffer_no_attachments`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.121 Extended Internal Format Query

The name string for extended internal format query is `GL_ARB_internalformat_query2`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.122 Invalidate SubData

The name string for invalidate subdata is `GL_ARB_invalidate_subdata`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.123 Multi Draw Indirect

The name string for multi draw indirect is `GL_ARB_multi_draw_indirect`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.124 Program Interface Queries

The name string for program interface queries is `GL_ARB_program_interface_query`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.125 Robust Buffer Access Behavior

The name string for robust buffer access behavior is `GL_ARB_robust_buffer_access_behavior`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.126 Shader Image Size Query

The name string for shader image size query is `GL_ARB_shader_image_size`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.127 Shader Storage in Buffer Objects

The name string for shader storage in buffer objects is `GL_ARB_shader_storage_buffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.128 Stencil Texturing

The name string for stencil texturing is `GL_ARB_stencil_texturing`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.129 Texture Buffer Range

The name string for texture buffer range is `GL_ARB_texture_buffer_range`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.130 Texture Query Levels

The name string for texture query levels is `GL_ARB_texture_query_levels`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.131 Texture Storage Multisample

The name string for texture storage multisample is `GL_ARB_texture_storage_multisample`. This extension is equivalent to new core functionality introduced in OpenGL 4.3, and is provided to enable this functionality in older drivers.

K.3.3.132 Robustness Application Isolation Context Creation

These extensions allow creation of OpenGL contexts which support robustness isolation through OpenGL 4.3 or the equivalent functionality in the `GL_ARB_robustness_isolation` extension (see K.3.3.116), and may also define additional constraints around how OpenGL context reset notification affects other contexts in the share group, or other applications on the system. There are equivalent sets of extensions for both GLX and WGL window-system binding layers.

The name strings for GLX robustness application isolation context creation are `GLX_ARB_robustness_application_isolation` and `GLX_ARB_robustness_share_group_isolation`.

The name strings for WGL robustness application isolation context creation are `WGL_ARB_robustness_application_isolation` and `WGL_ARB_robustness_share_group_isolation`.

K.3.3.133 Buffer Storage

The name string for buffer storage is `GL_ARB_buffer_storage`. This extension is equivalent to new core functionality introduced in OpenGL 4.4, and is provided to enable this functionality in older drivers.

K.3.3.134 Clear Texture

The name string for clear texture is `GL_ARB_clear_texture`. This extension is equivalent to new core functionality introduced in OpenGL 4.4, and is provided to enable this functionality in older drivers.

K.3.3.135 Enhanced Layouts

The name string for enhanced layouts is `GL_ARB_enhanced_layouts`. This extension is equivalent to new core functionality introduced in OpenGL 4.4, and is provided to enable this functionality in older drivers.

K.3.3.136 Multiple Bind

The name string for multiple bind is `GL_ARB_multi_bind`. This extension is equivalent to new core functionality introduced in OpenGL 4.4, and is provided to enable this functionality in older drivers.

K.3.3.137 Query Buffer Object

The name string for query buffer object is `GL_ARB_query_buffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 4.4, and is provided to enable this functionality in older drivers.

K.3.3.138 Texture Mirror Clamp To Edge

The name string for texture mirror clamp to edge is `GL_ARB_texture_mirror_clamp_to_edge`. This extension is equivalent to new core functionality introduced in OpenGL 4.4, and is provided to enable this functionality in older drivers.

K.3.3.139 Texture Stencil8

The name string for texture stencil8 is `GL_ARB_texture_stencil8`. This extension is equivalent to new core functionality introduced in OpenGL 4.4, and is provided to enable this functionality in older drivers.

K.3.3.140 Vertex Type 10f_11f_11f

The name string for vertex type 10f_11f_11f is `GL_ARB_vertex_type_10f_11f_11f_rev`. This extension is equivalent to new core functionality introduced in OpenGL 4.4, and is provided to enable this functionality in older drivers.

K.3.3.141 Bindless Textures

Bindless textures allows access to texture objects in shaders without first binding each texture to one of a limited number of texture image units.

The name string for bindless textures is `GL_ARB_bindless_texture`.

K.3.3.142 Compute Variable Group Size

Compute variable groups size allows writing generic compute shaders that operate on workgroups with arbitrary dimensions.

The name string for compute variable group size is `GL_ARB_compute_variable_group_size`.

K.3.3.143 Indirect Parameters

Indirect parameters allows buffer objects to store drawing parameters for some drawing commands, and introduces new drawing commands that source some of their parameters from buffers.

The name string for indirect parameters is `GL_ARB_indirect_parameters`.

K.3.3.144 Seamless Cubemap per Texture

This allows providing a per-texture setting for enabling seamless sampling from cube maps.

The name string for seamless cubemap per texture is `GL_ARB_seamless_cubemap_per_texture`.

K.3.3.145 Shader Draw Parameters

This adds two new built-in variables to the OpenGL Shading Language containing values passed in the *basevertex* and *baseinstance* parameters of drawinng commands, and an index of the draw command being processed by **MultiDraw*** commands.

The name string for shader draw parameters is `GL_ARB_shader_draw_parameters`.

K.3.3.146 Shader Group Vote

Shader group vote provides new built-in functions to compute the composite of a set of boolean conditions across a group of shader invocations. These composite results may be used to execute shaders more efficiently on a single-instruction multiple-data (SIMD) processor.

The name string for shader group vote is `GL_ARB_shader_group_vote`.

K.3.3.147 Sparse Textures

This extension allows the separation of the graphics processor's address space (reservation) from the requirement that all textures must be physically backed (commitment). This exposes a limited form of virtualization for textures. Use cases include sparse (or partially resident) textures, texture paging, on-demand and delayed loading of texture assets and application controlled level-of-detail.

The name string for sparse textures is `GL_ARB_sparse_texture`.

K.3.3.148 OpenGL ES 3.1 Compatibility

This extension adds supported for features of OpenGL ES 3.1 that are missing from OpenGL 4.5.

The name string for OpenGL ES 3.1 compatibility is `GL_ARB_ES3_1_compatibility`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.149 Clip Control

This extension provides additional clip control modes to configure how clip space is mapped to window space.

The name string for clip control is `GL_ARB_clip_control`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.150 Inverted Conditional Rendering

This extension adds new modes to **BeginConditionalRender** which invert the condition used to determine whether to draw or not.

The name string for inverted conditional rendering is `GL_ARB_conditional_render_inverted`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.151 Cull Distance Shader Output

This extension adds a new OpenGL Shading Language`gl_CullDistance` shader output, similar to `gl_ClipDistance`, but used for whole primitive culling.

The name string for cull distance shader output is `GL_ARB_cull_distance`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.152 Derivative Computation Granularity

This extension provides control over the spacial granularity at which the underlying implementation computes derivatives.

The name string for derivative computation granularity is `GL_ARB_derivative_control`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.153 Direct State Access

This extension allows state for most objects to be queried and modified without the object needing to be bound to a context.

The name string for direct state access is `GL_ARB_direct_state_access`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.154 Texture Subregion Readback

This extension adds commands allowing readback of subregions of texture images.

The name string for texture subregion readback is `GL_ARB_get_texture_sub_image`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.155 Shading Language Texture Sample Queries

This extension provides OpenGL Shading Language built-in functions allowing shaders to query the number of samples of a texture.

The name string for shading language texture sample queries is `GL_ARB_shader_texture_image_samples`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.156 Texture Barriers

This extension relaxes the restrictions on rendering to a currently bound texture and provides a mechanism to avoid read-after-write hazards.

The name string for texture barriers is `GL_ARB_texture_barrier`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.157 Context Flush Control

This extension allows querying the context flush behavior. There are associated EGL, GLX and WGL extensions which add new context creation parameters allowing specification of the desired behavior when a context is made non-current, as well as opting out of the implicit flush behavior of unextended OpenGL.

The name string for context flush control is `GL_KHR_context_flush_control`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.158 Robust Buffer Access Behavior

This extension specifies the behavior of out-of-bounds buffer and array accesses.

The name string for robust buffer access behavior is `GL_KHR_robust_buffer_access_behavior`.

K.3.3.159 Context Robustness

This extension adds APIs for safe access to application buffers by the GL implementation; mechanisms to learn about how graphics hardware resets affect the GL context; specification of behavior of GL commands after a graphics reset; and guarantees that out-of-bounds buffer object accesses by the GPU will have deterministic behavior.

The name string for context robustness is `GL_KHR_robustness`. This extension is equivalent to new core functionality introduced in OpenGL 4.5, and is provided to enable this functionality in older drivers.

K.3.3.160 Pipelin Statistics Queries

This extension introduces new query types that allow applications to get statistics information about different parts of the pipeline:

The name string for pipeline statistics queries is `GL_ARB_pipeline_statistics_query`.

K.3.3.161 Sparse Buffers

This extension adds a mechanism to decouple the virtual and physical storage requirements of buffer objects, and allows an application to create partially populated buffers that would over-subscribe available graphics memory if made fully resident.

The name string for sparse buffers is `GL_ARB_sparse_buffer`.

K.3.3.162 Transform Feedback Overflow Queries

This extension adds new query types which can be used to detect overflow of transform feedback buffers, and are accepted by conditional rendering commands.

The name string for transform feedback overflow queries is `GL_ARB_transform_feedback_overflow_query`.

K.3.3.163 Advanced Blend Equations

These extensions add a number of "advanced" blend equations. A base extension provides the new equations, but guarantees defined results only if each sample is touched no more than once in any single rendering pass, and defines a command to indicate pass boundaries. A layered extension guarantees that blending is done coherently and in API primitive order.

The name strings for advanced blend equations are `GL_KHR_blend_equation_advanced` and `GL_KHR_blend_equation_advanced_coherent`.

K.3.3.164 Undefined Error Behavior

This extension causes the GL implementation to have undefined behavior in any situation that would otherwise cause a GL error to be generated, allowing increased performance and decreased power usage for error-free applications. The extension is only supported for contexts created using associated EGL, GLX and WGL extensions which add corresponding new context creation parameters.

The name string for undefined error behavior is `GL_KHR_no_error`.

K.3.3.165 OpenGL ES 3.2 Compatibility

This extension adds supported for features of OpenGL ES 3.2 that are missing from OpenGL 4.5.

The name string for OpenGL ES 3.2 compatibility is `GL_ARB_ES3_2_compatibility`.

K.3.3.166 Fragment Shader Interlock

This extension provides new OpenGL Shading Language built-in functions that are used to delimit critical sections of fragment shader code. For pairs of shader invocations with "overlapping" coverage in a given pixel, the GL implementation will guarantee that the critical section of the fragment shader will be executed for only one fragment at a time.

The name string for fragment shader interlock is `GL_ARB_fragment_shader_interlock`.

K.3.3.167 64-bit Integer Shader Operations

This extension adds support for 64-bit integer data types to OpenGL Shading Language shaders, as well as related packing, conversion, relational, and mahtematical functions.

The name string for 64-bit integer shader operations is `GL_ARB_gpu_shader_int64`.

K.3.3.168 Parallel Shader Compilation

This extension provides a mechanism to allow compilation of OpenGL Shading Language shaders in multiple separate CPU threads, limit the number of threads to be used, and query if compilation is complete.

The name string for parallel shader compilation is `GL_ARB_parallel_shader_compile`.

K.3.3.169 Post Depth Coverage

This extension allows the fragment shader to control whether values in `gl_SampleMaskIn[]` reflect coverage after application of the early depth and stencil tests.

The name string for post depth coverage is `GL_ARB_post_depth_coverage`.

K.3.3.170 Explicit Sample Locations

This extension allows an application to modify the locations of samples within a pixel used in multisample rasterization, and to specify different sample locations for each pixel in a group of adjacent pixels

The name string for explicit sample locations is `GL_ARB_sample_locations`.

K.3.3.171 Shader Atomic Counter Operations

This extension extends the list of OpenGL Shading Language built-in functions that can operate on atomic counters.

The name string for shader atomic counter operations is `GL_ARB_shader_atomic_counter_ops`.

K.3.3.172 Shader Ballot

This extension provides the ability for a group of invocations which execute in lockstep to do limited forms of cross-invocation communication via a group broadcast of a invocation value, or broadcast of a bitarray representing a predicate value from each invocation in the group.

The name string for shader ballot is `GL_ARB_shader_ballot`.

K.3.3.173 Shader Clock

This extension exposes a 64-bit monotonically incrementing shader counter which may be used to derive local timing information within a single shader invocation.

The name string for shader clock is `GL_ARB_shader_clock`.

K.3.3.174 Shader Viewport Layer Array

This extension allows the `gl_Layer` and `gl_ViewportIndex` outputs to be written directly from the vertex shader with no geometry shader present.

The name string for shader viewport layer array is `GL_ARB_shader_viewport_layer_array`.

K.3.3.175 Sparse Texture 2

This extension adds additional shader functions and behaviors for use of sparse textures.

The name string for sparse texture 2 is `GL_ARB_sparse_texture2`.

K.3.3.176 Sparse Texture Clamp

This extension adds additional shader texture lookup functions that specify a minimum level-of-detail to use for lookups where the level-of-detail is computed automatically.

The name string for sparse texture clamp `GL_ARB_sparse_texture_clamp`.

K.3.3.177 Texture Filter Minmax

This extension provides a new texture and sampler parameter which allows applications to produce a filtered texel value by computing a component-wise minimum or maximum of the texels that would normally be averaged.

The name string for texture filter minmax is `GL_ARB_texture_filter_minmax`.

K.3.3.178 ASTC Sliced 3D Texture Compression

This extension supports slice-based 3D textures using the ASTC LDR profile, in the same fashion as the ASTC HDR profile when using the `GL_KHR_texture_compression_astc_ldr` extension.

The name string for astc sliced 3d texture compression is `GL_KHR_texture_compression_astc_sliced_3d`.

K.3.3.179 SPIR-V Shaders

This extension allows a SPIR-V module to be specified for use in a shader stage, rather than using OpenGL Shading Language, and modifies OpenGL Shading Language to be usable as a source language for creating SPIR-V modules, outside of the GL runtime.

The name string for SPIR-V shaders is `GL_ARB_gl_spirv`.

K.3.3.180 Create Context Undefined Error

These extensions allow the creation of an OpenGL or OpenGL ES context that doesn't generate errors, and has undefined behavior on errors. They are paired with `GL_KHR_no_error`.

The name strings for create context no error are `GLX_ARB_create_context_no_error` and `WGL_ARB_create_context_no_error`.

There is a similar `EGL_KHR_create_context_no_error` extension for creating contexts through EGL, located in the EGL Registry.

Index

x, 767
x_BIAS, 763
x_BITS, 814
x_SCALE, 763
x_SIZE, 767
#version, 4, 5, 713, 854, 857
2D, 696, 698, 815
2_BYTES, 700
3D, 696, 698
3D_COLOR, 696, 698
3D_COLOR_TEXTURE, 696, 698
3_BYTES, 700
4D_COLOR_TEXTURE, 696, 698
4_BYTES, 700

ACCUM, 641, 642
Accum, 27, 550, **642**, 642, 843
ACCUM_x_BITS, 814
ACCUM_ALPHA_BITS, 341
ACCUM_BLUE_BITS, 341
ACCUM_BUFFER_BIT, 636, 706, 843
ACCUM_CLEAR_VALUE, 756
ACCUM_GREEN_BITS, 341
ACCUM_RED_BITS, 341
ACTIVE_ATOMIC_COUNTER_-BUFFERS, 168, 778
ACTIVE_ATTRIBUTE_MAX_-LENGTH, 167, 772
ACTIVE_ATTRIBUTES, 167, 772
ACTIVE_PROGRAM, 169, 770
ACTIVE_RESOURCES, 106, 107, 779
ACTIVE_SUBROUTINE_MAX_-LENGTH, 174, 777
ACTIVE_SUBROUTINE_UNIFORM_-LOCATIONS, 153, 155, 156, 174, 777
ACTIVE_SUBROUTINE_UNIFORM_-MAX_LENGTH, 174, 777
ACTIVE_SUBROUTINE_UNI-FORMS, 174, 777
ACTIVE_SUBROUTINES, 153, 156, 174, 777
ACTIVE_TEXTURE, 179, 185, 278, 505, 685, 686, 752
ACTIVE_UNIFORM_BLOCK_-MAX_NAME_LENGTH, 167, 774
ACTIVE_UNIFORM_BLOCKS, 167, 774
ACTIVE_UNIFORM_MAX_LENGTH, 167, 772
ACTIVE_UNIFORMS, 167, 772
ACTIVE_VARIABLES, 110, 111, 138, 139, 780
ActiveShaderProgram, **124**, 140, 859, 871, 883
ActiveTexture, 156, **179**, 181, 604, 606
ADD, 599, 601, 602, 642
ADD_SIGNED, 602
ALIASED_LINE_WIDTH_RANGE, 794
ALIASED_POINT_SIZE_RANGE, 793
ALL_ATTRIB_BITS, 705, 706, 844

ALL_BARRIER_BITS, 163, 165
ALL_SHADER_BITS, 123
ALPHA, 193, 200, 217, 229, 230, 238, 241, 242, 246, 275, 285, 289, 323, 325, 366, 600, 601, 604, 623, 653, 657, 747, 749, 759, 763, 764, 766, 814, 842
ALPHA12, 242
ALPHA16, 242
ALPHA4, 241
ALPHA8, 241
ALPHA_BIAS, 227
ALPHA_BITS, 372, 843
ALPHA_INTEGER, 217
ALPHA_SCALE, 227, 599, 600, 753
ALPHA_TEST, 614, 754, 843
ALPHA_TEST_FUNC, 754
ALPHA_TEST_REF, 754, 876
AlphaFunc, **614**, 843
ALREADY_SIGNALED, 39
ALWAYS, 275, 325, 614–617, 754
AMBIENT, 517, 519, 740, 741
AMBIENT_AND_DIFFUSE, 517, 519, 521
AND, 627
AND_INVERTED, 627
AND_REVERSE, 627
Antialiasing, 568
ANY_SAMPLES_PASSED, 43, 46, 49, 438, 439, 617, 618, 883
ANY_SAMPLES_PASSED_CONSERVATIVE, 43, 46, 49, 438, 439, 617, 618, 883
ARB_arrays_of_arrays, 853, 863
ARB_base_instance, 846, 850
ARB_buffer_storage, 865
ARB_clear_buffer_object, 853, 863
ARB_clear_texture, 865
ARB_compressed_texture_pixel_storage, 845, 850
ARB_compute_shader, 853, 863
ARB_conservative_depth, 846, 850
ARB_copy_image, 853, 863
ARB_debug_group, 862
ARB_debug_label, 862
ARB_debug_output, 853, 863, 893
ARB_debug_output2, 863
ARB_enhanced_layouts, 866
ARB_ES3_compatibility, 853, 863
ARB_explicit_uniform_location, 853, 863
ARB_fragment_layer_viewport, 853, 863
ARB_framebuffer_no_attachments, 853, 863
ARB_gl_spirv, 898, 907
ARB_gpu_program5, 887
ARB_indirect_parameters, 907
ARB_internalformat_query, 846, 850
ARB_internalformat_query2, 853, 862
ARB_invalidate_subdata, 853, 863
ARB_map_buffer_alignment, 846, 850
ARB_multi_bind, 866
ARB_multi_draw_indirect, 853, 863
ARB_pipeline_statistics_query, 907
ARB_polygon_offset_clamp, 907
ARB_program_interface_query, 853, 863
ARB_query_buffer_object, 866
ARB_robust_buffer_access_behavior, 853, 863
ARB_robustness, 863
ARB_robustness_isolation, 863
ARB_shader_atomic_counter_ops, 907
ARB_shader_atomic_counters, 845, 850
ARB_shader_draw_parameters, 907
ARB_shader_group_vote, 907
ARB_shader_image_load_store, 846, 850
ARB_shader_image_size, 853, 862

ARB_shader_storage_buffer_object, 853, 863
ARB_shading_language_420pack, 846, 850
ARB_shading_language_packing, 850
ARB_spirv_extensions, 907
ARB_stencil_texturing, 853, 863
ARB_texture_buffer_range, 853, 863
ARB_texture_compression_bptc, 845, 850
ARB_texture_filter_anisotropic, 907
ARB_texture_mirror_clamp_to_edge, 866
ARB_texture_query_levels, 853, 862
ARB_texture_stencil8, 866
ARB_texture_storage, 845, 850
ARB_texture_storage_multisample, 853, 863
ARB_texture_view, 854, 863
ARB_transform_feedback_instanced, 846, 850
ARB_transform_feedback_overflow_query, 907
ARB_vertex_attrib_binding, 854, 863
ARB_vertex_type_10f_11f_11f_rev, 866
AreTexturesResident, **183**, 184, 702, 843
ARRAY_BUFFER, 61, 404, 412
ARRAY_BUFFER_BINDING, 412, 736
ARRAY_SIZE, 105, 110, 112, 136, 137, 445, 453, 780, 871, 888
ARRAY_STRIDE, 110, 112, 137, 780
ArrayElement, 410, **434**, 434, 436, 462, 697, 883
ArrayStride, 146, 147
ATOMIC_COUNTER_BARRIER_BIT, 163, 165
ATOMIC_COUNTER_BUFFER, 61, 62, 101, 105, 107–111, 138, 149
ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTER_INDICES, 139, 778
ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTERS, 139, 778
ATOMIC_COUNTER_BUFFER_BINDING, 84, 139, 778, 785
ATOMIC_COUNTER_BUFFER_DATA_SIZE, 139, 778
ATOMIC_COUNTER_BUFFER_INDEX, 110, 112, 137, 780
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_COMPUTE_SHADER, 139, 778
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_FRAGMENT_SHADER, 139, 778
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_GEOMETRY_SHADER, 139, 778
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_TESS_CONTROL_SHADER, 139, 778
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_TESS_EVALUATION_SHADER, 139
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_TESS_EVALUTION_SHADER, 778
ATOMIC_COUNTER_BUFFER_REFERENCED_BY_VERTEX_SHADER, 139, 778
ATOMIC_COUNTER_BUFFER_SIZE, 84, 785
ATOMIC_COUNTER_BUFFER_START, 84, 785

atomic_uint, 118, 149, 902
AtomicCounter, 828, 829
atomicCounter, 822
atomicCounterDecrement, 822
atomicCounterIncrement, 822
AtomicCounterMemory, 829
AtomicStorage, 833
AtomicStorageOps, 833
ATTACHED_SHADERS, 167, 170, 771
AttachShader, **94**, 702
ATTRIB_STACK_DEPTH, 815, 844
AUTO_GENERATE_MIPMAP, 715
AUTO_NORMAL, 688, 768
AUX0, 629
AUX_BUFFERS, 342, 629, 814

BACK, 347, 516, 518–521, 570, 573, 615, 619, 629, 631–635, 648, 743, 842, 890
BACK_LEFT, 347, 631, 632, 641
BACK_RIGHT, 347, 631, 632, 641
barrier, 473
BaseInstance, 832
BaseVertex, 832
Begin, 27, 376, 384, 389, 409, **432**, 432–434, 436, 437, 471, 494, 518, 519, 529, 562, 565, 689, 690, 697, 840
BeginConditionalRender, **437**, 438, 439, 935
BeginQuery, 44, **47**, 51, 52, 439, 467, 488, 498, 542, 597, 617, 670
BeginQueryIndexed, 44, **46**, 47, 48, 534, 535, 889, 905
BeginTransformFeedback, **527**, 527–532, 901
BGR, 200, 217, 653, 657
BGR_INTEGER, 217
BGRA, 200, 217, 220, 225, 399, 400, 403, 406, 411, 653
BGRA_INTEGER, 217, 220
BindAttribLocation, 131, **444**, 444, 702, 876, 892
BindBuffer, 29, 59, **60**, 62, 273, 401, 412, 702, 846, 858, 872
BindBufferBase, **62**, 63, 64, 85, 529, 532, 702, 847, 855, 858
BindBufferRange, 54, **62**, 63, 64, 83, 148, 150, 152, 529–532, 702, 847, 855, 858, 859, 874
BindBuffersBase, **63**, 64, 702
BindBuffersRange, **63**, 64, 65, 702
BindFragDataLocation, 131, **594**, 594, 595, 702
BindFragDataLocationIndexed, **594**, 595, 596, 624
BindFramebuffer, **339**, 340, 342, 369, 702
BindImageTexture, 54, **327**, 328–330, 336, 716, 721, 849
BindImageTextures, **329**, 329, 702
Binding, 128, 902
binding, 151
BindProgramPipeline, 98, **122**, 122–124, 156, 169, 466, 532
BindProgramPipelines, 702
BindRenderbuffer, **351**, 351, 352, 702, 884
BindSampler, 29, 185, **186**, 187, 189
BindSamplers, **187**, 187, 702
BindTexture, 156, 179, **180**, 180–183, 324, 888
BindTextures, **181**, 181, 702
BindTextureUnit, **182**
BindTransformFeedback, **526**, 526, 904
BindVertexArray, **396**, 397, 701, 898
BindVertexBuffer, **400**, 400, 402, 412, 872, 875
BindVertexBuffers, **401**, 401, 402, 702
BITMAP, 196, 203, 213, 215, 572, 580,

645, 657, 842
Bitmap, 27, 148, 373, 550, 576, **579**, 580, 591, 842
BITMAP_TOKEN, 698
BLEND, 599, 601, 618, 619, 625, 626, 755
BLEND_COLOR, 755
BLEND_DST_ALPHA, 755
BLEND_DST_RGB, 755
BLEND_EQUATION_ALPHA, 755
BLEND_EQUATION_RGB, 755
BLEND_SRC_ALPHA, 755
BLEND_SRC_RGB, 755
BlendColor, 621, **624**
BlendEquation, **619**, 619
BlendEquationi, **619**, 619
BlendEquationSeparate, **619**, 619
BlendEquationSeparatei, **619**, 619
BlendFunc, **621**, 622
BlendFunci, **622**, 622
BlendFuncSeparate, **621**, 622
BlendFuncSeparatei, **622**, 622
BlitFramebuffer, 27, 365, 644, **660**, 660, 661, 663, 702, 857, 860, 875, 876
BlitNamedFramebuffer, **660**, 660, 663
Block, 146
BLOCK_INDEX, 110, 112, 137, 780
BLUE, 193, 200, 217, 275, 323, 599, 653, 657, 747, 749, 759, 763, 764, 766, 814
BLUE_BIAS, 227
BLUE_BITS, 372, 843
BLUE_INTEGER, 217
BLUE_SCALE, 227
BOOL, 115
bool, 115, 144
BOOL_VEC2, 115
BOOL_VEC3, 115
BOOL_VEC4, 115
boolean, 141, 855
BUFFER, 679
BUFFER_ACCESS, 61, 67, 75, 737
BUFFER_ACCESS_FLAGS, 61, 67, 75, 78, 737
BUFFER_BINDING, 110, 112, 138, 139, 780
BUFFER_DATA_SIZE, 110, 112, 138, 139, 150, 151, 780, 899
BUFFER_IMMUTABLE_STORAGE, 61, 67–71, 737
BUFFER_MAP_LENGTH, 61, 67, 75, 78, 737
BUFFER_MAP_OFFSET, 61, 67, 75, 78, 737
BUFFER_MAP_POINTER, 61, 67, 75, 78, 81–83, 737
BUFFER_MAPPED, 61, 67, 75, 78, 737
BUFFER_SIZE, 61, 67, 70, 72, 75, 76, 80, 82, 152, 271, 737
BUFFER_STORAGE_FLAGS, 61, 67, 71, 73, 737
BUFFER_UPDATE_BARRIER_BIT, 162
BUFFER_USAGE, 61, 67, 74, 737
BUFFER_VARIABLE, 103, 110, 111, 113
BufferBlock, 146, 147, 829
BufferData, 56, 65, 67, **68**, 68–70, 78, 702, 858, 870
BufferStorage, **65**, 65, 67–69, 870, 871
BufferSubData, 56, 66, **70**, 70, 160, 164, 702, 874
BuiltIn, 128, 905
bvec2, 115, 141
bvec3, 115
bvec4, 115
BYTE, 201, 215, 335, 336, 398, 656, 657, 700

C3F_V3F, 427
C4F_N3F_V3F, 427
C4UB_V2F, 427
C4UB_V3F, 427
callback, **674**
CallList, 436, **699**, 700, 844
CallLists, 436, **699**, 700, 844
CAVEAT_SUPPORT, 715
CCW, 168, 569, 743, 776
ccw, 476
Centroid, 829
centroid, 588
centroid in, 588
CheckFramebufferStatus, 368, 369, **370**, 370, 702, 892
CheckNamedFramebufferStatus, **370**, 370
CLAMP, 276, 296, 297, 843
CLAMP_FRAGMENT_COLOR, 645, 662, 739, 841
CLAMP_READ_COLOR, 654, 739
CLAMP_TO_BORDER, 276, 290, 297
CLAMP_TO_EDGE, 276, 290, 297, 322, 661
CLAMP_VERTEX_COLOR, 524, 739, 841
ClampColor, 524, **645**, 654, 841
CLEAR, 627
Clear, 27, 374, 550, **636**, 637, 639, 843
CLEAR_BUFFER, 715
CLEAR_TEXTURE, 715, 877
ClearAccum, **637**, 843
ClearBuffer{if ui}v, **637**
ClearBufferData, **72**, 721
ClearBufferfi, **638**, 639
ClearBufferfv, 639
ClearBufferiv, 639
ClearBufferSubData, 66, **71**, 71, 72, 721, 858
ClearBufferuiv, 639
ClearColor, **636**, 638
ClearDepth, **637**, 638, 639, 856
ClearDepthf, **637**
ClearIndex, **636**
ClearNamedBufferData, **72**, 72
ClearNamedBufferSubData, **71**, 71
ClearNamedFramebuffer, **638**
ClearNamedFramebufferfi, **638**, 639
ClearNamedFramebufferfv, 639
ClearNamedFramebufferiv, 639
ClearNamedFramebufferuiv, 639
ClearStencil, **637**, 638, 639
ClearTexImage, **321**
ClearTexImageSubData, 71
ClearTexSubImage, **319**, 320, 321
CLIENT_ACTIVE_TEXTURE, 278, 407, 736
CLIENT_ALL_ATTRIB_BITS, 705, 706, 844
CLIENT_ATTRIB_STACK_DEPTH, 815, 844
CLIENT_MAPPED_BUFFER_BARRIER_BIT, 66, 162, 885
CLIENT_PIXEL_STORE_BIT, 706
CLIENT_STORAGE_BIT, 67
CLIENT_VERTEX_ARRAY_BIT, 706
ClientActiveTexture, **407**, 436, 701, 841, 872
ClientWaitSync, 37, 38, **39**, 39–41, 55, 702
CLIP_DEPTH_MODE, 738
CLIP_DISTANCEi, 738
CLIP_DISTANCE0, 539
CLIP_ORIGIN, 738
CLIP_PLANEi, 738
ClipControl, **537**
ClipDistance, 832, 833
CLIPPING_INPUT_PRIMITIVES, 44, 49, 542
CLIPPING_OUTPUT_PRIMITIVES,

44, 49, 542
ClipPlane, **538**, 841
COEFF, 692, 768
coherent, 163
COLOR, 198, 206, 208, 253, 500, 501, 505, 638, 639, 641, 658
Color, **394**, 394, 436, 438, 510, 519, 547
Color3, 394
Color4, 394
COLOR_ARRAY, 407, 426, 428, 732
COLOR_ARRAY_BUFFER_BINDING, 735
COLOR_ARRAY_POINTER, 710, 732
COLOR_ARRAY_SIZE, 732
COLOR_ARRAY_STRIDE, 732
COLOR_ARRAY_TYPE, 732
COLOR_ATTACHMENTm, 884
COLOR_ATTACHMENT0, 341, 634, 650
COLOR_BUFFER_BIT, 636, 639, 661–663, 706
COLOR_CLEAR_VALUE, 756
COLOR_COMPONENTS, 718
COLOR_ENCODING, 718
COLOR_INDEX, 196, 203, 213, 217, 228, 284, 572, 580, 644, 653, 658, 840
COLOR_INDEXES, 517, 522, 741
COLOR_LOGIC_OP, 626, 755
COLOR_MATERIAL, 519, 521, 740, 841
COLOR_MATERIAL_FACE, 740
COLOR_MATERIAL_PARAMETER, 740
COLOR_MATRIX, 209
COLOR_MATRIX (TRANSPOSE_COLOR_MATRIX), 738
COLOR_MATRIX_STACK_DEPTH, 209, 738
COLOR_RENDERABLE, 718
COLOR_SUM, 608, 739, 843
COLOR_TABLE, 197, 203, 229, 764
COLOR_TABLE_x_SIZE, 764
COLOR_TABLE_ALPHA_SIZE, 202
COLOR_TABLE_BIAS, 196–198, 202, 764
COLOR_TABLE_BLUE_SIZE, 202
COLOR_TABLE_FORMAT, 202, 764
COLOR_TABLE_GREEN_SIZE, 202
COLOR_TABLE_INTENSITY_SIZE, 202
COLOR_TABLE_LUMINANCE_SIZE, 202
COLOR_TABLE_RED_SIZE, 202
COLOR_TABLE_SCALE, 196–198, 202, 764
COLOR_TABLE_WIDTH, 202, 764
COLOR_WRITEMASK, 635, 756
ColorMask, **634**, 634, 636
ColorMaski, **634**, 634, 635, 876
ColorMaterial, **519**, 519–521, 687, 825, 841
ColorP*uiv, 395
ColorPointer, **405**, 406, 428, 436, 701, 840
ColorSubTable, 191, **199**, 199
ColorTable, 191, **196**, 198, 199, 203, 209, 212, 234, 235, 703
ColorTableParameter, **197**
ColorTableParameterfv, 196
Colorub, 547
Colorui, 547
Colorus, 547
COMBINE, 599, 600, 602, 606
COMBINE_ALPHA, 599, 600, 602, 603, 752
COMBINE_RGB, 599, 600, 602, 603, 752
COMMAND_BARRIER_BIT, 162, 902

COMPARE_REF_TO_TEXTURE, 275, 325
COMPATIBLE_SUBROUTINES, 110, 119, 155, 777, 780, 902
COMPILE, 697, 825
COMPILE_AND_EXECUTE, 697, 699, 700
COMPILE_STATUS, 89, 91, 100, 165, 166, 769
CompileShader, **89**, 89, 90, 702
Component, 128, 447, 595, 597
component, 113, 125, 442, 447, 595, 884
COMPRESSED_ALPHA, 246
COMPRESSED_INTENSITY, 246
COMPRESSED_LUMINANCE, 246
COMPRESSED_LUMINANCE_ALPHA, 246
COMPRESSED_R11_EAC, 246, 837
COMPRESSED_RED, 246
COMPRESSED_RED_RGTC1, 246, 310, 666, 836
COMPRESSED_RG, 246
COMPRESSED_RG11_EAC, 246, 837
COMPRESSED_RG_RGTC2, 246, 310, 666, 836
COMPRESSED_RGB, 246
COMPRESSED_RGB8_ETC2, 246, 837
COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2, 246, 837
COMPRESSED_RGB_BPTC_SIGNED_FLOAT, 246, 310, 666, 836
COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT, 246, 310, 666, 836
COMPRESSED_RGBA, 246
COMPRESSED_RGBA8_ETC2_EAC, 246, 837
COMPRESSED_RGBA_BPTC_UNORM, 246, 310, 666, 836
COMPRESSED_SIGNED_R11_EAC, 246, 837
COMPRESSED_SIGNED_RED_RGTC1, 246, 310, 666, 836
COMPRESSED_SIGNED_RG11_EAC, 246, 837
COMPRESSED_SIGNED_RG_RGTC2, 246, 310, 666, 836
COMPRESSED_SLUMINANCE, 246, 326
COMPRESSED_SLUMINANCE_ALPHA, 246, 326
COMPRESSED_SRGB, 246, 326
COMPRESSED_SRGB8_ALPHA8_ETC2_EAC, 246, 326, 837
COMPRESSED_SRGB8_ETC2, 246, 326, 837
COMPRESSED_SRGB8_PUNCHTHROUGH_ALPHA1_ETC2, 246, 326, 837
COMPRESSED_SRGB_ALPHA, 246, 326
COMPRESSED_SRGB_ALPHA_BPTC_UNORM, 246, 310, 326, 666, 836
COMPRESSED_TEXTURE_FORMATS, 239, 795, 840, 846
CompressedTexImage1D, 249, **261**, 262, 264–266, 721
CompressedTexImage2D, **261**, 262–266, 721
CompressedTexImage3D, **261**, 262–266, 287, 721, 867, 901
CompressedTexSubImage*D, 267
CompressedTexSubImage1D, **265**
CompressedTexSubImage2D, **265**

CompressedTexSubImage3D, **265**, 288
CompressedTextureSubImage*D, 267
CompressedTextureSubImage1D, **265**
CompressedTextureSubImage2D, **266**
CompressedTextureSubImage3D, **266**
COMPUTE_SHADER, 88, 154, 668, 770, 870
COMPUTE_SHADER_BIT, 123
COMPUTE_SHADER_INVOCATIONS, 44, 49, 670
COMPUTE_SUBROUTINE, 102, 154
COMPUTE_SUBROUTINE_UNIFORM, 102, 107, 110, 111, 119, 154
COMPUTE_TEXTURE, 715
COMPUTE_WORK_GROUP_SIZE, 168, 169, 669, 771
CONDITION_SATISFIED, 39
CONSTANT, 603, 604, 753
CONSTANT_ALPHA, 623
CONSTANT_ATTENUATION, 517, 741
CONSTANT_BORDER, 231–233
CONSTANT_COLOR, 623
CONTEXT_COMPATIBILITY_PROFILE_BIT, 712
CONTEXT_CORE_PROFILE_BIT, 712
CONTEXT_FLAG_DEBUG_BIT, 671, 712
CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT, 711
CONTEXT_FLAG_NO_ERROR_BIT, 712
CONTEXT_FLAG_ROBUST_ACCESS_BIT, 712
CONTEXT_FLAGS, 671, 711, 712, 796
CONTEXT_LOST, 17, 18, 20, 21
CONTEXT_PROFILE_MASK, 711, 712, 796, 885
CONTEXT_RELEASE_BEHAVIOR, 712, 792
CONTEXT_RELEASE_BEHAVIOR_FLUSH, 712
CONVOLUTION_xD, 765
CONVOLUTION_1D, 204, 206–208, 229, 251, 765
CONVOLUTION_2D, 203–205, 207, 208, 230, 250, 765
CONVOLUTION_BORDER_COLOR, 208, 232, 765
CONVOLUTION_BORDER_MODE, 208, 231, 765
CONVOLUTION_FILTER_BIAS, 203–205, 208, 765
CONVOLUTION_FILTER_SCALE, 203–206, 208, 765
CONVOLUTION_FORMAT, 208, 765
CONVOLUTION_HEIGHT, 208, 765
CONVOLUTION_WIDTH, 208, 765
ConvolutionFilter1D, 191, **204**, 205, 206
ConvolutionFilter2D, 191, **203**, 204–206
ConvolutionParameter, **204**, **231**
ConvolutionParameterfv, 203, 204, 232
ConvolutionParameteriv, 205, 232
COORD_REPLACE, 557, 560, 752
COPY, 627, 628, 755
COPY_INVERTED, 627
COPY_PIXEL_TOKEN, 698
COPY_READ_BUFFER, 61, 80, 846, 847, 858
COPY_READ_BUFFER_BINDING, 815, 847
COPY_WRITE_BUFFER, 61, 80, 846, 847, 858
COPY_WRITE_BUFFER_BINDING, 815, 847
CopyBufferSubData, 66, **80**, 80, 81, 891

CopyColorSubTable, **199**, 199, 658
CopyColorTable, **198**, 199
CopyConvolutionFilter1D, **206**
CopyConvolutionFilter2D, **205**
CopyImageSubData, 239, **664**, 664–666, 886, 904
CopyNamedBufferSubData, **80**, 80, 81
CopyPixels, 27, 190, 193, 198, 205, 206, 230, 253, 373, 374, 550, 644, **658**, 658–662, 695, 843
CopyTexImage, 320, 843
CopyTexImage1D, 230, **254**, 254, 257, 302, 721
CopyTexImage2D, 230, **253**, 254, 257, 302, 721
CopyTexSubImage1D, 230, **255**, 256
CopyTexSubImage2D, 230, **255**, 256
CopyTexSubImage3D, 230, **255**, 256
CopyTextureSubImage1D, **256**, 256
CopyTextureSubImage2D, **255**, 256
CopyTextureSubImage3D, **255**, 256, 257
CreateBuffers, 29, **59**, 59
CreateFramebuffers, **340**, 340
CreateProgram, 29, **93**, 702
CreateProgramPipelines, **122**, 122
CreateQueries, **45**, 45, 48, 52, 889, 905
CreateRenderbuffers, **352**, 352
CreateSamplers, **185**, 186, 887
CreateShader, **87**, 88, 702
CreateShaderProgram, 702
CreateShaderProgramv, **99**, 100
CreateTextures, 29, **182**, 183, 279
CreateTransformFeedbacks, **527**, 527
CreateVertexArrays, 29, **397**, 397, 898
CULL_FACE, 570, 743
CULL_FACE_MODE, 743
CullDistance, 832, 833
CullFace, 569, **570**, 570, 575
CURRENT_BIT, 706
CURRENT_COLOR, 731
CURRENT_FOG_COORD, 731
CURRENT_INDEX, 731
CURRENT_NORMAL, 731
CURRENT_PROGRAM, 771
CURRENT_QUERY, 48, 49, 815, 906
CURRENT_RASTER_COLOR, 731
CURRENT_RASTER_DISTANCE, 731
CURRENT_RASTER_INDEX, 731
CURRENT_RASTER_POSITION, 731
CURRENT_RASTER_POSITION_VALID, 731
CURRENT_RASTER_SECONDARY_COLOR, 731
CURRENT_RASTER_TEXTURE_COORDS, 731, 824
CURRENT_SECONDARY_COLOR, 731
CURRENT_TEXTURE_COORDS, 731
CURRENT_VERTEX_ATTRIB, 430, 782, 882
CW, 168, 569
cw, 476

DEBUG_CALLBACK_FUNCTION, 675, 681, 710, 807
DEBUG_CALLBACK_USER_PARAM, 681, 710, 807
DEBUG_GROUP_STACK_DEPTH, 807
DEBUG_LOGGED_MESSAGES, 675, 807
DEBUG_NEXT_LOGGED_MESSAGE_LENGTH, 675, 807
DEBUG_OUTPUT, 463, 671, 675–677, 807
DEBUG_OUTPUT_SYNCHRONOUS, 680, 681, 807
DEBUG_SEVERITY_HIGH, 673

DEBUG_SEVERITY_LOW, 673, 674
DEBUG_SEVERITY_MEDIUM, 673
DEBUG_SEVERITY_NOTIFICATION, 673, 678
DEBUG_SOURCE_API, 19, 672, 715
DEBUG_SOURCE_APPLICATION, 672, 677, 678
DEBUG_SOURCE_OTHER, 672
DEBUG_SOURCE_SHADER_COMPILER, 672
DEBUG_SOURCE_THIRD_PARTY, 672, 677, 678
DEBUG_SOURCE_WINDOW_SYSTEM, 672
DEBUG_TYPE_DEPRECATED_BEHAVIOR, 673
DEBUG_TYPE_ERROR, 19, 673
DEBUG_TYPE_MARKER, 673
DEBUG_TYPE_OTHER, 673
DEBUG_TYPE_PERFORMANCE, 673, 715
DEBUG_TYPE_POP_GROUP, 673, 678
DEBUG_TYPE_PORTABILITY, 673
DEBUG_TYPE_PUSH_GROUP, 673, 678
DEBUG_TYPE_UNDEFINED_BEHAVIOR, 673
DebugMessageCallback, **674**, 675, 681, 702, 876
DebugMessageControl, 674, **676**, 702, 857, 876
DebugMessageInsert, **677**, 677, 702, 893
DECAL, 599, 601
DECR, 616
DECR_WRAP, 616
DELETE_STATUS, 90, 165, 166, 769, 771
DeleteBuffers, 29, 54, **60**, 62, 65, 702, 858
DeleteFramebuffers, 340, **342**, 702
DeleteLists, **701**, 701, 844
DeleteProgram, **99**, 99, 702
DeleteProgramPipelines, **121**, 122–124, 169, 466, 702
DeleteQueries, **45**, 702
DeleteRenderbuffers, 54, 351, **352**, 369, 702
DeleteSamplers, 187, **189**, 702, 867
DeleteShader, **90**, 90, 702
DeleteSync, **38**, 39, 42
DeleteTextures, 54, **183**, 328, 369, 702
DeleteTransformFeedbacks, **525**, 527, 702, 848, 861
DeleteVertexArrays, **396**, 397, 701
DEPTH, 253, 347, 348, 638, 639, 641, 658, 749, 759, 763
Depth, 828
DEPTH24_STENCIL8, 245
DEPTH32F_STENCIL8, 245
DEPTH_ATTACHMENT, 341, 357, 367
DEPTH_BIAS, 193, 227
DEPTH_BITS, 372, 660, 814, 843
DEPTH_BUFFER_BIT, 636, 639, 661, 663, 706
DEPTH_CLAMP, 539, 738
DEPTH_CLEAR_VALUE, 756
DEPTH_COMPONENT, 196, 197, 203, 217, 238, 239, 245, 249, 275, 283, 284, 320, 321, 323, 324, 366, 459, 605, 644, 652–654, 658, 660, 748
DEPTH_COMPONENT16, 245
DEPTH_COMPONENT24, 245
DEPTH_COMPONENT32, 245
DEPTH_COMPONENT32F, 245
DEPTH_COMPONENTS, 718
DEPTH_FUNC, 754
DEPTH_RANGE, 738, 877

DEPTH_RENDERABLE, 718
DEPTH_SCALE, 193, 227
DEPTH_STENCIL, 196, 197, 203, 217, 220, 225, 226, 237–239, 245, 249, 253, 283, 284, 305, 307, 320, 321, 324, 325, 357, 363, 366, 459, 605, 638, 639, 644, 647, 648, 652–654, 658, 660
DEPTH_STENCIL_ATTACHMENT, 347, 350, 357, 363, 856
DEPTH_STENCIL_TEXTURE_MODE, 275, 305, 307, 323, 325, 459, 748, 868, 887
DEPTH_TEST, 616, 754
DEPTH_TEXTURE_MODE, 275, 323–325, 748, 842
DEPTH_WRITEMASK, 756
DepthFunc, **617**, 648
DepthMask, **635**, 636, 648
DepthRange, 15, **544**, 544, 579, 825
DepthRangeArrayv, **544**, 544
DepthRangef, **544**
DepthRangeIndexed, **544**, 544
DerivativeControl, 833
DetachShader, **94**, 702
Device, 828
dFdx, 704
dFdy, 704
DIFFUSE, 517, 519, 740, 741
Disable, 179, 234–236, 290, **409**, 504, 506, 509, 512, 516, 518, 519, 539, 550, 553–556, 562, 565, 569, 570, 572, 583, 584, 605, 606, 608, 612, 614–616, 619, 626, 671, 680, 686, 688, 841–843, 866
DisableClientState, **407**, 407, 426, 428, 436, 701, 841
Disablei, 583, **618**, 619
DisableVertexArrayAttrib, **404**, 404
DisableVertexAttribArray, **404**, 404, 430, 701
DISPATCH_INDIRECT_BUFFER, 61, 162, 413, 669, 670, 858, 904
DISPATCH_INDIRECT_BUFFER_BINDING, 791
DispatchCompute, **668**, 669, 885
DispatchComputeIndirect, 162, 413, **669**, 702, 858, 872, 901, 904
DISPLAY_LIST, 679
DITHER, 626, 755
dmat$C$, 144
dmat$C$xR, 144
dmat*, 847, 849
dmat2, 115, 443
dmat2x3, 115, 443, 446
dmat2x4, 115, 443, 446
dmat3, 115, 141, 443, 446
dmat3x2, 115, 443
dmat3x4, 116, 443, 446
dmat4, 115, 443, 446
dmat4x2, 116, 443
dmat4x3, 116, 443, 446
do, 473
DOMAIN, 692, 768
DONT_CARE, 676, 703, 704, 790
DOT3_RGB, 602
DOT3_RGBA, 602
DOUBLE, 115, 398, 435
double, 115, 126, 144, 443, 544, 854
DOUBLE_MAT2, 115
DOUBLE_MAT2x3, 115
DOUBLE_MAT2x4, 115
DOUBLE_MAT3, 115
DOUBLE_MAT3x2, 115
DOUBLE_MAT3x4, 116
DOUBLE_MAT4, 115
DOUBLE_MAT4x2, 116
DOUBLE_MAT4x3, 116

DOUBLE_VEC2, 115
DOUBLE_VEC3, 115
DOUBLE_VEC4, 115
DOUBLEBUFFER, 346, 813
DRAW_BUFFER, 634
DRAW_BUFFER*i*, 758
DRAW_BUFFER0, 634
DRAW_FRAMEBUFFER, 339–347, 349, 356–360, 370, 629, 630, 640, 643, 660, 757, 856
DRAW_FRAMEBUFFER_BINDING, 301, 343, 371, 372, 433, 581, 644, 660, 757
DRAW_INDIRECT, 417, 424
DRAW_INDIRECT_BUFFER, 61, 162, 413, 904
DRAW_INDIRECT_BUFFER_BINDING, 736
DRAW_PIXEL_TOKEN, 698
DrawArrays, 376, 380, 397, 409, **414**, 415, 416, 462, 697, 892
DrawArraysIndirect, 413, **415**, 417, 860, 872
DrawArraysInstanced, **415**, 416, 533, 860
DrawArraysInstancedBaseInstance, **415**, 415, 420, 860, 906
DrawArraysOneInstance, **414**, 414, 892
DrawBuffer, 628, **629**, 629–631, 633, 636, 639, 856, 876
DrawBuffers, 629, **630**, 630, 632, 633, 856
DrawElements, 158, 397, 409, 412, **419**, 420–422, 697
DrawElementsBaseVertex, 412, **422**, 425, 426
DrawElementsIndirect, 413, **422**, 423, 424, 860, 872
DrawElementsInstanced, 412, **420**, 421, 860, 898
DrawElementsInstancedBaseInstance, **420**, 420, 860
DrawElementsInstancedBaseVertex, 413, **422**, 423, 860
DrawElementsInstancedBaseVertexBaseInstance, **422**, 860
DrawElementsOneInstance, **418**, 418–420, 422, 892, 898
DrawIndex, 832
DrawParameters, 833
DrawPixels, 27, 148, 190, 191, 193, 196, 203, 219, 230, 373, 533, 550, 572, 576, 580, 591, **644**, 644–647, 658, 695, 842
DrawRangeElements, 412, **421**, 422, 697, 794
DrawRangeElementsBaseVertex, 412, **422**, 422
DrawTransformFeedback, **533**, 534
DrawTransformFeedbackInstanced, **533**, 533, 860
DrawTransformFeedbackStream, **533**, 533
DrawTransformFeedbackStreamInstanced, **533**, 533, 534, 860
DST_ALPHA, 623
DST_COLOR, 623
dvec2, 115, 443
dvec3, 115, 443, 446, 847, 849
dvec4, 115, 443, 444, 446, 847, 849
DYNAMIC_COPY, 61, 69
DYNAMIC_DRAW, 61, 67, 69
DYNAMIC_READ, 61, 69
DYNAMIC_STORAGE_BIT, 66, 67, 69, 71, 871

early_fragment_tests, 597
EDGE_FLAG, 731
EDGE_FLAG_ARRAY, 407, 426, 734

EDGE_FLAG_ARRAY_BUFFER_-
BINDING, 735
EDGE_FLAG_ARRAY_POINTER,
710, 734
EDGE_FLAG_ARRAY_STRIDE, 734
EdgeFlag, **389**, 434, 436
EdgeFlagPointer, **405**, 405, 406, 436,
701, 840
EdgeFlagv, **389**
EGL_KHR_create_context_no_error,
941
ELEMENT_ARRAY_BARRIER_BIT,
161
ELEMENT_AR-
RAY_BUFFER, 61, 161, 412,
431
ELEMENT_ARRAY_BUFFER_BIND-
ING, 428, 431, 735, 892
EMISSION, 517, 519, 740
EmitStreamVertex, 493
Enable, 179, 234–236, 290, **409**, 504,
506, 509, 512, 516, 518, 519,
539, 550, 553–556, 562, 565,
569, 570, 572, 583, 584, 605,
606, 608, 612, 614–616, 619,
626, 671, 680, 686, 688, 709,
841–843, 866
ENABLE_BIT, 706
EnableClientState, **407**, 407, 426, 428,
436, 701, 840
Enablei, 583, **618**, 619
EnableVertexArrayAttrib, **404**, 404
EnableVertexAttribArray, 397, **404**,
404, 430, 701
End, 27, 376, 389, 409, **432**, 432–434,
436, 437, 518, 519, 562, 689,
690, 697, 840
EndConditionalRender, **437**, 438, 439
EndList, **699**, 699, 844
EndPrimitive, 374, 492, 498
EndQuery, **48**, 51, 52, 617, 618
EndQueryIndexed, **47**, 47, 905
EndStreamPrimitive, 492, 498
EndTransformFeedback, 56, 57, **527**,
527, 528, 532, 534
EQUAL, 168, 275, 325, 614, 616, 617,
776
equal_spacing, 476, 482
EQUIV, 627
EVAL_BIT, 706
EvalCoord, 436, 687
EvalCoord1, **686**, 687, 690
EvalCoord12, **687**
EvalCoord1d, 689
EvalCoord1f, 689
EvalCoord2, **687**, 687, 689–691
EvalMesh1, 27, **689**, 691
EvalMesh2, 27, **689**, 690, 691
EvalPoint, 436
EvalPoint1, **690**
EvalPoint2, **691**
ExecutionMode, 93
EXP, 608, 610, 739
exp(x), 834
EXP2, 608
exp2(x), 834
exp2(y * log2(x)), 834
EXT_texture_filter_anisotropic, 907
EXTENSIONS, 192, 710–713, 796,
844, 910
EYE_LINEAR, 508–510, 752
EYE_PLANE, 508, 509, 752

FALSE, 14, 15, 42, 46, 48, 50, 60, 61,
67, 69, 78, 88–91, 93, 95, 98–
100, 121, 130, 131, 140, 141,
165, 166, 168, 175, 176, 183,
184, 189, 191, 193, 209–212,
228, 235, 236, 275, 322, 328–
330, 332, 343, 349, 353, 363,

389, 390, 397, 399, 400, 410, 430, 432, 438, 464, 465, 513, 515, 526, 536, 550, 557, 575, 585, 618, 645, 648, 651, 654, 671, 676, 680, 718, 720, 722, 723, 732–734, 736–745, 748, 750, 752, 754, 755, 758, 759, 762–771, 776, 778, 782–784, 787, 807, 815
false, 590
FASTEST, 703, 704
FEEDBACK, 693, 695, 696, 826
FEEDBACK_BUFFER_POINTER, 710, 815
FEEDBACK_BUFFER_SIZE, 815
FEEDBACK_BUFFER_TYPE, 815
FeedbackBuffer, 693, **695**, 696, 701, 710, 844
FenceSync, 29, **37**, 37, 38, 41, 56, 66, 67, 702, 885
FILL, 573, 575, 576, 689, 743, 825
FILTER, 715
Finish, **21**, 21, 37, 56, 66, 67, 702, 824
FIRST_VERTEX_CONVENTION, 497, 536
FIXED, 398
FIXED_ONLY, 646, 654, 662, 667, 739
FLAT, 537, 825
Flat, 829
flat, 491, 537, 889
FLOAT, 115, 195, 196, 215, 281, 332, 335, 348, 398, 426–428, 432, 435, 654–656, 700, 720, 732–734
float, 115, 126, 144, 443
Float64, 833
FLOAT_32_UNSIGNED_INT_24_8_REV, 215, 216, 218, 220, 224, 652, 656, 657
FLOAT_MAT2, 115
FLOAT_MAT2x3, 115
FLOAT_MAT2x4, 115
FLOAT_MAT3, 115
FLOAT_MAT3x2, 115
FLOAT_MAT3x4, 115
FLOAT_MAT4, 115
FLOAT_MAT4x2, 115
FLOAT_MAT4x3, 115
FLOAT_VEC2, 115
FLOAT_VEC3, 115
FLOAT_VEC4, 115
Flush, **21**, 21, 41, 702, 824, 901
FlushMappedBufferRange, 56, 74, **77**, 77, 702
FlushMappedNamedBufferRange, **77**, 77
fma(), 834
FOG, 608, 739, 843
Fog, **608**, 609, 843
FOG_BIT, 706
FOG_COLOR, 609, 739
FOG_COORD, 577, 608, 609
FOG_COORD_ARRAY, 407, 426, 732
FOG_COORD_ARRAY_BUFFER_BINDING, 735
FOG_COORD_ARRAY_POINTER, 710, 732
FOG_COORD_ARRAY_STRIDE, 732
FOG_COORD_ARRAY_TYPE, 732
FOG_COORD_SRC, 579, 608, 610, 739
FOG_DENSITY, 609, 739
FOG_END, 609, 739
FOG_HINT, 704, 790, 844
FOG_INDEX, 609, 739
FOG_MODE, 608, 610, 739
FOG_START, 609, 739
FogCoord, **393**, 436
FogCoordPointer, **405**, 406, 436, 701, 840
for, 473

FRACTIONAL_EVEN, 168
fractional_even_spacing, 476
FRACTIONAL_ODD, 168
fractional_odd_spacing, 476
FragCoord, 832
FragDepth, 832
FRAGMENT_DEPTH, 608–610, 739
FRAGMENT_INTERPOLATION_OFFSET_BITS, 588, 810
FRAGMENT_SHADER, 88, 154, 172, 770
FRAGMENT_SHADER_BIT, 123
FRAGMENT_SHADER_DERIVATIVE_HINT, 704, 790
FRAGMENT_SHADER_INVOCATIONS, 44, 49, 597
FRAGMENT_SUBROUTINE, 102, 154
FRAGMENT_SUBROUTINE_UNIFORM, 102, 107, 110, 111, 119, 154
FRAGMENT_TEXTURE, 715
FRAMEBUFFER, 340, 343–347, 349, 356–360, 370, 640, 679, 856
FRAMEBUFFER_ATTACHMENT_x_SIZE, 759
FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE, 348
FRAMEBUFFER_ATTACHMENT_BLUE_SIZE, 348
FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, 254, 348, 620, 625, 627, 642, 662, 759
FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, 348, 759
FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE, 348
FRAMEBUFFER_ATTACHMENT_GREEN_SIZE, 348
FRAMEBUFFER_ATTACHMENT_LAYERED, 349, 363, 759, 873
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, 348–350, 356, 363, 366, 759
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, 347–350, 356, 362, 366, 372, 759, 882, 901
FRAMEBUFFER_ATTACHMENT_RED_SIZE, 348
FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE, 348
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE, 349, 363, 759
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER, 349, 362, 363, 372, 759, 868
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, 301, 349, 363, 759
FRAMEBUFFER_BARRIER_BIT, 162, 165
FRAMEBUFFER_BINDING, 343
FRAMEBUFFER_BLEND, 716
FRAMEBUFFER_COMPLETE, 370
FRAMEBUFFER_DEFAULT, 347
FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS, 343, 344, 346, 758
FRAMEBUFFER_DEFAULT_HEIGHT, 343, 344, 346, 368, 758
FRAMEBUFFER_DEFAULT_LAYERS, 343, 344, 346, 758

FRAMEBUFFER_DEFAULT_SAMPLES, 343, 344, 346, 758
FRAMEBUFFER_DEFAULT_WIDTH, 343, 344, 346, 368, 758
FRAMEBUFFER_INCOMPLETE_ATTACHMENT, 368
FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS, 368
FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT, 368
FRAMEBUFFER_INCOMPLETE_MULTISAMPLE, 368
FRAMEBUFFER_RENDERABLE, 370, 716, 903
FRAMEBUFFER_RENDERABLE_LAYERED, 371, 716, 903
FRAMEBUFFER_SRGB, 620, 625, 627, 642, 661, 717, 755, 873
FRAMEBUFFER_UNDEFINED, 367
FRAMEBUFFER_UNSUPPORTED, 368, 369
FramebufferParameteri, **343**, 343, 344, 346, 868
FramebufferRenderbuffer, **356**, 356, 357, 369, 702, 869
FramebufferTexture, 54, **358**, 358, 359, 361, 702, 892
FramebufferTexture1D, **359**, 360, 702
FramebufferTexture2D, **359**, 360, 702
FramebufferTexture3D, **359**, 360, 702
FramebufferTextureLayer, **361**, 702, 898
FRONT, 347, 516, 518, 519, 521, 570, 573, 615, 619, 629, 631, 633–635, 648, 842, 890
FRONT_AND_BACK, 516, 518–521, 570, 573, 615, 619, 631, 633–635, 648
FRONT_FACE, 743
FRONT_LEFT, 347, 631, 632, 641
FRONT_RIGHT, 347, 631, 632, 641
FrontFace, 516, **569**, 569, 590, 841
FrontFacing, 832
Frustum, 502, **503**, 825, 841
ftransform, 463
FULL_SUPPORT, 370, 371, 715, 903
FUNC_ADD, 621, 625, 755
FUNC_REVERSE_SUBTRACT, 621
FUNC_SUBTRACT, 621
Function, 828
fwidth, 704

GenBuffers, 29, **59**, 59, 60, 702
GENERATE_MIPMAP, 275, 276, 304, 322, 748, 843
GENERATE_MIPMAP_HINT, 704, 790, 844
GenerateMipmap, **303**, 303, 304, 702, 882
GenerateTextureMipmap, **303**, 303, 304
GenFramebuffers, 339, 340, **342**, 342, 702
GenLists, **700**, 701, 844
GenProgramPipelines, **121**, 121–124, 169, 466, 702
GenQueries, **44**, 44, 45, 702
GenRenderbuffers, 351, **352**, 352, 702
GenSamplers, **185**, 185–190, 702
GenTextures, **180**, 180, 183, 311, 702, 847
GenTransformFeedbacks, **525**, 525–527, 702
GenVertexArrays, **396**, 396, 397, 701
Geometry, 833
GEOMETRY_INPUT_TYPE, 167, 168, 489, 773

GEOMETRY_OUTPUT_TYPE, 167, 168, 491, 773
GEOMETRY_SHADER, 88, 154, 488, 770
GEOMETRY_SHADER_BIT, 123
GEOMETRY_SHADER_INVOCATIONS, 44, 49, 167, 168, 498, 773
GEOMETRY_SHADER_PRIMITIVES_EMITTED, 44, 49, 498
GEOMETRY_SUBROUTINE, 102, 154
GEOMETRY_SUBROUTINE_UNIFORM, 102, 107, 110, 111, 119, 154
GEOMETRY_TEXTURE, 716
GEOMETRY_VERTICES_OUT, 167, 168, 491, 495, 773
GeometryPointSize, 833
GeometryStreams, 833
GEQUAL, 275, 325, 614, 616, 617
GET_TEXTURE_IMAGE_FORMAT, 718
GET_TEXTURE_IMAGE_TYPE, 718
GetActiveSubroutineUniformName, 777
GetActiveSubroutineName, 777
GetActiveSubroutineUniformiv, 777
GetActiveUniformBlockiv, 775, 776
GetActiveAtomicCounterBufferiv, 778
GetActiveAtomicCounterBufferiv, **138**, 139
GetActiveAttrib, **445**, 445, 772, 861, 884
GetActiveSubroutineName, **154**, 898
GetActiveSubroutineUniformiv, **155**, 155
GetActiveSubroutineUniformName, **155**, 898
GetActiveUniform, **136**, 141, 772, 871
GetActiveUniformBlockiv, **137**, 138
GetActiveUniformBlockName, **137**
GetActiveUniformName, **135**
GetActiveUniformsiv, **136**, 137, 774, 775, 778
GetAttachedShaders, **169**, 771
GetAttribLocation, 444, **446**, 772
GetBooleani_v, 635, **709**, 756, 784
GetBooleanv, 14, 410, 505, 536, 635, **708**, 724, 731, 740, 744, 756, 762, 763, 787, 792, 795, 809, 813, 814
GetBufferParameteri64v, **81**, 737
GetBufferParameteriv, **81**, 737, 893
GetBufferPointerv, **83**, 83, 737
GetBufferSubData, **82**, 82, 737
GetClipPlane, **542**, 738
GetColorTable, **200**, 203, 651, 764
GetColorTableParameterfv, 764
GetColorTableParameteriv, 764
GetColorTableParameter, **202**
GetCompressedTexImage, 264, 266, 281, **286**, 287, 288, 651, 704
GetCompressedTexSubImage, 288
GetCompressedTextureImage, **286**, 287
GetCompressedTextureSubImage, **288**
GetConvolutionFilter, 765
GetConvolutionParameterfv, 765
GetConvolutionParameteriv, 765, 793
GetConvolutionFilter, **207**, 651
GetConvolutionParameter, **208**
GetConvolutionParameteriv, 204
GetDebugMessageLog, 675, **681**, 681, 682, 702, 877
GetDoublei_v, **709**, 738
GetDoublev, 15, 505, **708**
GetError, **16**, 16, 19, 20, 815
GetFloati_v, **709**, 738
GetFloatv, **11**, 15, 209, 505, 506, 546,

554, **708**, 731, 738–740, 742–744, 754–756, 763, 766–768, 792–794, 810, 876
GetFragDataIndex, **596**, 884
GetFragDataLocation, **596**, 884
GetFramebufferAttachment-Parameteriv, 759
GetFramebufferParameteriv, 758
GetFramebufferAttachmentParameteriv, **346**, 347, 349, 868, 872, 875, 891, 892, 901
GetFramebufferParameteriv, **345**, 345, 346, 350, 856, 891
GetGraphicsResetStatus, **19**, 20
GetHistogram, **210**, 211, 651, 766
GetHistogramParameteriv, 766
GetHistogramParameter, **211**
GetInteger, 713
GetInteger64i_v, 85, **709**, 724, 735, 785–788
GetInteger64v, 15, 40, 52, 419, **708**, 792, 804, 809, 847
GetIntegeri_v, 83, 85, 585, 625, 669, **709**, 724, 726, 735, 744, 754, 755, 784–788, 802
GetIntegeriv, 815
GetIntegerv, 15, 20, 52, 133, 143, 147, 149, 151, 179, 186, 209, 239, 343, 350, 351, 421, 431, 497, 505, 538, 622, 625, 632, 634, 650, 651, 669, 700, **708**, 711, 712, 724–726, 731–736, 738–740, 742, 743, 745, 746, 752, 754–758, 760, 762, 763, 767, 771, 785, 786, 788, 790–810, 812–815
GetInternalformati64v, **714**, 721, 898
GetInternalformativ, 270, 354, 370, **714**, 811, 898
GetLight, **519**
GetLightfv, 741
GetMap, **691**
GetMapfv, 768
GetMapiv, 768
GetMaterial, **521**
GetMaterialfv, 740, 741
GetMinmax, **212**, 651, 767
GetMinmaxParameteriv, 767
GetMinmaxParameter, **213**
GetMultisamplefv, 456, **553**, 554, 813, 869
GetNamedBufferParameteri64v, **81**
GetNamedBufferParameteriv, **81**
GetNamedBufferPointerv, **83**, 83
GetNamedBufferSubData, **82**, 82
GetNamedFramebufferAttachmentParameteriv, **346**, 347, 350, 891
GetNamedFramebufferParameteriv, **345**, 345, 346, 350, 891
GetNamedRenderbufferParameteriv, **355**, 355, 356
GetnColorTable, **200**, 202
GetnCompressedTexImage, **286**, 287
GetnConvolutionFilter, **207**, 207
GetnHistogram, **210**, 210
GetnMap, **691**
GetnMinmax, **212**, 212
GetnPixelMap, **195**
GetnPolygonStipple, **572**, 572
GetnSeparableFilter, **207**, 208
GetnTexImage, **282**, 282, 283, 287, 884
GetnUniformdv, **173**
GetnUniformfv, **173**
GetnUniformiv, **173**
GetnUniformuiv, **173**
GetObjectLabel, **682**, 683, 734, 737, 748, 751, 758, 761, 769–771, 783, 787, 815
GetObjectPtrLabel, **683**, 789
GetPixelMap, **195**, 767

GetPointerv, 681, **710**, 732–734, 807, 815, 854, 888
GetPolygonStipple, **572**, 651, 743, 876
GetProgramInterfaceiv, 779
GetProgramResourceiv, 780, 781
GetProgramStageiv, 777
GetProgramBinary, **129**, 130–132, 771, 883, 900
GetProgramInfoLog, 97, 131, **170**, 170, 771, 881
GetProgramInterfaceiv, **106**, 109
GetProgramiv, 95, 130, 131, **166**, 166, 170, 465, 469, 489, 491, 495, 669, 771–774, 776, 778, 891
GetProgramPipelineInfoLog, **170**, 171, 881
GetProgramPipelineiv, **169**, 170, 466, 770
GetProgramPiplineInfoLog, 770
GetProgramResourceIndex, **108**
GetProgramResourceiv, **109**, 110, 111, 137–139, 144, 150, 595, 871, 883, 890, 898, 902
GetProgramResourceLocation, **119**, 119, 120
GetProgramResourceLocationIndex, **119**, 119, 120
GetProgramResourceName, **108**
GetProgramStageiv, **174**
GetQueryBufferObjecti64v, **50**
GetQueryBufferObjectiv, **50**
GetQueryBufferObjectui64v, **50**
GetQueryBufferObjectuiv, **50**
GetQueryIndexediv, **48**, 905
GetQueryiv, **50**, 809, 815
GetQueryObjecti64v, **50**
GetQueryObjectiv, **50**, 783
GetQueryObjectui64v, **50**
GetQueryObjectuiv, 21, **50**, 783
GetRenderbufferParameteriv, 761
GetRenderbufferParameteriv, **355**, 355, 356, 372
GetSamplerParameter, **189**
GetSamplerParameterfv, 751
GetSamplerParameterI{i ui}v, **189**
GetSamplerParameterIiv, 190
GetSamplerParameterIuiv, 14, 190
GetSamplerParameteriv, 751
GetSeparableFilter, 765
GetSeparableFilter, **207**, 651
GetShaderInfoLog, 89, **170**, 170, 769, 881
GetShaderiv, 89, 90, **165**, 170, 171, 769
GetShaderPrecisionFormat, 90, **172**
GetShaderSource, **171**, 769
GetString, **711**, 711, 712, 796, 844, 888, 910
GetStringi, **712**, 796, 910
GetSubroutineIndex, **154**
GetSubroutineUniformLocation, **155**, 859
GetSynciv, 21, 38, **42**, 42, 789, 898
GetTexEnv, **604**
GetTexEnvfv, 752, 753
GetTexEnviv, 752, 753
GetTexGen, **510**
GetTexGenfv, 752
GetTexGeniv, 752
GetTexImage, 162, 200, 201, 207, 210, 212, **282**, 283, 285, 287, 324, 332, 334, 651, 718, 746
GetTexLevelParameter, **279**
GetTexLevelParameteriv, 749, 750
GetTexParameter, **278**, 312, 334, 372, 719
GetTexParameterfv, 184, 324, 747, 748
GetTexParameterI, **278**
GetTexParameteriv, 184, 324, 747, 748
GetTextureImage, **282**, 282–287, 906
GetTextureLevelParameter, **279**

GetTextureLevelParameter*, 281
GetTextureParameter, **278**
GetTextureParameter*, 279
GetTextureParameterI, **278**
GetTextureParameteriv, 747
GetTextureSubImage, **285**, 288, 289, 906
GetTransformFeedbackVarying, 773
GetTransformFeedback*, 725
GetTransformFeedbacki64_v, **725**, 725
GetTransformFeedbacki_v, **725**, 725
GetTransformFeedbackiv, **725**, 725
GetTransformFeedbackVarying, **453**, 453, 876
GetUniform, 772
GetUniformBlockIndex, **137**
GetUniformdv, **173**
GetUniformfv, **172**
GetUniformIndices, **135**, 871
GetUniformiv, **173**
GetUniformLocation, **135**, 157, 158, 772
GetUniformSubroutineuiv, **173**
GetUniformuiv, **173**
GetVertexAttribPointerv, 734
GetVertexArrayIndexed64iv, **429**, 429
GetVertexArrayIndexediv, **428**, 429, 899
GetVertexArrayiv, **428**
GetVertexAttribdv, **430**, 430
GetVertexAttribfv, **430**, 430, 782
GetVertexAttribIiv, **430**, 431
GetVertexAttribIuiv, **430**, 431
GetVertexAttribiv, **430**, 430, 734, 735
GetVertexAttribLdv, **430**, 431
GetVertexAttribPointerv, **431**
gl_, 106
GL_APPLE_flush_buffer_range, 918
GL_APPLE_vertex_array_object, 918
GL_ARB_arrays_of_arrays, 929
GL_ARB_base_instance, 927
GL_ARB_bindless_texture, 934
GL_ARB_blend_func_extended, 922
GL_ARB_buffer_storage, 878
GL_ARB_buffer_storage, 932
GL_ARB_cl_event, 926
GL_ARB_clear_buffer_object, 929
GL_ARB_clear_texture, 878
GL_ARB_clear_texture, 933
GL_ARB_clip_control, 880, 894, 935
GL_ARB_color_buffer_float, 915
GL_ARB_compatibility, 839, 919
GL_ARB_compressed_texture_pixel_storage, 927
GL_ARB_compute_shader, 929
GL_ARB_compute_variable_group_size, 934
GL_ARB_conditional_render_inverted, 881, 894, 935
GL_ARB_conservative_depth, 927
GL_ARB_copy_buffer, 919
GL_ARB_copy_image, 929
GL_ARB_cull_distance, 880, 894, 936
GL_ARB_debug_group, 928
GL_ARB_debug_label, 929
GL_ARB_debug_output, 926, 928
GL_ARB_debug_output2, 928
GL_ARB_depth_buffer_float, 916
GL_ARB_depth_clamp, 919
GL_ARB_depth_texture, 912
GL_ARB_derivative_control, 881, 894, 936
GL_ARB_direct_state_access, 881, 888, 894, 936
GL_ARB_draw_buffers, 915
GL_ARB_draw_buffers_blend, 921
GL_ARB_draw_elements_base_vertex, 920
GL_ARB_draw_indirect, 924
GL_ARB_draw_instanced, 916, 917

GL_ARB_enhanced_layouts, 878
GL_ARB_enhanced_layouts, 933
GL_ARB_ES2_compatibility, 925
GL_ARB_ES3_1_compatibility, 881, 894, 935
GL_ARB_ES3_2_compatibility, 938
GL_ARB_ES3_compatibility, 930
GL_ARB_explicit_attrib_location, 922
GL_ARB_explicit_uniform_location, 930
GL_ARB_fragment_coord_conventions, 920
GL_ARB_fragment_layer_viewport, 930
GL_ARB_fragment_program, 913, 914
GL_ARB_fragment_program_shadow, 915
GL_ARB_fragment_shader, 914
GL_ARB_fragment_shader_interlock, 939
GL_ARB_framebuffer_no_attachments, 930
GL_ARB_framebuffer_object, 32, 916
GL_ARB_framebuffer_sRGB, 917
GL_ARB_geometry_shader4, 917
GL_ARB_get_program_binary, 925
GL_ARB_get_texture_sub_image, 881, 894, 936
GL_ARB_gl_spirv, 897, 941
GL_ARB_gpu_shader5, 924
GL_ARB_gpu_shader_fp64, 924
GL_ARB_gpu_shader_int64, 939
GL_ARB_half_float_pixel, 916
GL_ARB_half_float_vertex, 917
GL_ARB_imaging, 192
GL_ARB_indirect_parameters, 896
GL_ARB_indirect_parameters, 934
GL_ARB_instanced_arrays, 917
GL_ARB_internalformat_query, 927
GL_ARB_internalformat_query2, 930
GL_ARB_invalidate_subdata, 931
GL_ARB_map_buffer_alignment, 928
GL_ARB_map_buffer_range, 917
GL_ARB_matrix_palette, 912
GL_ARB_multi_bind, 878
GL_ARB_multi_draw_indirect, 931
GL_ARB_multi_bind, 933
GL_ARB_multisample, 911
GL_ARB_multitexture, 911
GL_ARB_occlusion_query, 914
GL_ARB_occlusion_query2, 923
GL_ARB_parallel_shader_compile, 939
GL_ARB_pipeline_statistics_query, 896, 937
GL_ARB_pixel_buffer_object, 916
GL_ARB_point_parameters, 911
GL_ARB_point_sprite, 914
GL_ARB_polygon_offset_clamp, 897
GL_ARB_post_depth_coverage, 939
GL_ARB_program_interface_query, 931
GL_ARB_provoking_vertex, 920
GL_ARB_query_buffer_object, 878
GL_ARB_query_buffer_object, 933
GL_ARB_robust_buffer_access_behavior, 931
GL_ARB_robustness, 894, 926
GL_ARB_robustness_isolation, 930, 932
GL_ARB_sample_locations, 939
GL_ARB_sample_shading, 921
GL_ARB_sampler_objects, 923
GL_ARB_seamless_cube_map, 920
GL_ARB_seamless_cubemap_per_texture, 934
GL_ARB_separate_shader_objects, 925
GL_ARB_shader_atomic_counter_ops, 897, 940
GL_ARB_shader_atomic_counters, 928
GL_ARB_shader_ballot, 940
GL_ARB_shader_bit_encoding, 923
GL_ARB_shader_clock, 940

GL_ARB_shader_draw_parameters, 897
GL_ARB_shader_group_vote, 897
GL_ARB_shader_image_load_store, 928
GL_ARB_shader_image_size, 931
GL_ARB_shader_objects, 914
GL_ARB_shader_precision, 925
GL_ARB_shader_stencil_export, 926
GL_ARB_shader_storage_buffer_object, 931
GL_ARB_shader_subroutine, 924
GL_ARB_shader_texture_image_samples, 881, 894, 936
GL_ARB_shader_texture_lod, 919
GL_ARB_shader_viewport_layer_array, 940
GL_ARB_shader_draw_parameters, 934
GL_ARB_shader_group_vote, 935
GL_ARB_shading_language_100, 914
GL_ARB_shading_language_420pack, 927
GL_ARB_shading_language_include, 922
GL_ARB_shading_language_packing, 928
GL_ARB_shadow, 913, 914
GL_ARB_shadow_ambient, 913
GL_ARB_sparse_buffer, 938
GL_ARB_sparse_texture2, 940
GL_ARB_sparse_texture_clamp, 940
GL_ARB_sparse_texture, 935
GL_ARB_spirv_extensions, 897
GL_ARB_stencil_texturing, 931
GL_ARB_sync, 920
GL_ARB_tessellation_shader, 924
GL_ARB_texture_barrier, 881, 894, 937
GL_ARB_texture_border_clamp, 911
GL_ARB_texture_buffer_object, 918
GL_ARB_texture_buffer_object_rgb32, 924
GL_ARB_texture_buffer_range, 932
GL_ARB_texture_compression, 911
GL_ARB_texture_compression_bptc, 922
GL_ARB_texture_compression_rgtc, 918
GL_ARB_texture_cube_map, 911
GL_ARB_texture_cube_map_array, 921
GL_ARB_texture_env_add, 911
GL_ARB_texture_env_combine, 912
GL_ARB_texture_env_crossbar, 912
GL_ARB_texture_env_dot3, 912
GL_ARB_texture_filter_anisotropic, 897
GL_ARB_texture_filter_minmax, 941
GL_ARB_texture_float, 916
GL_ARB_texture_gather, 921
GL_ARB_texture_mirror_clamp_to_edge, 878
GL_ARB_texture_mirrored_repeat, 912
GL_ARB_texture_multisample, 920
GL_ARB_texture_non_power_of_two, 914
GL_ARB_texture_query_levels, 932
GL_ARB_texture_query_lod, 922
GL_ARB_texture_rectangle, 915
GL_ARB_texture_rg, 918
GL_ARB_texture_rgb10_a2ui, 923
GL_ARB_texture_stencil8, 878
GL_ARB_texture_storage, 928
GL_ARB_texture_storage_multisample, 932
GL_ARB_texture_swizzle, 923
GL_ARB_texture_view, 929
GL_ARB_texture_mirror_clamp_to_edge, 933
GL_ARB_texture_stencil8, 933
GL_ARB_timer_query, 923
GL_ARB_transform_feedback2, 924
GL_ARB_transform_feedback3, 925

GL_ARB_transform_feedback_instanced, 927
GL_ARB_transform_feedback_overflow_query, 897, 938
GL_ARB_transpose_matrix, 911
GL_ARB_uniform_buffer_object, 919
GL_ARB_vertex_array_bgra, 921
GL_ARB_vertex_array_object, 918
GL_ARB_vertex_attrib_64bit, 925
GL_ARB_vertex_attrib_binding, 929
GL_ARB_vertex_blend, 912
GL_ARB_vertex_buffer_object, 913
GL_ARB_vertex_program, 913
GL_ARB_vertex_shader, 914
GL_ARB_vertex_type_10f_11f_11f_rev, 878
GL_ARB_vertex_type_2_10_10_10_rev, 923
GL_ARB_vertex_type_10f_11f_11f_rev, 933
GL_ARB_viewport_array, 925
GL_ARB_window_pos, 913
GL_ARB_*name*, 910
gl_BackColor, 127, 462, 471, 472, 485, 487, 493, 495, 515, 536, 841
gl_BackSecondaryColor, 127, 462, 471, 472, 485, 487, 493, 495, 515, 537, 841
gl_BaseInstance, 445, 461
gl_BaseVertex, 445, 461, 462
gl_ClipDistance, 463, 470, 472, 485, 487, 496, 935
gl_ClipDistance[], 125, 493, 539
gl_ClipVertex, 463, 471, 472, 485, 487, 493, 496, 539
gl_Color, 127, 590
gl_CullDistance, 463, 470, 472, 485, 487, 496, 538, 935
gl_CullDistance[], 125, 493, 538
gl_DrawID, 417, 421, 424, 425, 445, 461, 462, 906
GL_EXT_framebuffer_blit, 917
GL_EXT_framebuffer_multisample, 917
GL_EXT_framebuffer_object, 32, 917
GL_EXT_texture_compression_rgtc, 918
gl_FogFragCoord, 127, 463, 471, 472, 485, 487, 494, 495, 577
gl_FragColor, 592, 593, 632
gl_FragCoord, 567, 589, 900
gl_FragCoord.z, 819
gl_FragData, 632, 633
gl_FragData[n], 592, 593
gl_FragDepth, 592–594, 819
gl_FrontColor, 127, 462, 471, 472, 485, 487, 493, 495, 515, 536
gl_FrontFacing, 590
gl_FrontSecondaryColor, 127, 462, 471, 472, 485, 487, 493, 495, 515, 536
gl_in, 470, 471, 485
gl_in[], 493
gl_InstanceID, 414, 419, 445, 461, 462, 861
gl_InvocationID, 469, 471, 472, 492
GL_KHR_blend_equation_advanced, 938
GL_KHR_blend_equation_advanced_coherent, 938
GL_KHR_context_flush_control, 881, 937
GL_KHR_debug, 929
GL_KHR_no_error, 897, 938, 941
GL_KHR_robust_buffer_access_behavior, 937
GL_KHR_robustness, 881, 937
GL_KHR_texture_compression_astc_ldr, 928, 941

GL_KHR_texture_compression_astc_sliced_3d, 941
GL_KHR_*name*, 910
gl_Layer, 373, 374, 496, 497, 794, 940
gl_MaxPatchVertices, 470, 471, 485, 486
gl_NextBuffer, 105, 108, 114, 450
gl_NumSamples, 591
gl_NumWorkGroups, 668
GL_NV_depth_buffer_float, 916
GL_NV_half_float, 917
GL_OES_sample_shading, 888
GL_OES_sample_variables, 888
gl_out, 472
gl_PatchVerticesIn, 471, 485
gl_PerFragment, 126–128
gl_PerVertex, 124, 126–128
gl_PointCoord, 560
gl_PointSize, 124, 126, 463, 470, 472, 485, 487, 494, 496, 555
gl_Position, 448, 463, 470, 472, 485, 487, 494, 496, 543, 826
gl_PrimitiveID, 471, 485, 496, 590, 591, 826
gl_PrimitiveIDIn, 494
gl_SampleID, 591, 592
gl_SampleMask, 592, 593, 885, 886
gl_SampleMaskIn, 591, 592, 885, 888, 904
gl_SampleMaskIn[], 939
gl_SamplePosition, 591, 592
gl_SecondaryColor, 127, 590
gl_SkipComponents, 531
gl_SkipComponents1, 105, 108, 114, 450
gl_SkipComponents2, 105, 108, 114, 450
gl_SkipComponents3, 105, 108, 114, 450
gl_SkipComponents4, 105, 108, 114, 450
gl_TessCoord, 475, 485, 486, 821, 893
gl_TessLevelInner, 472, 473, 486, 487
gl_TessLevelInner[1], 486
gl_TessLevelOuter, 472, 473, 486, 487
gl_TessLevelOuter[2], 486
gl_TessLevelOuter[3], 486
gl_TexCoord, 462, 471, 472, 485, 487
gl_TexCoord[], 125, 127, 494, 495
gl_Vertex, 444
gl_VertexID, 414, 418, 445, 461, 462, 591, 826, 861, 892
gl_VerticesOut, 847
gl_ViewportIndex, 496, 497, 543, 544, 794, 940
gl_MaxPatchVertices, 887
glBlitFramebuffer, 17
GlobalInvocationId, 832
GLSLPacked, 146
GLSLShared, 146
glSpecializeShader, 902
GLX_ARB_create_context, 671, 918
GLX_ARB_create_context_no_error, 941
GLX_ARB_create_context_profile, 839, 922
GLX_ARB_create_context_robustness, 926
GLX_ARB_fbconfig_float, 915
GLX_ARB_framebuffer_sRGB, 917
GLX_ARB_robustness_application_isolation, 932
GLX_ARB_robustness_share_group_isolation, 932
GREATER, 275, 325, 614, 616, 617
GREEN, 193, 200, 217, 275, 323, 599, 653, 657, 747, 749, 759, 763, 764, 766, 814
GREEN_BIAS, 227
GREEN_BITS, 372, 843

GREEN_INTEGER, 217
GREEN_SCALE, 227
GUILTY_CONTEXT_RESET, 20

HALF_FLOAT, 215, 335, 398, 435, 654–656
HelperInvocation, 832
HIGH_FLOAT, 172
HIGH_INT, 172
Hint, **703**, 844
HINT_BIT, 706
HISTOGRAM, 209–211, 235, 766
Histogram, **209**, 209, 211, 235, 703
HISTOGRAM_x_SIZE, 766
HISTOGRAM_ALPHA_SIZE, 211
HISTOGRAM_BLUE_SIZE, 211
HISTOGRAM_FORMAT, 211, 766
HISTOGRAM_GREEN_SIZE, 211
HISTOGRAM_LUMINANCE_SIZE, 211
HISTOGRAM_RED_SIZE, 211
HISTOGRAM_SINK, 211, 766
HISTOGRAM_WIDTH, 211, 766

if, 101, 473
iimage1D, 117
iimage1DArray, 118
iimage2D, 117
iimage2DArray, 118
iimage2DMS, 118
iimage2DMSArray, 118
iimage2DRect, 118
iimage3D, 118
iimageBuffer, 118
iimageCube, 118
iimageCubeArray, 118
Image, 828, 829
Image1D, 833
image1D, 117
image1DArray, 117
image2D, 117
image2DArray, 117
image2DMS, 117
image2DMSArray, 117
image2DRect, 117
image3D, 117
IMAGE_1D, 117
IMAGE_1D_ARRAY, 117
IMAGE_2D, 117
IMAGE_2D_ARRAY, 117
IMAGE_2D_MULTISAMPLE, 117
IMAGE_2D_MULTISAMPLE_ARRAY, 117
IMAGE_2D_RECT, 117
IMAGE_3D, 117
IMAGE_BINDING_ACCESS, 784
IMAGE_BINDING_FORMAT, 784
IMAGE_BINDING_LAYER, 784
IMAGE_BINDING_LAYERED, 784
IMAGE_BINDING_LEVEL, 784
IMAGE_BINDING_NAME, 784
IMAGE_BUFFER, 117
IMAGE_CLASS_10_10_10_2, 719
IMAGE_CLASS_11_11_10, 719
IMAGE_CLASS_1_X_16, 719
IMAGE_CLASS_1_X_32, 719
IMAGE_CLASS_1_X_8, 719
IMAGE_CLASS_2_X_16, 719
IMAGE_CLASS_2_X_32, 718
IMAGE_CLASS_2_X_8, 719
IMAGE_CLASS_4_X_16, 719
IMAGE_CLASS_4_X_32, 718
IMAGE_CLASS_4_X_8, 719
IMAGE_COMPATIBILITY_CLASS, 718
IMAGE_CUBE, 117
IMAGE_CUBE_MAP_ARRAY, 117
IMAGE_FORMAT_COMPATIBILITY_BY_CLASS, 334, 719

IMAGE_FORMAT_COMPATIBILITY_BY_SIZE, 334, 719
IMAGE_FORMAT_COMPATIBILITY_TYPE, 279, 334, 719, 748
IMAGE_PIXEL_FORMAT, 719
IMAGE_PIXEL_TYPE, 719
IMAGE_TEXEL_SIZE, 719
ImageBuffer, 833
imageBuffer, 117
imageCube, 117
ImageCubeArray, 833
imageCubeArray, 117
ImageGatherExtended, 833
ImageMSArray, 833
ImageQuery, 833
ImageRect, 833
IMPLEMENTATION_COLOR_READ_FORMAT, 346, 371, 651, 652, 813, 861
IMPLEMENTATION_COLOR_READ_TYPE, 346, 371, 651, 652, 813, 861
in, 486
INCR, 616
INCR_WRAP, 616
INDEX, 348, 814
Index, **395**, 436, 595, 597
index, 595
INDEX_ARRAY, 407, 426, 733
INDEX_ARRAY_BUFFER_BINDING, 735
INDEX_ARRAY_POINTER, 710, 733
INDEX_ARRAY_STRIDE, 733
INDEX_ARRAY_TYPE, 733
INDEX_CLEAR_VALUE, 756
INDEX_LOGIC_OP, 626, 755
INDEX_MODE, 814
INDEX_OFFSET, 193, 195, 227, 763
INDEX_SHIFT, 193, 195, 227, 763
INDEX_WRITEMASK, 756
IndexMask, **634**
IndexPointer, **405**, 405, 406, 436, 701, 840
INFO_LOG_LENGTH, 166, 167, 169, 170, 769–771
InitNames, **692**, 692, 844
INNOCENT_CONTEXT_RESET, 20
Input, 128, 447, 828, 829
InstanceId, 832
INT, 115, 201, 215, 281, 332, 335, 348, 398, 656, 657, 700, 720
int, 115, 126, 144, 443
Int64Atomics, 830
INT_2_10_10_10_REV, 390, 392–394, 398–400, 410, 433
INT_IMAGE_1D, 117
INT_IMAGE_1D_ARRAY, 118
INT_IMAGE_2D, 117
INT_IMAGE_2D_ARRAY, 118
INT_IMAGE_2D_MULTISAMPLE, 118
INT_IMAGE_2D_MULTISAMPLE_ARRAY, 118
INT_IMAGE_2D_RECT, 118
INT_IMAGE_3D, 118
INT_IMAGE_BUFFER, 118
INT_IMAGE_CUBE, 118
INT_IMAGE_CUBE_MAP_ARRAY, 118
INT_SAMPLER_1D, 116
INT_SAMPLER_1D_ARRAY, 116
INT_SAMPLER_2D, 116
INT_SAMPLER_2D_ARRAY, 116
INT_SAMPLER_2D_MULTISAMPLE, 116
INT_SAMPLER_2D_MULTISAMPLE_ARRAY, 116
INT_SAMPLER_2D_RECT, 117
INT_SAMPLER_3D, 116

INT_SAMPLER_BUFFER, 117
INT_SAMPLER_CUBE, 116
INT_SAMPLER_CUBE_MAP_ARRAY, 116
INT_VEC2, 115
INT_VEC3, 115
INT_VEC4, 115
INTENSITY, 212, 229, 230, 238, 244, 246, 275, 285, 289, 325, 600, 601, 749, 764, 842
INTENSITY12, 244
INTENSITY16, 244
INTENSITY4, 244
INTENSITY8, 244
INTERLEAVED_ATTRIBS, 167, 176, 450, 451, 773
InterleavedArrays, **426**, 426, 427, 436, 701, 841
INTERNALFORMAT_ALPHA_SIZE, 719
INTERNALFORMAT_ALPHA_TYPE, 720
INTERNALFORMAT_BLUE_SIZE, 719
INTERNALFORMAT_BLUE_TYPE, 720
INTERNALFORMAT_DEPTH_SIZE, 720
INTERNALFORMAT_DEPTH_TYPE, 720
INTERNALFORMAT_GREEN_SIZE, 719
INTERNALFORMAT_GREEN_TYPE, 720
INTERNALFORMAT_PREFERRED, 719
INTERNALFORMAT_RED_SIZE, 719
INTERNALFORMAT_RED_TYPE, 720
INTERNALFORMAT_SHARED_SIZE, 720
INTERNALFORMAT_STENCIL_SIZE, 720
INTERNALFORMAT_STENCIL_TYPE, 720
INTERNALFORMAT_SUPPORTED, 714, 720
INTERPOLATE, 602
interpolateAtOffset, 810
interpolateAtCentroid, 588
interpolateAtOffset, 588
interpolateAtSample, 588
InterpolationFunction, 833
INVALID_ENUM, 17, 18, 38, 42, 45–47, 49, 51, 52, 62, 63, 65, 70, 72, 75, 77, 78, 81–83, 88, 91, 99, 100, 107–109, 119, 120, 132, 156, 166, 168, 169, 172, 174, 179, 181, 188, 190–193, 195, 197–200, 202, 207, 208, 210, 212, 213, 248, 249, 253, 259, 262, 265, 267, 269–271, 277, 279, 281–284, 304, 313, 314, 316, 317, 324, 340, 344, 346, 349–351, 354, 356–360, 370, 388, 392, 400, 407, 414, 419, 428, 429, 431, 433, 439, 451, 477, 509, 518, 527, 528, 537, 542, 553, 556, 569, 570, 604, 620, 622, 630, 633, 639–641, 650, 652, 663, 665, 676–679, 683, 685, 686, 692, 693, 695, 697, 704, 709–711, 713, 724, 725, 869, 886, 891, 893, 905
INVALID_FRAMEBUFFER_OPERATION, 18, 199, 206, 260, 371, 433, 581, 644, 653, 658, 660, 663
INVALID_INDEX, 105, 108, 174

INVALID_OPERATION, 17, 18, 46, 47, 51, 52, 62, 65, 68, 70–72, 75, 77–83, 89–94, 96, 98, 99, 107–109, 119, 120, 122–124, 130, 132, 141, 143, 148, 152, 156, 157, 166, 168–171, 173, 174, 179, 181, 182, 187, 188, 190, 192, 195, 196, 202, 207, 208, 210, 212, 213, 216, 219, 239, 249, 250, 254, 259, 260, 262, 265, 267–271, 277–279, 281–284, 286–289, 304, 310, 311, 313, 315–321, 330, 340, 341, 344, 346, 349–351, 354, 356–362, 370, 397, 398, 400–404, 408, 416, 418, 423, 425, 428, 429, 431, 436, 439, 445, 446, 450, 451, 463, 465, 466, 468, 489, 497, 505, 526–534, 572, 594–596, 604, 606, 624, 630, 633, 639–641, 643, 644, 648, 650–653, 655, 657, 658, 660, 663, 665, 666, 669, 670, 676, 685, 686, 692, 693, 695, 699, 702, 725, 839–841, 856, 859, 860, 884, 886, 900, 901, 905, 906
INVALID_VALUE, 17, 18, 38–40, 42, 45–47, 49, 51, 59, 60, 63, 65, 68, 70, 72, 75, 77, 80–82, 85, 89–91, 93, 94, 96, 97, 99, 100, 107–109, 119–124, 130, 132, 141, 142, 148, 152, 156, 163, 165, 166, 168, 170–174, 180, 182–187, 189, 191, 194–197, 199, 202, 204–210, 212, 248, 249, 254, 260, 262, 267, 269, 271, 277, 282, 284, 286–289, 310, 311, 313, 318–320, 329, 330, 341–344, 352–354, 359, 360, 362, 388, 392, 396, 397, 400–405, 407–409, 414, 416–419, 422–426, 428, 429, 431, 436, 439, 445, 446, 450, 451, 465, 504, 518, 525–527, 530, 534, 544–546, 553, 555, 556, 562, 572, 581, 583, 585, 594, 609, 619, 620, 622, 628, 633, 635, 636, 639, 640, 644, 651, 658, 663, 666, 669, 670, 676–678, 680, 682, 683, 685, 686, 688, 689, 692, 693, 695, 697, 699–701, 709, 710, 713, 724, 725, 839, 841, 843, 856, 857, 859, 872, 874, 886, 891, 893, 906
InvalidateBufferData, **80**
InvalidateBufferSubData, **79**, 80
InvalidateFramebuffer, **641**
InvalidateNamedFramebufferData, 640, **641**
InvalidateNamedFramebufferSubData, **640**, 884
InvalidateSubFramebuffer, **640**, 640, 856, 884, 889
InvalidateTexImage, **319**
InvalidateTexSubImage, **318**, 318, 319, 906
inversesqrt(), 834
INVERT, 616, 627
Invocation, 828
InvocationId, 832
IS_PER_PATCH, 110, 112, 780
IS_ROW_MAJOR, 110, 112, 137, 780
isampler1D, 116
isampler1DArray, 116
isampler2D, 116
isampler2DArray, 116
isampler2DMS, 116
isampler2DMSArray, 116

isampler2DRect, 117
isampler3D, 116
isamplerBuffer, 117
isamplerCube, 116
isamplerCubeArray, 116
IsBuffer, **60**, 60
IsEnabled, 583, 625, **709**, 709, 727, 732–734, 736, 738–745, 752, 754, 755, 764–768, 782, 807, 815
IsEnabledi, 583, 619, 625, **710**, 710, 754, 755
IsFramebuffer, **343**, 343
IsList, **701**, 844
ISOLINES, 168
isolines, 474, 476, 482, 485
IsProgram, **99**, 99
IsProgramPipeline, **121**, 121
IsQuery, **48**, 48
IsRenderbuffer, **353**, 353
IsSampler, 185, **189**, 189
IsShader, **90**, 90
IsSync, **42**, 42
IsTexture, **183**, 183, 847
IsTransformFeedback, **526**, 526
IsVertexArray, **397**, 397
ivec2, 115, 443
ivec3, 115, 443
ivec4, 115, 332, 443

KEEP, 616, 754
KHR_debug, 854
KHR_debug_output, 862
KHR_no_error, 907

LAST_VERTEX_CONVENTION, 497, 536, 537, 739
Layer, 832
LAYER_PROVOKING_VERTEX, 497, 794
Layered images, 345
layout, 105, 106, 113, 114, 125, 126, 143, 145, 146, 148, 150, 151, 168, 332, 334, 442, 443, 449–453, 468, 469, 472, 474, 476, 479, 482, 489, 491, 492, 590, 595, 597, 669, 884
LEFT, 619, 629, 631, 633, 634, 648
LEQUAL, 275, 323, 325, 614, 616, 617, 748, 751
LESS, 275, 325, 614, 616, 617, 754
Light, **516**, 516–518
LIGHTi, 741
LIGHT0, 518
LIGHT_MODEL_AMBIENT, 517, 740
LIGHT_MODEL_COLOR_CONTROL, 517, 740
LIGHT_MODEL_LOCAL_VIEWER, 517, 740
LIGHT_MODEL_TWO_SIDE, 517, 740
LIGHTING, 512, 740, 841
LIGHTING_BIT, 706
LightModel, **516**, 517
LINE, 573, 575, 576, 689, 690, 743
LINE_BIT, 706
LINE_LOOP, 381, 490, 497, 529
LINE_RESET_TOKEN, 698
LINE_SMOOTH, 562, 568, 742
LINE_SMOOTH_HINT, 704, 790
LINE_STIPPLE, 565, 742, 841
LINE_STIPPLE_PATTERN, 742
LINE_STIPPLE_REPEAT, 742
LINE_STRIP, 167, 381, 490, 497, 529, 689, 690
LINE_STRIP_ADJACENCY, 385, 490, 497
LINE_TOKEN, 698
LINE_WIDTH, 742
LINEAR, 254, 275, 277, 290, 297, 300,

301, 303, 305, 307, 322, 348, 349, 455, 608, 661, 663, 715, 718, 747, 751, 875
LINEAR_ATTENUATION, 517, 741
LINEAR_MIPMAP_LINEAR, 275, 301, 303
LINEAR_MIPMAP_NEAREST, 275, 301, 302
LINES, 167, 381, 490, 497, 527–529, 565
lines, 490
LINES_ADJACENCY, 167, 384, 490, 497
lines_adjacency, 490
LineStipple, **565**, 841
LineWidth, **562**, 839, 841
LINK_STATUS, 95, 130, 131, 166, 771, 891
LinkProgram, 92, **95**, 96–99, 102, 105, 125, 131, 132, 148, 157, 166, 444, 451, 463, 495, 532, 595, 702, 861, 891, 899
LIST_BASE, 815
LIST_BIT, 706
LIST_INDEX, 815
LIST_MODE, 815
ListBase, **700**, 703, 844
LOAD, 642
LoadIdentity, **502**, 841
LoadMatrix, **501**, 501, 502, 841
LoadName, **692**, 692, 844
LoadTransposeMatrix, **501**, 841
LocalInvocationId, 832
LocalInvocationIndex, 832
LocalSize, 830
LOCATION, 111, 113, 120, 595, 780
Location, 128, 441, 447, 595, 597
location, 106, 125, 126, 441, 447, 595, 884
LOCATION_COMPONENT, 111, 113, 595, 780
LOCATION_INDEX, 111, 113, 120, 595, 780
log(), 834
log2(), 834
LOGIC_OP, 626
LOGIC_OP_MODE, 755
Logical, 828
LogicOp, **627**, 627
LOSE_CONTEXT_ON_RESET, 20
LOW_FLOAT, 172
LOW_INT, 172
LOWER_LEFT, 537, 541, 543, 556, 560, 569, 738
LUMINANCE, 200, 217, 226, 229, 230, 238, 244, 246, 275, 285, 289, 323, 325, 600, 601, 653, 654, 657, 748, 749, 764, 766, 842
LUMINANCE12, 244
LUMINANCE12_ALPHA12, 244
LUMINANCE12_ALPHA4, 244
LUMINANCE16, 244
LUMINANCE16_ALPHA16, 244
LUMINANCE4, 244
LUMINANCE4_ALPHA4, 244
LUMINANCE6_ALPHA2, 244
LUMINANCE8, 244, 322, 749, 887
LUMINANCE8_ALPHA8, 244
LUMINANCE_ALPHA, 200, 217, 226, 229, 230, 238, 244, 246, 285, 600, 601, 653, 654, 657, 842

main, 473
MAJOR_VERSION, 711, 796
MANUAL_GENERATE_MIPMAP, 716
Map1, 15, **684**, 685, 686
MAP1_x, 768
MAP1_COLOR_4, 685
MAP1_GRID_DOMAIN, 768

MAP1_GRID_SEGMENTS, 768
MAP1_INDEX, 685
MAP1_NORMAL, 685
MAP1_TEXTURE_COORD_1, 685, 687
MAP1_TEXTURE_COORD_2, 685, 687
MAP1_TEXTURE_COORD_3, 685
MAP1_TEXTURE_COORD_4, 685
MAP1_VERTEX_3, 685
MAP1_VERTEX_4, 685
Map1f, 685
Map2, 15, **686**, 686
MAP2_x, 768
MAP2_GRID_DOMAIN, 768
MAP2_GRID_SEGMENTS, 768
MAP2_VERTEX_3, 687, 688
MAP2_VERTEX_4, 687, 688
Map2f, 685
MAP_COHERENT_BIT, 66–68, 73, 76, 870, 885, 889
MAP_COLOR, 193, 228, 763
MAP_FLUSH_EXPLICIT_BIT, 74, 76, 77
MAP_INVALIDATE_BUFFER_BIT, 74, 76
MAP_INVALIDATE_RANGE_BIT, 74, 76
MAP_PERSISTENT_BIT, 66–68, 71–73, 76, 77, 79–82
MAP_READ_BIT, 66–69, 73–76
MAP_STENCIL, 193, 229, 763
MAP_UNSYNCHRONIZED_BIT, 75, 76
MAP_WRITE_BIT, 66–69, 73–76, 871
MAP_COHERENT_BIT, 73
MapBuffer, 62, 71, 72, **76**, 76, 80, 533, 702
MapBufferRange, 66, 71, 72, **73**, 73, 75, 77, 80, 702, 870, 890
MapGrid1, **688**
MapGrid2, **688**
MapNamedBuffer, **76**, 76
MapNamedBufferRange, 66, **73**, 73, 75, 77, 889
mat$C$, 144
mat$C$xR, 144
mat2, 115, 443
mat2x3, 115, 443
mat2x4, 115, 443
mat3, 115, 443
mat3x2, 115, 443
mat3x4, 115, 443
mat4, 115, 443
mat4x2, 115, 443
mat4x3, 115, 443
Material, 436, **516**, 516–518, 522, 825
Matrix, 833
MATRIX_MODE, 504, 738
MATRIX_STRIDE, 110, 113, 137, 144, 780
MatrixMode, **500**, 841
MatrixStride, 146, 147
MAX, 621
MAX_3D_TEXTURE_SIZE, 248, 280, 360–362, 792
MAX_ARRAY_TEXTURE_LAYERS, 248, 269, 362, 792, 867
MAX_ATOMIC_COUNTER_BUFFER_BINDINGS, 84, 149, 804
MAX_ATOMIC_COUNTER_BUFFER_SIZE, 804, 849
MAX_ATTRIB_STACK_DEPTH, 705, 793, 844, 877
MAX_CLIENT_ATTRIB_STACK_DEPTH, 705, 793, 844, 877
MAX_CLIP_DISTANCES, 538, 792
MAX_COLOR_ATTACHMENTS, 340, 350, 357–360, 370, 630, 631,

641, 650, 810
MAX_COLOR_MATRIX_STACK_DEPTH, 209, 793
MAX_COLOR_TEXTURE_SAMPLES, 723, 809
MAX_COMBINED_ATOMIC_COUNTER_BUFFERS, 149, 464, 804, 867
MAX_COMBINED_ATOMIC_COUNTERS, 460, 464, 804
MAX_COMBINED_CLIP_AND_CULL_DISTANCES, 538, 792
MAX_COMBINED_COMPUTE_UNIFORM_COMPONENTS, 133, 802
MAX_COMBINED_DIMENSIONS, 721
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS, 133, 806
MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS, 133, 806
MAX_COMBINED_IMAGE_UNIFORMS, 461, 464, 805, 849
MAX_COMBINED_IMAGE_UNITS_AND_FRAGMENT_OUTPUTS, 855
MAX_COMBINED_SHADER_OUTPUT_RESOURCES, 336, 464, 805, 855
MAX_COMBINED_SHADER_STORAGE_BLOCKS, 151, 461, 464, 804
MAX_COMBINED_TESS_CONTROL_UNIFORM_COMPONENTS, 133, 806
MAX_COMBINED_TESS_EVALUATION_UNIFORM_COMPONENTS, 133, 806
MAX_COMBINED_TEXTURE_IMAGE_UNITS, 141, 179, 180, 187, 278, 458, 464, 803, 859
MAX_COMBINED_UNIFORM_BLOCKS, 143, 464, 803, 862
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS, 133, 806
MAX_COMPUTE_ATOMIC_COUNTER_BUFFERS, 149, 802
MAX_COMPUTE_ATOMIC_COUNTERS, 460, 802
MAX_COMPUTE_IMAGE_UNIFORMS, 461, 802
MAX_COMPUTE_SHADER_STORAGE_BLOCKS, 151, 461, 802
MAX_COMPUTE_SHARED_MEMORY_SIZE, 670, 802, 861
MAX_COMPUTE_TEXTURE_IMAGE_UNITS, 458, 802
MAX_COMPUTE_UNIFORM_BLOCKS, 143, 802, 862
MAX_COMPUTE_UNIFORM_COMPONENTS, 133, 802, 889
MAX_COMPUTE_WORK_GROUP_COUNT, 669, 802
MAX_COMPUTE_WORK_GROUP_INVOCATIONS, 669, 802
MAX_COMPUTE_WORK_GROUP_SIZE, 669, 802
MAX_CONVOLUTION_HEIGHT, 204, 208, 793
MAX_CONVOLUTION_WIDTH, 204, 205, 208, 793
MAX_CUBE_MAP_TEXTURE_SIZE, 248, 280, 311, 360, 362, 792

MAX_CULL_DISTANCES, 538, 792
MAX_DEBUG_GROUP_STACK_DEPTH, 678, 808
MAX_DEBUG_LOGGED_MESSAGES, 675, 808
MAX_DEBUG_MESSAGE_LENGTH, 674, 677, 678, 808
MAX_DEPTH, 721, 722
MAX_DEPTH_TEXTURE_SAMPLES, 723, 809
MAX_DRAW_BUFFERS, 594, 595, 619, 620, 622, 632, 633, 635, 639, 810
MAX_DUAL_SOURCE_DRAW_BUFFERS, 594, 595, 622, 624, 810
MAX_ELEMENT_INDEX, 419, 792
MAX_ELEMENTS_INDICES, 421, 794
MAX_ELEMENTS_VERTICES, 421, 794
MAX_EVAL_ORDER, 685, 686, 793
MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS, 149, 801
MAX_FRAGMENT_ATOMIC_COUNTERS, 460, 801
MAX_FRAGMENT_IMAGE_UNIFORMS, 461, 805, 849
MAX_FRAGMENT_INPUT_COMPONENTS, 592, 801
MAX_FRAGMENT_INTERPOLATION_OFFSET, 588, 810, 898
MAX_FRAGMENT_SHADER_STORAGE_BLOCKS, 151, 461, 801
MAX_FRAGMENT_UNIFORM_BLOCKS, 143, 801, 862
MAX_FRAGMENT_UNIFORM_COMPONENTS, 133, 801
MAX_FRAGMENT_UNIFORM_VECTORS, 133, 801
MAX_FRAMEBUFFER_HEIGHT, 344, 366, 809
MAX_FRAMEBUFFER_LAYERS, 344, 367, 809
MAX_FRAMEBUFFER_SAMPLES, 344, 367, 809
MAX_FRAMEBUFFER_WIDTH, 344, 366, 809
MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS, 149, 800
MAX_GEOMETRY_ATOMIC_COUNTERS, 460, 800
MAX_GEOMETRY_IMAGE_UNIFORMS, 460, 805
MAX_GEOMETRY_INPUT_COMPONENTS, 494, 800
MAX_GEOMETRY_OUTPUT_COMPONENTS, 496, 800
MAX_GEOMETRY_OUTPUT_VERTICES, 495, 800
MAX_GEOMETRY_SHADER_INVOCATIONS, 800
MAX_GEOMETRY_SHADER_STORAGE_BLOCKS, 151, 461, 800
MAX_GEOMETRY_TEXTURE_IMAGE_UNITS, 458, 800
MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS, 495, 800
MAX_GEOMETRY_UNIFORM_BLOCKS, 143, 800, 862
MAX_GEOMETRY_UNIFORM_COMPONENTS, 133, 800, 889

MAX_HEIGHT, 722
MAX_IMAGE_SAMPLES, 331, 805
MAX_IMAGE_UNITS, 141, 157, 327, 329, 805
MAX_INTEGER_SAMPLES, 355, 723, 809
MAX_LABEL_LENGTH, 680, 808
MAX_LAYERS, 722
MAX_LIGHTS, 518, 793
MAX_LIST_NESTING, 700, 793
MAX_MODELVIEW_STACK_DEPTH, 793
MAX_NAME_LENGTH, 106, 107, 109, 779
MAX_NAME_STACK_DEPTH, 793
MAX_NUM_ACTIVE_VARIABLES, 107, 779
MAX_NUM_COMPATIBLE_SUBROUTINES, 107, 779
MAX_PATCH_VERTICES, 388, 798
MAX_PIXEL_MAP_TABLE, 194, 195, 227, 793
MAX_PROGRAM_TEXEL_OFFSET, 294, 803
MAX_PROGRAM_TEXTURE_GATHER_OFFSET, 294, 300, 801
MAX_PROJECTION_STACK_DEPTH, 793
MAX_RECTANGLE_TEXTURE_SIZE, 248, 795
MAX_RENDERBUFFER_SIZE, 354, 792
MAX_SAMPLE_MASK_WORDS, 585, 809
MAX_SAMPLES, 355, 723, 809
MAX_SERVER_WAIT_TIMEOUT, 40, 809
MAX_SHADER_STORAGE_BLOCK_SIZE, 151, 804
MAX_SHADER_STORAGE_BUFFER_BINDINGS, 84, 152, 804
MAX_SUBROUTINE_UNIFORM_LOCATIONS, 153, 803
MAX_SUBROUTINES, 153, 803
MAX_TESS_CONTROL_ATOMIC_COUNTER_BUFFERS, 149, 798
MAX_TESS_CONTROL_ATOMIC_COUNTERS, 460, 798
MAX_TESS_CONTROL_IMAGE_UNIFORMS, 460, 805
MAX_TESS_CONTROL_INPUT_COMPONENTS, 471, 798
MAX_TESS_CONTROL_OUTPUT_COMPONENTS, 473, 798
MAX_TESS_CONTROL_SHADER_STORAGE_BLOCKS, 151, 461, 798
MAX_TESS_CONTROL_TEXTURE_IMAGE_UNITS, 458, 798
MAX_TESS_CONTROL_TOTAL_OUTPUT_COMPONENTS, 473, 798, 888
MAX_TESS_CONTROL_UNIFORM_BLOCKS, 143, 798, 861
MAX_TESS_CONTROL_UNIFORM_COMPONENTS, 133, 798
MAX_TESS_EVALUATION_ATOMIC_COUNTER_BUFFERS, 149, 799
MAX_TESS_EVALUATION_ATOMIC_COUNTERS, 460, 799
MAX_TESS_EVALUATION_IMAGE_UNIFORMS, 460, 805
MAX_TESS_EVALUATION_

INPUT_COMPONENTS, 487, 799
MAX_TESS_EVALUATION_OUTPUT_COMPONENTS, 487, 799
MAX_TESS_EVALUATION_SHADER_STORAGE_BLOCKS, 151, 461, 799
MAX_TESS_EVALUATION_TEXTURE_IMAGE_UNITS, 458, 799
MAX_TESS_EVALUATION_UNIFORM_BLOCKS, 143, 799, 862
MAX_TESS_EVALUATION_UNIFORM_COMPONENTS, 133, 799
MAX_TESS_GEN_LEVEL, 476, 798
MAX_TESS_PATCH_COMPONENTS, 473, 487, 798
MAX_TEXTURE_BUFFER_SIZE, 271, 795
MAX_TEXTURE_COORDS, 179, 180, 278, 393, 405, 407, 431, 505, 801, 843
MAX_TEXTURE_IMAGE_UNITS, 458, 801, 882
MAX_TEXTURE_LOD_BIAS, 293, 792
MAX_TEXTURE_MAX_ANISOTROPY, 291, 292, 792
MAX_TEXTURE_SIZE, 248, 269, 280, 360, 361, 792
MAX_TEXTURE_STACK_DEPTH, 793
MAX_TEXTURE_UNITS, 179, 378, 604, 606, 707, 801, 843
MAX_TRANSFORM_FEEDBACK_BUFFERS, 84, 451, 812
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS, 451, 452, 812
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS, 451, 812
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS, 451, 812
MAX_UNIFORM_BLOCK_SIZE, 143, 803, 888
MAX_UNIFORM_BUFFER_BINDINGS, 84, 147, 148, 803, 857
MAX_UNIFORM_LOCATIONS, 134, 803
MAX_VARYING_COMPONENTS, 448, 803, 840
MAX_VARYING_FLOATS, 840
MAX_VARYING_VECTORS, 448, 803
MAX_VERTEX_ATOMIC_COUNTER_BUFFERS, 149, 797
MAX_VERTEX_ATOMIC_COUNTERS, 460, 797
MAX_VERTEX_ATTRIB_BINDINGS, 401–403, 408, 429, 431, 795
MAX_VERTEX_ATTRIB_RELATIVE_OFFSET, 400, 795
MAX_VERTEX_ATTRIB_STRIDE, 401, 402, 404, 795, 866, 875
MAX_VERTEX_ATTRIBS, 389, 391, 392, 395, 400, 403, 405, 409, 429, 431, 444–446, 797
MAX_VERTEX_IMAGE_UNIFORMS, 460, 805
MAX_VERTEX_OUTPUT_COMPONENTS, 448, 471, 473, 487, 495, 496, 592, 797
MAX_VERTEX_SHADER_STORAGE_BLOCKS, 151, 461, 797

MAX_VERTEX_STREAMS, 43, 46, 47, 49, 534, 800
MAX_VERTEX_TEXTURE_IMAGE_UNITS, 458, 797
MAX_VERTEX_UNIFORM_BLOCKS, 143, 797, 861
MAX_VERTEX_UNIFORM_COMPONENTS, 133, 797
MAX_VERTEX_UNIFORM_VECTORS, 133, 797
MAX_VIEWPORT_DIMS, 546, 641, 794
MAX_VIEWPORTS, 543–546, 583, 584, 794
MAX_WIDTH, 722
MEDIUM_FLOAT, 172
MEDIUM_INT, 172
Memory Semantics, 829
MemoryBarrier, 66, **161**, 163–165, 702, 902
memoryBarrier, 160, 164
MemoryBarrierByRegion, **164**, 165
MIN, 621
MIN/MAG, 715
MIN_FRAGMENT_INTERPOLATION_OFFSET, 588, 810
MIN_MAP_BUFFER_ALIGNMENT, 74, 76, 795, 849
MIN_PROGRAM_TEXEL_OFFSET, 294, 803
MIN_PROGRAM_TEXTURE_GATHER_OFFSET, 294, 300, 801
MIN_SAMPLE_SHADING_VALUE, 554, 744
MINMAX, 211–213, 236, 767
Minmax, **211**, 236
MINMAX_FORMAT, 213, 767
MINMAX_SINK, 213, 767
MINOR_VERSION, 711, 796
MinSampleShading, **554**, 904
MIPMAP, 722
MIRROR_CLAMP_TO_EDGE, 186, 276, 277, 297, 875, 891
MIRRORED_REPEAT, 186, 276, 277, 297
MODELVIEW, 500, 501, 505
MODELVIEW_MATRIX, 506
MODELVIEW_MATRIX (TRANSPOSE_MODELVIEW_MATRIX), 738
MODELVIEW_STACK_DEPTH, 738
MODULATE, 599, 601, 602, 604, 752
MULT, 642
MultiDrawArrays, **416**, 860
MultiDrawArraysIndirect, 413, **417**, 418, 860, 892
MultiDrawArraysIndirectCount, **418**
MultiDrawElements, 412, 413, **421**, 860, 892
MultiDrawElementsBaseVertex, 413, **425**, 860
MultiDrawElementsIndirect, 413, **423**, 424, 425, 860, 892
MultiDrawElementsIndirectCount, **424**
MULTISAMPLE, 553, 554, 561, 568, 575, 581, 584, 588, 612, 628, 647, 744
MULTISAMPLE_BIT, 706
MultiTexCoord, 393, 436
MultiViewport, 833
MultMatrix, **501**, 501, 502, 841
MultTransposeMatrix, **501**, 841

N3F_V3F, 427
Name, 93
NAME_LENGTH, 111, 113, 137, 138, 781
NAME_STACK_DEPTH, 815

NamedBufferData, **68**, 68–70
NamedBufferStorage, **65**, 65, 67, 68
NamedBufferSubData, **70**, 70
NamedFramebufferDrawBuffer, **629**, 629, 630
NamedFramebufferDrawBuffers, **630**, 630, 633
NamedFramebufferParameteri, **343**, 343, 344
NamedFramebufferReadBuffer, **648**, 648, 650
NamedFramebufferRenderbuffer, **356**, 356, 357
NamedFramebufferTexture, **358**, 358, 359, 361
NamedFramebufferTextureLayer, **361**
NamedRenderbufferStorage, **354**
NamedRenderbufferStorageMultisample, **353**, 353, 354
NAND, 627
NEAREST, 275, 277, 290, 296, 297, 300, 301, 303, 305, 307, 325, 455, 661, 663, 715
NEAREST_MIPMAP_LINEAR, 275, 301, 303, 322, 751
NEAREST_MIPMAP_NEAREST, 275, 301, 302, 307, 325, 715, 888
NEGATIVE_ONE_TO_ONE, 537, 538, 541, 543, 738
NEVER, 275, 325, 614, 616, 617
NewList, **697**, 699, 700, 844
NICEST, 703, 704
NO_ERROR, 16, 19, 20
NO_RESET_NOTIFICATION, 20
NONE, 105, 260, 275, 281, 322, 323, 325, 347, 348, 350, 363, 366, 453, 459, 612, 624, 628, 630–634, 638, 639, 648, 650, 652, 653, 712, 715, 718–720, 722, 724, 747–749, 751, 759, 783, 856, 868, 876, 882, 901
NOOP, 627
NoPerspective, 829
noperspective, 541
NOR, 627
Normal, 393, 436
Normal3, **393**
NORMAL_ARRAY, 407, 428, 732
NORMAL_ARRAY_BUFFER_BINDING, 735
NORMAL_ARRAY_POINTER, 710, 732
NORMAL_ARRAY_STRIDE, 732
NORMAL_ARRAY_TYPE, 732
NORMAL_MAP, 508, 509
NORMALIZE, 506, 738, 841
NormalP3uiv, 393
NormalPointer, **405**, 405, 406, 428, 436, 701, 840
NOTEQUAL, 275, 325, 614, 616, 617
NULL, 674, 679, 682, 683, 734, 737, 807
NUM_ACTIVE_VARIABLES, 110, 112, 113, 138, 139, 781
NUM_COMPATIBLE_SUBROUTINES, 110, 119, 155, 777, 781, 902
NUM_COMPRESSED_TEXTURE_FORMATS, 239, 795, 840, 846
NUM_EXTENSIONS, 712, 796
NUM_PROGRAM_BINARY_FORMATS, 130, 131, 795
NUM_SAMPLE_COUNTS, 722, 811
NUM_SHADER_BINARY_FORMATS, 86, 91, 795
NUM_SHADING_LANGUAGE_VERSIONS, 713, 796, 857
NUM_SPIR_V_EXTENSIONS, 713,

796
NumWorkgroups, 832

OBJECT_LINEAR, 508–510
OBJECT_PLANE, 508, 509, 752
OBJECT_TYPE, 38, 42, 789
ObjectLabel, **679**, 680, 682
ObjectPtrLabel, **680**
OFFSET, 111, 113, 137, 781
Offset, 146, 147, 449, 453, 829, 830
ONE, 275, 600, 621, 623, 625, 755
ONE_MINUS_CONSTANT_ALPHA, 623
ONE_MINUS_CONSTANT_COLOR, 623
ONE_MINUS_DST_ALPHA, 623
ONE_MINUS_DST_COLOR, 623
ONE_MINUS_SRC1_ALPHA, 622–624
ONE_MINUS_SRC1_COLOR, 622–624
ONE_MINUS_SRC_ALPHA, 603, 623
ONE_MINUS_SRC_COLOR, 603, 623
Op*Name, 105
OpCapability, 827, 829, 833
OpEntryPoint, 93, 128, 447, 597
OPERAND0_ALPHA, 753
OPERAND0_RGB, 753
OPERAND1_ALPHA, 753
OPERAND1_RGB, 753
OPERAND2_ALPHA, 753
OPERAND2_RGB, 753
OPERAND*n*_ALPHA, 600, 603
OPERAND*n*_RGB, 600, 603
OpFAdd, 834
OpFDiv, 834
OpFMul, 834
OpFOrdEqual, 834
OpFOrdGreaterThan, 834
OpFOrdGreaterThanEqual, 834
OpFOrdLessThan, 834
OpFOrdLessThanEqual, 834
OpFSub, 834
OpFUnordEqual, 834
OpFUnordGreaterThan, 834
OpFUnordGreaterThanEqual, 834
OpFUnordLessThan, 834
OpFUnordLessThanEqual, 834
OpImageQueryLevels, 828, 904
OpImageQuerySizeLod, 828, 904
OpImageWrite, 829, 900
OpMemberName, 105
OpName, 105
OpSampledImage, 828
OpSMod, 831
OpSRem, 831
OpTypeBool, 830, 900
OpTypeFloat, 597
OpTypeImage, 828
OpTypeInt, 597
OpTypeRuntimeArray, 829, 905
OpTypeStruct, 829
OR, 627
OR_INVERTED, 627
OR_REVERSE, 627
ORDER, 692, 768
Ortho, 502, **504**, 825, 841
out, 472
OUT_OF_MEMORY, 16–19, 699, 867, 870, 905
Output, 128, 597, 828

PACK_ALIGNMENT, 651, 763
PACK_COMPRESSED_BLOCK_DEPTH, 287, 288, 651, 763
PACK_COMPRESSED_BLOCK_HEIGHT, 287, 288, 651, 763
PACK_COMPRESSED_BLOCK_SIZE, 287, 651, 763

PACK_COMPRESSED_BLOCK_-WIDTH, 287, 288, 651, 763
PACK_IMAGE_HEIGHT, 283, 287, 651, 763
PACK_LSB_FIRST, 651, 763, 840, 854, 862
PACK_ROW_LENGTH, 287, 651, 763
PACK_SKIP_IMAGES, 283, 287, 651, 763
PACK_SKIP_PIXELS, 287, 651, 763
PACK_SKIP_ROWS, 287, 651, 763
PACK_SWAP_BYTES, 651, 763
PARAMETER_BUFFER, 61, 162, 418, 424, 425, 902
PARAMETER_BUFFER_BINDING, 736
PASS_THROUGH_TOKEN, 698
PassThrough, **696**, 697, 844
Patch, 829
patch, 112, 469
patch in, 486
patch out, 472
PATCH_DE-FAULT_INNER_LEVEL, 474, 477, 731
PATCH_DEFAULT_OUTER_LEVEL, 474, 477, 731
PATCH_VERTICES, 387, 388, 731
PATCHES, 387, 410, 468, 792
PatchParameterfv, **474**
PatchParameteri, **387**, 484
PatchVertices, 832
PauseTransformFeedback, **528**, 529
PERSPECTIVE_CORRECTION_-HINT, 704, 790, 844
PIXEL_BUFFER_BARRIER_BIT, 162
PIXEL_MAP_A_TO_A, 194, 228
PIXEL_MAP_B_TO_B, 194, 228
PIXEL_MAP_G_TO_G, 194, 228
PIXEL_MAP_I_TO_A, 194, 228
PIXEL_MAP_I_TO_B, 194, 228
PIXEL_MAP_I_TO_G, 194, 228
PIXEL_MAP_I_TO_I, 194, 228
PIXEL_MAP_I_TO_R, 194, 228
PIXEL_MAP_R_TO_R, 194, 228
PIXEL_MAP_S_TO_S, 194, 229
PIXEL_MODE_BIT, 706
PIXEL_PACK_BUFFER, 61, 162, 191, 650
PIXEL_PACK_BUFFER_BINDING, 195, 283, 655, 763
PIXEL_UNPACK_BUFFER, 61, 162, 191
PIXEL_UNPACK_BUFFER_BIND-ING, 194, 216, 261, 762
PixelMap, **194**, 194, 195, 667
PixelStore, **190**, 436, 651, 667, 702, 874
PixelTransfer, **193**, 193, 233, 667
PixelZoom, **646**, 647, 842
POINT, 573, 575, 576, 689, 690, 743
POINT_BIT, 706
POINT_DISTANCE_ATTENUATION, 556, 742
POINT_FADE_THRESHOLD_SIZE, 556, 742
point_mode, 476
POINT_SIZE, 742
POINT_SIZE_GRANULARITY, 794
POINT_SIZE_MAX, 556, 742
POINT_SIZE_MIN, 556, 742
POINT_SIZE_RANGE, 794
POINT_SMOOTH, 556, 561, 742, 841
POINT_SMOOTH_HINT, 704, 790, 844
POINT_SPRITE, 556, 557, 561, 598, 599, 604, 742, 841
POINT_SPRITE_COORD_ORIGIN, 556, 560, 742
POINT_TOKEN, 698
PointCoord, 832

Pointer, 830
PointParameter, **556**
POINTS, 167, 380, 489, 490, 497, 527–529, 573, 689, 690
points, 489, 493
PointSize, **555**, 555, 832
POLYGON, 381, 388, 389, 437, 529, 591, 842
POLYGON_BIT, 706
POLYGON_MODE, 743
POLYGON_OFFSET_CLAMP, 575, 743
POLYGON_OFFSET_FACTOR, 743
POLYGON_OFFSET_FILL, 575, 743
POLYGON_OFFSET_LINE, 575, 743
POLYGON_OFFSET_POINT, 575, 743
POLYGON_OFFSET_UNITS, 743
POLYGON_SMOOTH, 569, 575, 743
POLYGON_SMOOTH_HINT, 704, 790
POLYGON_STIPPLE, 572, 743, 842
POLYGON_STIPPLE_BIT, 706
POLYGON_TOKEN, 698
PolygonMode, 568, **573**, 573, 576, 694, 695, 842
PolygonOffset, **574**, 574
PolygonOffsetClamp, **574**, 574
PolygonStipple, 191, **572**, 842
PopAttrib, 704, **705**, 705, 707, 727, 826, 844
PopClientAttrib, 436, 701, 704, **705**, 705, 707, 727, 844
PopDebugGroup, **678**, 678
PopMatrix, **505**, 841
PopName, **692**, 692, 844
POSITION, 517, 519, 741
Position, 832
POST_COLOR_MATRIX_x_BIAS, 766
POST_COLOR_MATRIX_x_SCALE, 766
POST_COLOR_MATRIX_ALPHA_BIAS, 234
POST_COLOR_MATRIX_ALPHA_SCALE, 234
POST_COLOR_MATRIX_BLUE_BIAS, 234
POST_COLOR_MATRIX_BLUE_SCALE, 234
POST_COLOR_MATRIX_COLOR_TABLE, 197, 235, 764
POST_COLOR_MATRIX_GREEN_BIAS, 234
POST_COLOR_MATRIX_GREEN_SCALE, 234
POST_COLOR_MATRIX_RED_BIAS, 234
POST_COLOR_MATRIX_RED_SCALE, 234
POST_CONVOLUTION_x_BIAS, 766
POST_CONVOLUTION_x_SCALE, 766
POST_CONVOLUTION_ALPHA_BIAS, 234
POST_CONVOLUTION_ALPHA_SCALE, 233
POST_CONVOLUTION_BLUE_BIAS, 234
POST_CONVOLUTION_BLUE_SCALE, 233
POST_CONVOLUTION_COLOR_TABLE, 197, 234, 764
POST_CONVOLUTION_GREEN_BIAS, 234
POST_CONVOLUTION_GREEN_SCALE, 233
POST_CONVOLUTION_RED_BIAS, 234
POST_CONVOLUTION_RED_SCALE, 233
pow(x,y), 834

PREVIOUS, 603, 604, 753
PRIMARY_COLOR, 603
PRIMITIVE_RESTART, 409, 410, 736
PRIMITIVE_RESTART_FIXED_INDEX, 410, 736, 877
PRIMITIVE_RESTART_FOR_PATCHES_SUPPORTED, 410, 792, 872
PRIMITIVE_RESTART_INDEX, 736
PrimitiveId, 832
PrimitiveRestartIndex, **409**, 410, 701
PRIMITIVES_GENERATED, 43, 46, 47, 49, 534
PRIMITIVES_SUBMITTED, 43, 49, 439
PrioritizeTextures, **184**, 184, 843
Private, 828
PROGRAM, 679
PROGRAM_BINARY_FORMATS, 131, 795
PROGRAM_BINARY_LENGTH, 130, 771
PROGRAM_BINARY_RETRIEVABLE_HINT, 98, 99, 131, 168, 771
PROGRAM_INPUT, 102, 105, 106, 110, 111, 119, 445, 446, 861
PROGRAM_OUTPUT, 102, 105, 106, 110, 111, 119, 595, 596, 861
PROGRAM_PIPELINE, 679
PROGRAM_PIPELINE_BINDING, 771
PROGRAM_POINT_SIZE, 494, 555, 782
PROGRAM_SEPARABLE, 98–100, 123, 131, 168, 464, 771, 848, 899
ProgramBinary, 97–99, **130**, 131, 132, 166, 532, 861, 866, 899
ProgramParameteri, **98**, 131
ProgramUniform, **142**
ProgramUniform{1234}ui, **142**
ProgramUniform{1234}uiv, **142**
ProgramUniformMatrix{234}, **142**
ProgramUniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}, **142**
PROJECTION, 500, 501, 505
PROJECTION_MATRIX (TRANSPOSE_PROJECTION_MATRIX), 738
PROJECTION_STACK_DEPTH, 738
PROVOKING_VERTEX, 497, 739
ProvokingVertex, 497, **535**
PROXY_COLOR_TABLE, 197, 203, 703
PROXY_HISTOGRAM, 209, 211, 703
PROXY_POST_COLOR_MATRIX_COLOR_TABLE, 197, 703
PROXY_POST_CONVOLUTION_COLOR_TABLE, 197, 703
PROXY_TEXTURE_1D, 238, 251, 280, 313, 323, 703
PROXY_TEXTURE_1D_ARRAY, 238, 250, 280, 314, 315, 323, 703
PROXY_TEXTURE_2D, 238, 250, 280, 314, 323, 703
PROXY_TEXTURE_2D_ARRAY, 236, 238, 280, 315, 316, 323, 703
PROXY_TEXTURE_2D_MULTISAMPLE, 238, 268, 280, 316, 317, 323
PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY, 238, 269, 280, 317, 323
PROXY_TEXTURE_3D, 236, 280, 315, 316, 323, 703
PROXY_TEXTURE_CUBE_MAP, 238, 250, 280, 314, 324, 703
PROXY_TEXTURE_CUBE_MAP_ARRAY, 236, 239, 249,

280, 315, 316, 323, 703
PROXY_TEXTURE_RECTANGLE, 238, 250, 262, 267, 280, 314, 323
PushAttrib, **704**, 704, 705, 707, 727, 844
PushClientAttrib, 436, 701, 704, **705**, 705, 707, 727, 844
PushDebugGroup, **677**, 678
PushMatrix, **505**, 841
PushName, **692**, 692, 693, 844

Q, 508–510
QUAD_STRIP, 383, 388, 529, 591, 689, 842
QUADRATIC_ATTENUATION, 517, 741
QUADS, 168, 383, 384, 388, 389, 529, 591, 776, 842
quads, 474, 476, 479, 485
QUADS_FOLLOW_PROVOKING_VERTEX_CONVENTION, 536, 809, 848
QUERY, 679
QUERY_BUFFER, 61, 162
QUERY_BUFFER_BARRIER_BIT, 162
QUERY_BUFFER_BINDING, 815
QUERY_BY_REGION_NO_WAIT, 438, 439
QUERY_BY_REGION_NO_WAIT_INVERTED, 438, 439
QUERY_BY_REGION_WAIT, 438, 439
QUERY_BY_REGION_WAIT_INVERTED, 438, 439
QUERY_COUNTER_BITS, 48, 49, 809, 906
QUERY_NO_WAIT, 438, 439
QUERY_NO_WAIT_INVERTED, 438, 439
QUERY_RESULT, 45, 46, 50, 51, 783
QUERY_RESULT_AVAILABLE, 21, 45, 46, 50, 51, 783, 874, 882
QUERY_RESULT_NO_WAIT, 51
QUERY_TARGET, 51, 783
QUERY_WAIT, 438, 439
QUERY_WAIT_INVERTED, 438, 439
QueryCounter, 44, **52**, 52

R, 508–510
R11F_G11F_B10F, 240, 243, 310, 333, 335
r11f_g11f_b10f, 333
R16, 242, 272, 310, 333, 336
r16, 333
R16_SNORM, 242, 310, 334, 336
r16_snorm, 334
R16F, 242, 272, 310, 333, 335
r16f, 333
R16I, 243, 272, 310, 333, 335
r16i, 333
R16UI, 243, 272, 310, 333, 335
r16ui, 333
R32F, 243, 272, 310, 333, 335
r32f, 333
R32I, 243, 272, 310, 333, 335
r32i, 333
R32UI, 243, 272, 310, 333, 335
r32ui, 333
R3_G3_B2, 242
R8, 242, 272, 310, 329, 333, 336, 784
r8, 333
R8_SNORM, 242, 310, 334, 336
r8_snorm, 334
R8I, 243, 272, 310, 333, 335
r8i, 333
R8UI, 243, 272, 310, 333, 335
r8ui, 333

RASTERIZER_DISCARD, 371, 542, 550, 742
RasterPos, 371, **576**, 693, 825
RasterPos2, 576
RasterPos3, 576
RasterPos4, 576
READ_BUFFER, 650, 758
READ_FRAMEBUFFER, 339–347, 349, 356–360, 370, 640, 643, 648, 660, 757
READ_FRAMEBUFFER_BINDING, 199, 206, 260, 343, 371, 651, 653, 658, 660, 757
READ_ONLY, 61, 75–77, 328, 329, 784
READ_PIXELS, 716
READ_PIXELS_FORMAT, 722
READ_PIXELS_TYPE, 722
READ_WRITE, 61, 67, 75–77, 328, 329, 737
ReadBuffer, 631, **648**, 648, 876
ReadnPixels, **650**, 651
ReadPixels, 162, 190, 193, 200, 219, 230, 253, 283, 371, 374, 533, 572, 644, 649, **650**, 650, 651, 653, 655, 657, 658, 660, 702, 722, 842
Rect, **437**, 437, 569
RECTANGLE_TEXTURE, 905
RED, 193, 197, 200, 217, 238, 242, 243, 246, 275, 285, 289, 323, 325, 335, 336, 366, 599–601, 653, 657, 747, 749, 759, 763, 764, 766, 814
RED_BIAS, 227
RED_BITS, 372, 843
RED_INTEGER, 217, 335
RED_SCALE, 227
REDUCE, 231, 232, 234, 765
REFERENCED_BY_COMPUTE_SHADER, 111, 114, 138, 139, 781
REFERENCED_BY_FRAGMENT_SHADER, 111, 114, 138, 139, 781
REFERENCED_BY_GEOMETRY_SHADER, 111, 114, 138, 139, 781
REFERENCED_BY_TESS_CONTROL_SHADER, 111, 114, 138, 139, 781
REFERENCED_BY_TESS_EVALUATION_SHADER, 111, 114, 138, 139, 781
REFERENCED_BY_VERTEX_SHADER, 111, 114, 138, 139, 781
REFLECTION_MAP, 508, 509
Relaxed, 829
ReleaseShaderCompiler, **89**, 89
RENDER, 693, 694, 815
RENDER_MODE, 815
RENDERBUFFER, 347, 349, 351–356, 358, 372, 664, 665, 679, 714, 722, 723, 760
RENDERBUFFER_ALPHA_SIZE, 355, 761
RENDERBUFFER_BINDING, 351, 760
RENDERBUFFER_BLUE_SIZE, 355, 761
RENDERBUFFER_DEPTH_SIZE, 355, 761
RENDERBUFFER_GREEN_SIZE, 355, 761
RENDERBUFFER_HEIGHT, 353, 355, 761
RENDERBUFFER_INTERNAL_FORMAT, 353, 355, 761
RENDERBUFFER_RED_SIZE, 355, 761

RENDERBUFFER_SAMPLES, 350, 353, 355, 368, 761
RENDERBUFFER_STENCIL_SIZE, 355, 761
RENDERBUFFER_WIDTH, 353, 355, 761
RenderbufferStorage, **354**, 702, 721
RenderbufferStorage*, 369
RenderbufferStorageMultisample, 344, **353**, 353, 354, 702, 721, 867
RENDERER, 711, 796
RenderMode, **693**, 693–696, 701, 844
REPEAT, 186, 276, 277, 297, 322, 751
REPLACE, 599, 601, 602, 616
REPLICATE_BORDER, 231, 233
RESCALE_NORMAL, 506, 738, 841
RESET_NOTIFICATION_STRATEGY, 20, 815
ResetHistogram, **210**
ResetMinmax, **213**
ResumeTransformFeedback, 527, **528**, 529, 532, 885
RETURN, 642
RG, 197, 217, 238, 242, 243, 246, 285, 335, 336, 366, 600, 601, 653, 657
RG16, 242, 272, 310, 333, 336
rg16, 333
RG16_SNORM, 242, 310, 333, 336
rg16_snorm, 333
RG16F, 243, 272, 310, 333, 335
rg16f, 333
RG16I, 243, 272, 310, 333, 335
rg16i, 333
RG16UI, 243, 272, 310, 333, 335
rg16ui, 333
RG32F, 243, 272, 310, 333, 335, 666
rg32f, 333
RG32I, 243, 272, 310, 333, 335, 666
rg32i, 333
RG32UI, 243, 273, 310, 333, 335, 666
rg32ui, 333
RG8, 242, 272, 310, 333, 336
rg8, 333
RG8_SNORM, 242, 310, 334, 336
rg8_snorm, 334
RG8I, 243, 272, 310, 333, 335
rg8i, 333
RG8UI, 243, 272, 310, 333, 335
rg8ui, 333
RG_INTEGER, 217, 335
RGB, 200, 217, 220, 225, 229, 230, 238, 241–243, 246, 285, 335, 366, 600, 601, 604, 623, 653, 655, 657
RGB10, 242
RGB10_A2, 242, 310, 333, 335
rgb10_a2, 333
RGB10_A2UI, 242, 310, 333, 335
rgb10_a2ui, 333
RGB12, 242
RGB16, 242, 310
RGB16_SNORM, 242, 310
RGB16F, 243, 310
RGB16I, 243, 310
RGB16UI, 243, 310
RGB32F, 243, 273, 310
RGB32I, 243, 273, 310
RGB32UI, 243, 273, 310
RGB4, 242, 867
RGB5, 242
RGB565, 242, 867
RGB5_A1, 242
RGB8, 242, 310
RGB8_SNORM, 242, 310
RGB8I, 243, 310
RGB8UI, 243, 310
RGB9_E5, 197, 240, 243, 310, 327, 655, 856
RGB_INTEGER, 217, 220

RGB_SCALE, 599, 600, 753
RGBA, 198, 200, 202, 206, 211, 212, 217, 220, 225, 229, 230, 238, 242, 243, 246, 285, 320, 321, 332, 335, 336, 366, 600, 601, 653, 658, 761, 764–767, 842
RGBA12, 242
RGBA16, 242, 273, 310, 333, 335, 666
rgba16, 333
RGBA16_SNORM, 242, 310, 333, 336, 666
rgba16_snorm, 333
RGBA16F, 243, 273, 310, 333, 335, 666
rgba16f, 333
RGBA16I, 243, 273, 310, 333, 335, 666
rgba16i, 333
RGBA16UI, 243, 273, 310, 333, 335, 666
rgba16ui, 333
RGBA2, 242
RGBA32F, 243, 273, 310, 333, 335, 666
rgba32f, 333
RGBA32I, 243, 273, 310, 333, 335, 666
rgba32i, 333
RGBA32UI, 243, 273, 310, 333, 335, 666
rgba32ui, 333
RGBA4, 242
RGBA8, 242, 273, 310, 333, 335
rgba8, 333
RGBA8_SNORM, 242, 310, 333, 336
rgba8_snorm, 333
RGBA8I, 243, 273, 310, 333, 335
rgba8i, 333
RGBA8UI, 243, 273, 310, 333, 335
rgba8ui, 333
RGBA_INTEGER, 217, 220, 332, 335
RGBA_MODE, 814
RIGHT, 619, 629, 631, 633, 634, 648
Rotate, **502**, 502, 825, 841
S, 508–510
Sample, 829
sample, 588, 592
sample in, 554, 588
SAMPLE_ALPHA_TO_COVERAGE, 612, 744
SAMPLE_ALPHA_TO_ONE, 612, 614, 744
SAMPLE_BUFFERS, 159, 260, 346, 350, 552, 561, 568, 575, 581, 584, 612, 617, 628, 636, 647, 651, 652, 660, 662, 813, 873, 876, 892
SAMPLE_COVERAGE, 584, 585, 592, 744
SAMPLE_COVERAGE_INVERT, 584, 585, 744
SAMPLE_COVERAGE_VALUE, 584, 585, 744
SAMPLE_MASK, 584, 585, 592, 744
SAMPLE_MASK_VALUE, 15, 584, 585, 744, 848
SAMPLE_POSITION, 346, 553, 813, 888
SAMPLE_SHADING, 554, 744
SampleCoverage, **585**
Sampled, 828, 904
Sampled Type, 828
Sampled1D, 833
SampledBuffer, 833
SampledCubeArray, 833
SampledImageArrayDynamicIndexing, 833
SampledRect, 833
SampleId, 832
SampleMask, 832
SampleMaski, **585**
SamplePosition, 832
SAMPLER, 679
sampler*, 156

sampler*Shadow, 459
sampler1D, 116
sampler1DArray, 116
sampler1DArrayShadow, 116
sampler1DShadow, 116
sampler2D, 116, 156
sampler2DArray, 116
sampler2DArrayShadow, 116
sampler2DMS, 116
sampler2DMSArray, 116
sampler2DRect, 116
sampler2DRectShadow, 116
sampler2DShadow, 116
sampler3D, 116
samplerCubeArray, 116
SAMPLER_1D, 116
SAMPLER_1D_ARRAY, 116
SAMPLER_1D_ARRAY_SHADOW, 116
SAMPLER_1D_SHADOW, 116
SAMPLER_2D, 116
SAMPLER_2D_ARRAY, 116
SAMPLER_2D_ARRAY_SHADOW, 116
SAMPLER_2D_MULTISAMPLE, 116
SAMPLER_2D_MULTISAMPLE_ARRAY, 116
SAMPLER_2D_RECT, 116
SAMPLER_2D_RECT_SHADOW, 116
SAMPLER_2D_SHADOW, 116
SAMPLER_3D, 116
SAMPLER_BINDING, 186, 726, 746
SAMPLER_BUFFER, 116
SAMPLER_CUBE, 116
SAMPLER_CUBE_MAP_ARRAY, 116
SAMPLER_CUBE_MAP_ARRAY_SHADOW, 116
SAMPLER_CUBE_SHADOW, 116
SampleRateShading, 833
samplerBuffer, 116
samplerCube, 116
samplerCubeArrayShadow, 116
samplerCubeShadow, 116
SamplerParameter, **188**
SamplerParameterI{i ui}v, **188**
SamplerParameterIiv, 188
SamplerParameterIuiv, 188
SamplerParameteriv, 188, 867
SAMPLES, 270, 346, 350, 552–554, 617, 662, 663, 722, 723, 811, 813, 873
SAMPLES_PASSED, 43, 46, 49, 438, 439, 617
Scale, 502, **503**, 825, 841
Scissor, **582**, 583, 584
SCISSOR_BIT, 706
SCISSOR_BOX, 754
SCISSOR_TEST, 583, 584, 754
ScissorArrayv, **582**, 582, 583
ScissorIndexed, **582**, 583, 584
ScissorIndexedv, **582**, 584
Scope, 828
SECONDARY_COLOR_ARRAY, 407, 426, 733
SECONDARY_COLOR_ARRAY_BUFFER_BINDING, 735
SECONDARY_COLOR_ARRAY_POINTER, 710, 733
SECONDARY_COLOR_ARRAY_SIZE, 733
SECONDARY_COLOR_ARRAY_STRIDE, 733
SECONDARY_COLOR_ARRAY_TYPE, 733
SecondaryColor, 394, 436
SecondaryColor3, **394**
SecondaryColorP*uiv, 395
SecondaryColorPointer, **405**, 406, 436, 701, 840
SELECT, 693, 694, 826

SelectBuffer, **693**, 693, 694, 701, 710, 844
SELECTION_BUFFER_POINTER, 710, 815
SELECTION_BUFFER_SIZE, 815
SEPARABLE_2D, 205, 207, 208, 230, 250, 765
SeparableFilter2D, 191, **205**
SEPARATE_ATTRIBS, 167, 450, 451
SEPARATE_SPECULAR_COLOR, 514
SET, 627
SHADE_MODEL, 739
ShadeModel, **537**, 841
SHADER, 679
Shader, 833
SHADER_BINARY_FORMAT_SPIR_V, 91, 92
SHADER_BINARY_FORMATS, 91, 795
SHADER_COMPILER, 86, 795
SHADER_IMAGE_ACCESS_BARRIER_BIT, 161, 164, 165
SHADER_IMAGE_ATOMIC, 716
SHADER_IMAGE_LOAD, 716
SHADER_IMAGE_STORE, 716
SHADER_SOURCE_LENGTH, 166, 171, 769
SHADER_STORAGE_BARRIER_BIT, 163, 165
SHADER_STORAGE_BLOCK, 103, 107, 110, 111
SHADER_STORAGE_BUFFER, 61, 62, 151, 152, 871
SHADER_STORAGE_BUFFER_BINDING, 84, 786
SHADER_STORAGE_BUFFER_OFFSET_ALIGNMENT, 84, 804, 889, 893
SHADER_STORAGE_BUFFER_SIZE, 84, 786
SHADER_STORAGE_BUFFER_START, 84, 786
SHADER_TYPE, 165, 166, 175, 769
ShaderBinary, 88, **90**, 91, 92, 166, 827, 859
ShaderSource, **88**, 88, 89, 91, 171, 702
ShaderStorageBlockBinding, **151**, 152, 871
SHADING_LANGUAGE_VERSION, 711–713, 796, 857
shared, 133, 670, 886
SHININESS, 517, 740
SHORT, 201, 215, 335, 336, 398, 656, 657, 700
SIGNALED, 21, 38, 42
SIGNED_NORMALIZED, 281, 348, 720
SIMULTANEOUS_TEXTURE_AND_DEPTH_TEST, 716
SIMULTANEOUS_TEXTURE_AND_DEPTH_WRITE, 716
SIMULTANEOUS_TEXTURE_AND_STENCIL_TEST, 717
SIMULTANEOUS_TEXTURE_AND_STENCIL_WRITE, 717
SINGLE_COLOR, 513, 514, 740
SLUMINANCE, 244, 326
SLUMINANCE8, 326
SLUMINANCE8_ALPHA8, 244, 326
SLUMINANCE_ALPHA, 326
SMOOTH, 537, 739
SMOOTH_LINE_WIDTH_GRANULARITY, 794
SMOOTH_LINE_WIDTH_RANGE, 794
SpecializeShader, 91, **92**
SPECULAR, 517, 519, 740, 741

SPHERE_MAP, 508, 509
SPIR_V_BINARY, 88, 89, 91, 93, 96, 105, 166, 769
SPIR_V_EXTENSIONS, 713, 796
SPOT_CUTOFF, 517, 741
SPOT_DIRECTION, 517, 519, 741
SPOT_EXPONENT, 517, 741
sqrt(), 834
SRC0_ALPHA, 753
SRC0_RGB, 753
SRC1_ALPHA, 622–624, 753
SRC1_COLOR, 622–624
SRC1_RGB, 753
SRC2_ALPHA, 753
SRC2_RGB, 753
SRC_ALPHA, 603, 604, 623, 624, 753
SRC_ALPHA_SATURATE, 623
SRC_COLOR, 603, 604, 623, 624, 753
SRC*n*_ALPHA, 600, 603
SRC*n*_RGB, 600, 603
SRGB, 254, 326, 348, 620, 625, 627, 642, 662, 718
SRGB8, 242, 310, 326
SRGB8_ALPHA8, 242, 310, 326
SRGB_ALPHA, 326
SRGB_READ, 717
SRGB_WRITE, 717
STACK_OVERFLOW, 18, 505, 678, 693, 705, 854, 858
STACK_UNDERFLOW, 18, 505, 678, 692, 705, 854, 858
STATIC_COPY, 61, 69
STATIC_DRAW, 61, 69, 737
STATIC_READ, 61, 69
std140, 133, 145, 146, 150, 886
std430, 146, 150, 670, 901
STENCIL, 347, 348, 638, 639, 641, 658, 749, 759
STENCIL_ATTACHMENT, 341, 357, 367
STENCIL_BACK_FAIL, 754
STENCIL_BACK_FUNC, 754
STENCIL_BACK_PASS_DEPTH_FAIL, 754
STENCIL_BACK_PASS_DEPTH_PASS, 754
STENCIL_BACK_REF, 754
STENCIL_BACK_VALUE_MASK, 754
STENCIL_BACK_WRITEMASK, 756
STENCIL_BITS, 372, 660, 814, 843
STENCIL_BUFFER_BIT, 636, 639, 661, 663, 706
STENCIL_CLEAR_VALUE, 756
STENCIL_COMPONENTS, 723
STENCIL_FAIL, 754
STENCIL_FUNC, 754
STENCIL_INDEX, 196, 203, 213, 217, 238, 245, 249, 250, 253, 275, 283, 284, 305, 307, 320, 321, 325, 366, 459, 644, 647, 648, 652, 653, 658, 660, 888, 898
STENCIL_INDEX1, 245
STENCIL_INDEX16, 245
STENCIL_INDEX4, 245
STENCIL_INDEX8, 245, 856
STENCIL_PASS_DEPTH_FAIL, 754
STENCIL_PASS_DEPTH_PASS, 754
STENCIL_REF, 754
STENCIL_RENDERABLE, 723
STENCIL_TEST, 615, 754
STENCIL_VALUE_MASK, 754
STENCIL_WRITEMASK, 15, 756
StencilFunc, **615**, 615, 616, 824
StencilFuncSeparate, **615**, 615, 616
StencilMask, **635**, 635, 636, 648, 824
StencilMaskSeparate, **635**, 635, 636, 648
StencilOp, **615**, 615, 616
StencilOpSeparate, **615**, 615, 616

STEREO, 346, 813
Storage Class, 828
StorageBufferArrayDynamicIndexing, 833
StorageImageArrayDynamicIndexing, 833
StorageImageExtendedFormats, 833
StorageImageMultisample, 833
StorageImageWriteWithoutFormat, 829, 833, 900
STREAM_COPY, 61, 69
STREAM_DRAW, 61, 68
STREAM_READ, 61, 69
Subgroup, 828
SubgroupVoteKHR, 833
SUBPIXEL_BITS, 792
SUBTRACT, 602
switch, 473
SYNC_CONDITION, 38, 42, 789
SYNC_FENCE, 38, 42, 789
SYNC_FLAGS, 38, 42, 789, 885
SYNC_FLUSH_COMMANDS_BIT, 21, 39–41, 885
SYNC_GPU_COMMANDS_COMPLETE, 38, 42, 66, 67, 789
SYNC_STATUS, 21, 38, 42, 789

T, 508–510
T2F_C3F_V3F, 427
T2F_C4F_N3F_V3F, 427
T2F_C4UB_V3F, 427
T2F_N3F_V3F, 427
T2F_V3F, 427
T4F_C4F_N3F_V4F, 427
T4F_V4F, 427
TABLE_TOO_LARGE, 18, 197, 209
TESS_CONTROL_OUTPUT_VERTICES, 167, 168, 469, 776
TESS_CONTROL_SHADER, 88, 154, 469, 770
TESS_CONTROL_SHADER_BIT, 123
TESS_CONTROL_SHADER_PATCHES, 44, 49, 488
TESS_CONTROL_SUBROUTINE, 102, 154
TESS_CONTROL_SUBROUTINE_UNIFORM, 102, 107, 110, 111, 119, 154
TESS_CONTROL_TEXTURE, 717
TESS_EVALUATION_SHADER, 88, 154, 484, 770
TESS_EVALUATION_SHADER_BIT, 123
TESS_EVALUATION_SHADER_INVOCATIONS, 44, 49, 488
TESS_EVALUATION_SUBROUTINE, 102, 154
TESS_EVALUATION_SUBROUTINE_UNIFORM, 102, 107, 110, 111, 119, 154
TESS_EVALUATION_TEXTURE, 717
TESS_GEN_MODE, 168, 776
TESS_GEN_POINT_MODE, 168, 169, 776
TESS_GEN_SPACING, 168, 776
TESS_GEN_VERTEX_ORDER, 168, 776
TessCoord, 832
Tessellation, 833
TessellationPointSize, 833
TessLevelInner, 832
TessLevelOuter, 832
TexBuffer, **271**, 702, 721
TexBufferRange, **270**, 270
TexCoord, **392**, 436
TexCoordPointer, **405**, 406, 407, 426, 436, 701, 840
texelFetch, 455, 887
TexEnv, 179, 504, **598**, 606, 843
TexGen, 278, 504, **507**, 508, 509

TexImage, 179, 257, 320
TexImage*D, 190, 261
TexImage1D, 230, 231, 244, **251**, 251, 252, 254, 255, 261, 265, 302, 323, 703, 721
TexImage2D, 230, 231, 244, 249, **250**, 250–253, 257, 261, 265, 302, 323, 332, 703, 721
TexImage2DMultisample, **268**, 268, 317, 323, 344, 721, 871
TexImage3D, **236**, 237, 244, 247, 249, 250, 252, 261, 265, 283, 302, 320, 323, 703, 721, 867
TexImage3DMultisample, **268**, 268, 269, 317, 323, 721, 867, 871
TexParameter, 56, 179, 188, **274**, 843
TexParameter*, 867
TexParameterf, 184
TexParameterfv, 184
TexParameterI, **274**
TexParameteri, 184, 274
TexParameterIiv, 274
TexParameterIuiv, 274
TexParameteriv, 184, 274
TexStorage, 320
TexStorage1D, **313**, 313, 702, 721
TexStorage2D, **313**, 314, 315, 702, 721
TexStorage2DMultisample, **316**, 316, 317, 721
TexStorage3D, **315**, 315, 316, 702, 721, 898
TexStorage3DMultisample, **317**, 317, 721
TexSubImage, 162
TexSubImage1D, 230, **255**, 256
TexSubImage2D, 230, **255**, 256, 257, 310
TexSubImage3D, **255**, 256, 258, 285, 318, 319, 321, 335
TEXTURE, 347, 349, 362, 372, 500, 504, 505, 603, 604, 679, 753
TEXTURE0, 179–181, 393, 432, 434, 685, 686, 696, 707, 736, 752
TEXTURE1, 707
TEXTURE_x_SIZE, 749
TEXTURE_x_TYPE, 749
TEXTURE_xD, 745, 746
TEXTURE_1D, 180, 238, 251, 254, 256, 274, 278, 280, 282, 283, 286, 303, 309, 311, 313, 331, 360, 605, 714, 843
TEXTURE_1D_ARRAY, 180, 238, 250, 253, 256, 274, 278, 280, 282, 283, 286, 303, 309, 312, 314, 315, 331, 714, 745, 746, 843
TEXTURE_2D, 156, 179, 180, 238, 249, 250, 253, 256, 274, 278, 280, 282, 283, 286, 303, 309, 311, 314, 331, 360, 605, 714, 843
TEXTURE_2D_ARRAY, 180, 236, 238, 247, 249, 256, 265, 268, 274, 278, 280, 282, 283, 303, 309, 311, 312, 315, 316, 331, 714, 745, 746, 843
TEXTURE_2D_MULTISAMPLE, 180, 238, 268, 274, 277, 279, 280, 309, 311, 316, 317, 319, 331, 360, 714, 722, 723, 745
TEXTURE_2D_MULTISAMPLE_ARRAY, 180, 238, 268, 274, 277, 279, 280, 309, 312, 317–319, 331, 714, 722, 723, 745
TEXTURE_3D, 180, 236, 247, 256, 274, 278, 280, 282, 283, 303, 309, 311, 315, 316, 323, 331, 360, 605, 714, 843
TEXTURE_ALPHA_SIZE, 281
TEXTURE_ALPHA_TYPE, 281
TEXTURE_BASE_LEVEL, 247, 275–

277, 291, 308, 322, 747
TEXTURE_BINDING_xD, 745
TEXTURE_BINDING_1D, 725
TEXTURE_BINDING_1D_ARRAY, 725, 745
TEXTURE_BINDING_2D, 725
TEXTURE_BINDING_2D_ARRAY, 725, 745
TEXTURE_BINDING_2D_MULTISAMPLE, 725, 745
TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY, 725, 745
TEXTURE_BINDING_3D, 726
TEXTURE_BINDING_BUFFER, 726, 745
TEXTURE_BINDING_CUBE_MAP, 726, 745
TEXTURE_BINDING_CUBE_MAP_ARRAY, 726, 745
TEXTURE_BINDING_RECTANGLE, 726, 745
TEXTURE_BIT, 705, 706
TEXTURE_BLUE_SIZE, 281
TEXTURE_BLUE_TYPE, 281
TEXTURE_BORDER, 264, 266, 281, 749
TEXTURE_BORDER_COLOR, 188, 190, 274, 275, 277, 279, 297, 322, 747, 751
TEXTURE_BUFFER, 61, 180, 270, 271, 273, 280, 309, 319, 331, 664, 665, 714, 745
TEXTURE_BUFFER_BINDING, 815, 873
TEXTURE_BUFFER_DATA_STORE_BINDING, 750
TEXTURE_BUFFER_OFFSET, 750
TEXTURE_BUFFER_OFFSET_ALIGNMENT, 271, 795, 889, 893
TEXTURE_BUFFER_SIZE, 750
TEXTURE_COMPARE_FAIL_VALUE_ARB, 913
TEXTURE_COMPARE_FUNC, 275, 323, 324, 748, 751
TEXTURE_COMPARE_MODE, 275, 323–325, 459, 748, 751
TEXTURE_COMPONENTS, 281, 842
TEXTURE_COMPRESSED, 723, 750
TEXTURE_COMPRESSED_BLOCK_HEIGHT, 723
TEXTURE_COMPRESSED_BLOCK_SIZE, 724
TEXTURE_COMPRESSED_BLOCK_WIDTH, 724
TEXTURE_COMPRESSED_IMAGE_SIZE, 264, 266, 281, 282, 287, 750
TEXTURE_COMPRESSION_HINT, 704, 790
TEXTURE_COORD_ARRAY, 407, 426, 733
TEXTURE_COORD_ARRAY_BUFFER_BINDING, 735
TEXTURE_COORD_ARRAY_POINTER, 710, 733
TEXTURE_COORD_ARRAY_SIZE, 733
TEXTURE_COORD_ARRAY_STRIDE, 733
TEXTURE_COORD_ARRAY_TYPE, 733
TEXTURE_CUBE_MAP, 180, 238, 251, 252, 256, 259, 265, 268, 274, 278, 280, 282–284, 286, 287, 303, 304, 309, 311, 312, 314, 331, 605, 714, 745, 843, 884, 887, 901
TEXTURE_CUBE_MAP_ARRAY, 180, 236, 238, 247, 249, 252, 256,

259, 265, 268, 274, 278, 280, 282–284, 286, 287, 303, 304, 309, 311, 312, 315, 316, 331, 714, 745, 746, 884, 887, 898, 901, 903
TEXTURE_CUBE_MAP_NEGATIVE_X, 290, 374, 746
TEXTURE_CUBE_MAP_NEGATIVE_Y, 290, 374, 746
TEXTURE_CUBE_MAP_NEGATIVE_Z, 290, 374, 746
TEXTURE_CUBE_MAP_POSITIVE_X, 280, 290, 329, 374, 746
TEXTURE_CUBE_MAP_POSITIVE_Y, 290, 374, 746
TEXTURE_CUBE_MAP_POSITIVE_Z, 290, 374, 746
TEXTURE_CUBE_MAP_SEAMLESS, 290, 815
TEXTURE_DEPTH, 264, 266, 281, 749
TEXTURE_DEPTH_SIZE, 281
TEXTURE_DEPTH_TYPE, 281
TEXTURE_ENV, 598, 599, 604, 843
TEXTURE_ENV_COLOR, 599, 752
TEXTURE_ENV_MODE, 599, 600, 606, 752
TEXTURE_FETCH_BARRIER_BIT, 161, 165
TEXTURE_FILTER_CONTROL, 598, 599, 604, 843
TEXTURE_FIXED_SAMPLE_LOCATIONS, 269, 281, 368, 749
TEXTURE_GATHER, 717
TEXTURE_GATHER_SHADOW, 717
TEXTURE_GEN_x, 752
TEXTURE_GEN_MODE, 508, 509, 752
TEXTURE_GEN_Q, 509
TEXTURE_GEN_R, 509
TEXTURE_GEN_S, 509
TEXTURE_GEN_T, 509
TEXTURE_GREEN_SIZE, 281
TEXTURE_GREEN_TYPE, 281
TEXTURE_HEIGHT, 259, 264, 266, 268, 269, 281, 749
TEXTURE_IMAGE_FORMAT, 724
TEXTURE_IMAGE_TYPE, 724
TEXTURE_IMMUTABLE_FORMAT, 270, 279, 308, 311, 312, 322, 748
TEXTURE_IMMUTABLE_LEVELS, 279, 302, 308, 312, 323, 748, 906
TEXTURE_INTENSITY_SIZE, 281
TEXTURE_INTENSITY_TYPE, 281
TEXTURE_INTERNAL_FORMAT, 264, 266, 269, 281, 842
TEXTURE_INTERNAL_FORMAT (TEXTURE_COMPONENTS), 749
TEXTURE_LOD_BIAS, 275, 293, 599, 747, 751, 752, 843
TEXTURE_LUMINANCE_SIZE, 281
TEXTURE_LUMINANCE_TYPE, 281
TEXTURE_MAG_FILTER, 275, 305, 322, 325, 747, 751
TEXTURE_MATRIX (TRANSPOSE_TEXTURE_MATRIX), 738
TEXTURE_MAX_ANISOTROPHY, 292
TEXTURE_MAX_ANISOTROPY, 275, 277, 291, 292, 748, 751, 906
TEXTURE_MAX_LEVEL, 275–277, 291, 308, 322, 747

TEXTURE_MAX_LOD, 275, 276, 291, 322, 747, 751
TEXTURE_MIN_FILTER, 186, 275, 277, 296, 297, 301, 305, 307, 322, 325, 747, 751
TEXTURE_MIN_LOD, 275, 276, 291, 322, 747, 751
TEXTURE_PRIORITY, 184, 274, 275, 322, 747, 843
TEXTURE_RECTANGLE, 180, 238, 249, 250, 253, 256, 260, 262, 267, 274, 277, 278, 280, 282–284, 286, 309, 311, 314, 315, 319, 331, 360, 605, 714, 745, 746, 873
TEXTURE_RECTANGLE_ARB, 915
TEXTURE_RED_SIZE, 281
TEXTURE_RED_TYPE, 281
TEXTURE_RESIDENT, 184, 279, 323, 747
TEXTURE_SAMPLES, 269, 281, 350, 368, 749
TEXTURE_SHADOW, 717
TEXTURE_SHARED_SIZE, 281, 749
TEXTURE_STACK_DEPTH, 738
TEXTURE_STENCIL_SIZE, 281
TEXTURE_SWIZZLE_A, 274, 275, 323, 599, 600, 747
TEXTURE_SWIZZLE_B, 274, 275, 323, 599, 600, 747
TEXTURE_SWIZZLE_G, 274, 275, 323, 599, 600, 747
TEXTURE_SWIZZLE_R, 274, 275, 323, 599, 747
TEXTURE_SWIZZLE_RGBA, 188, 274, 275, 277
TEXTURE_TARGET, 256, 279, 285, 287, 288, 747
TEXTURE_UPDATE_BARRIER_BIT, 162
TEXTURE_VIEW, 717
TEXTURE_VIEW_MIN_LAYER, 279, 308, 323, 748
TEXTURE_VIEW_MIN_LEVEL, 279, 308, 310, 323, 748
TEXTURE_VIEW_NUM_LAYERS, 279, 309, 311, 312, 323, 748
TEXTURE_VIEW_NUM_LEVELS, 279, 309, 311, 312, 323, 359, 361, 748, 906
TEXTURE_WIDTH, 259, 264, 266, 268, 269, 281, 749
TEXTURE_WRAP_R, 186, 276, 296, 297, 747, 751, 843
TEXTURE_WRAP_S, 186, 276, 277, 296, 297, 747, 751, 843
TEXTURE_WRAP_T, 186, 276, 277, 296, 297, 747, 751, 843
TextureBarrier, **364**, 365
TextureBuffer, **271**
TextureBufferRange, **270**, 270, 271, 906
textureGather, 294, 299, 300, 801, 921
textureGather*, 887
textureGatherOffset, 300
textureGatherOffsets, 300, 887
textureLOD, 921
TextureParameter, **274**
TextureParameter*, 277
TextureParameterI, **274**
textureQueryLevels, 457
textureSize, 457, 901
TextureStorage*, 313
TextureStorage1D, **313**, 313, 906
TextureStorage2D, **314**, 315
TextureStorage2DMultisample, **316**, 317
TextureStorage3D, **315**, 316
TextureStorage3DMultisample, **317**, 317
TextureSubImage1D, **255**, 256

TextureSubImage2D, **255**, 256, 257
TextureSubImage3D, **255**, 256, 257
TextureView, 239, 276, **308**, 309, 310, 717, 721, 860, 886, 891
TEXTURE*n*, 603
TIME_ELAPSED, 43, 49, 51, 52, 858
TIMEOUT_EXPIRED, 39
TIMEOUT_IGNORED, 40
TIMESTAMP, 43, 45–49, 52, 858
TOP_LEVEL_ARRAY_SIZE, 111, 114, 781, 871, 890
TOP_LEVEL_ARRAY_STRIDE, 111, 114, 781, 890
TRANSFORM_BIT, 706
TRANSFORM_FEEDBACK, 526, 527, 679
TRANSFORM_FEEDBACK_ACTIVE, 725, 787, 847
TRANSFORM_FEEDBACK_BARRIER_BIT, 162
TRANSFORM_FEEDBACK_BINDING, 738
TRANSFORM_FEEDBACK_BUFFER, 61, 62, 102, 105, 107–111, 452, 529, 530, 532
TRANSFORM_FEEDBACK_BUFFER_ACTIVE, 847
TRANSFORM_FEEDBACK_BUFFER_BINDING, 84, 526, 725, 738, 787, 904
TRANSFORM_FEEDBACK_BUFFER_INDEX, 111, 114
TRANSFORM_FEEDBACK_BUFFER_MODE, 167, 773
TRANSFORM_FEEDBACK_BUFFER_PAUSED, 847
TRANSFORM_FEEDBACK_BUFFER_SIZE, 84, 725, 787
TRANSFORM_FEEDBACK_BUFFER_START, 84, 725, 787
TRANSFORM_FEEDBACK_BUFFER_STRIDE, 111, 114
TRANSFORM_FEEDBACK_OVERFLOW, 43, 49, 438, 439, 535
TRANSFORM_FEEDBACK_PAUSED, 725, 787, 847
TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, 43, 46, 47, 49, 531, 534
TRANSFORM_FEEDBACK_STREAM_OVERFLOW, 43, 46, 47, 49, 438, 439, 535
TRANSFORM_FEEDBACK_VARYING, 102, 105, 108, 110, 111, 113, 452, 453
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, 167, 773
TRANSFORM_FEEDBACK_VARYINGS, 167, 453, 773
TransformFeedback, 833
TransformFeedbackBufferBase, **529**, 529
TransformFeedbackBufferRange, **529**, 529, 530
TransformFeedbackVarying, 453
TransformFeedbackVaryings, 102, 105, 108, 114, **449**, 451, 453, 702
Translate, 502, **503**, 825, 841
TRANSPOSE_COLOR_MATRIX, 209, 505, 506
TRANSPOSE_MODELVIEW_MATRIX, 505
TRANSPOSE_PROJECTION_MATRIX, 505, 506
TRANSPOSE_TEXTURE_MATRIX,

505, 506
TRIANGLE_FAN, 382, 388, 490, 497, 529
TRIANGLE_STRIP, 167, 382, 388, 490, 497, 529, 544, 773
TRIANGLE_STRIP_ADJACENCY, 386, 388, 490, 498
TRIANGLES, 167, 168, 383, 388, 389, 490, 497, 527–529, 531, 773
triangles, 474, 476, 477, 490
TRIANGLES_ADJACENCY, 167, 385, 388, 490, 498
triangles_adjacency, 490
TRUE, 14, 15, 21, 42, 45, 48, 50, 60, 61, 67, 68, 70, 71, 75, 78, 86, 89–91, 93, 95, 98–100, 105, 121, 131, 132, 141, 165, 166, 168, 183, 184, 189, 191, 193, 209–212, 269, 270, 275, 276, 304, 308, 311, 312, 328–331, 343, 349, 353, 363, 368, 389, 390, 397, 399, 410, 430, 438, 465, 513, 515, 524, 526, 536, 540, 557, 560, 585, 618, 634, 645, 651, 654, 671, 676, 701, 718, 720, 722, 723, 731, 737, 739, 744, 749, 755, 756, 783, 795, 807, 841, 882
true, 590
TYPE, 105, 111, 112, 114, 136, 137, 445, 453, 781

uimage1D, 118
uimage1DArray, 118
uimage2D, 118
uimage2DArray, 118
uimage2DMS, 118
uimage2DMSArray, 118
uimage2DRect, 118
uimage3D, 118
uimageBuffer, 118
uimageCube, 118
uimageCubeArray, 118
uint, 115, 126, 144, 149, 443
UNDEFINED_VERTEX, 497
UNIFORM, 101, 105, 110, 111, 113, 119, 135, 136
Uniform, **11**, 146, 147, 828, 829
Uniform1f, **11**
Uniform1i, **11**
Uniform2f, **11**
Uniform2i, **11**
Uniform3f, **11**
Uniform3i, **11**
Uniform4f, **11, 12**
Uniform4f{v}, 141
Uniform4i, **12**
UNIFORM_ARRAY_STRIDE, 137, 144, 149, 775, 859, 902
UNIFORM_ATOMIC_COUNTER_BUFFER_INDEX, 137, 778
UNIFORM_BARRIER_BIT, 161, 165
UNIFORM_BLOCK, 101, 107, 110, 111, 136, 137
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, 138, 775
UNIFORM_BLOCK_ACTIVE_UNIFORMS, 138, 775
UNIFORM_BLOCK_BINDING, 138, 775
UNIFORM_BLOCK_DATA_SIZE, 138, 148, 775
UNIFORM_BLOCK_INDEX, 137, 774
UNIFORM_BLOCK_NAME_LENGTH, 138, 775, 877
UNIFORM_BLOCK_REFERENCED_BY_COMPUTE_SHADER, 138, 776

UNIFORM_BLOCK_REFERENCED_-BY_FRAGMENT_SHADER, 138, 776
UNIFORM_BLOCK_REFERENCED_-BY_GEOMETRY_SHADER, 138, 776
UNIFORM_BLOCK_REFERENCED_-BY_TESS_CONTROL_-SHADER, 138, 775
UNIFORM_BLOCK_REFERENCED_-BY_TESS_EVALUATION_-SHADER, 138
UNIFORM_BLOCK_REFERENCED_-BY_TESS_EVALUTION_-SHADER, 775
UNIFORM_BLOCK_REFERENCED_-BY_VERTEX_SHADER, 138, 775
UNIFORM_BUFFER, 61, 62, 147
UNIFORM_BUFFER_BINDING, 84, 788
UNIFORM_BUFFER_OFFSET_-ALIGNMENT, 84, 803, 889, 893
UNIFORM_BUFFER_SIZE, 84, 788
UNIFORM_BUFFER_START, 84, 788
UNIFORM_IS_ROW_MAJOR, 137, 775
UNIFORM_MATRIX_STRIDE, 137, 775
UNIFORM_NAME_LENGTH, 137, 774, 777
UNIFORM_OFFSET, 137, 774
UNIFORM_SIZE, 137, 774, 777
UNIFORM_TYPE, 137, 774
Uniform{1234}{ifd ui}, **139**
Uniform{1234}{ifd ui}v, **139**
UniformBlockBinding, **148**, 148
UniformBufferArrayDynamicIndexing, 833
UniformConstant, 828
UniformMatrix2x4fv, 140
UniformMatrix3dv, 141
UniformMatrix{234}, **140**
UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}, **140**
UniformSubroutinesuiv, **155**, 867
Unknown, 829
UNKNOWN_CONTEXT_RESET, 20
UnmapBuffer, 56, 60, 67, 69, 74, **78**, 78, 702
UnmapNamedBuffer, **78**, 78
UNPACK_ALIGNMENT, 191, 218, 237, 263, 762
UNPACK_COMPRESSED_BLOCK_-DEPTH, 191, 261, 762
UNPACK_COMPRESSED_BLOCK_-HEIGHT, 191, 261, 762
UNPACK_COMPRESSED_BLOCK_-SIZE, 191, 261, 762
UNPACK_COMPRESSED_BLOCK_-WIDTH, 191, 261, 762
UNPACK_IMAGE_HEIGHT, 191, 237, 262, 264, 762
UNPACK_LSB_FIRST, 191, 645, 762, 840, 854, 862
UNPACK_ROW_LENGTH, 191, 216, 218, 237, 262, 263, 762
UNPACK_SKIP_IMAGES, 191, 237, 250, 262, 264, 762
UNPACK_SKIP_PIXELS, 191, 218, 262, 263, 645, 762
UNPACK_SKIP_ROWS, 191, 218, 262, 263, 645, 762
UNPACK_SWAP_BYTES, 191, 216, 218, 762
UNSIGNALED, 38, 42, 789
UNSIGNED_BYTE, 201, 215, 335, 336, 398, 400, 410, 419, 427, 656, 657, 700

UNSIGNED_BYTE_2_-3_3_REV, 201, 215, 220, 221, 656
UNSIGNED_BYTE_3_3_2, 201, 215, 220, 221, 656
UNSIGNED_INT, 115, 195, 196, 201, 215, 281, 332, 335, 348, 398, 410, 419, 656, 657, 700, 720
UNSIGNED_INT_10_10_10_2, 201, 215, 220, 223, 656
UNSIGNED_INT_10F_11F_11F_REV, 215, 220, 223, 225, 335, 390, 392, 399, 400, 405, 407, 411, 655, 656
UNSIGNED_INT_24_8, 215, 220, 223, 652, 656, 657
UNSIGNED_INT_2_-10_10_10_REV, 201, 215, 220, 223, 335, 390, 392–394, 398–400, 410, 433, 656
UNSIGNED_INT_5_9_9_9_REV, 215, 220, 223, 225, 241, 655, 656
UNSIGNED_INT_8_8_8_8, 201, 215, 220, 223, 656
UNSIGNED_INT_8_8_-8_8_REV, 201, 215, 220, 223, 656
UNSIGNED_INT_ATOMIC_-COUNTER, 118
UNSIGNED_INT_IMAGE_1D, 118
UNSIGNED_INT_IMAGE_1D_AR-RAY, 118
UNSIGNED_INT_IMAGE_2D, 118
UNSIGNED_INT_IMAGE_2D_AR-RAY, 118
UNSIGNED_INT_IMAGE_2D_MUL-TISAMPLE, 118
UNSIGNED_INT_IMAGE_2D_MUL-TISAMPLE_ARRAY, 118
UNSIGNED_INT_IMAGE_2D_RECT, 118
UNSIGNED_INT_IMAGE_3D, 118
UNSIGNED_INT_IMAGE_BUFFER, 118
UNSIGNED_INT_IMAGE_CUBE, 118
UNSIGNED_INT_IMAGE_CUBE_-MAP_ARRAY, 118
UNSIGNED_INT_SAMPLER_1D, 117
UNSIGNED_INT_SAMPLER_1D_AR-RAY, 117
UNSIGNED_INT_SAMPLER_2D, 117
UNSIGNED_INT_SAMPLER_2D_AR-RAY, 117
UNSIGNED_INT_SAMPLER_2D_-MULTISAMPLE, 117
UNSIGNED_INT_SAMPLER_2D_-MULTISAMPLE_ARRAY, 117
UNSIGNED_INT_SAMPLER_2D_-RECT, 117
UNSIGNED_INT_SAMPLER_3D, 117
UNSIGNED_INT_SAMPLER_-BUFFER, 117
UNSIGNED_INT_SAMPLER_CUBE, 117
UNSIGNED_INT_SAMPLER_CUBE_-MAP_ARRAY, 117
UNSIGNED_INT_VEC2, 115
UNSIGNED_INT_VEC3, 115
UNSIGNED_INT_VEC4, 115
UNSIGNED_NORMALIZED, 281, 348, 720
UNSIGNED_SHORT, 195, 196, 201, 215, 335, 336, 398, 410, 419, 656, 657, 700
UNSIGNED_SHORT_1_5_-5_5_REV, 201, 215, 220, 222, 656
UNSIGNED_SHORT_4_4_4_4, 201,

215, 220, 222, 656
UNSIGNED_SHORT_4_4_-4_4_REV, 201, 215, 220, 222, 656
UNSIGNED_SHORT_5_5_5_1, 201, 215, 220, 222, 656
UNSIGNED_SHORT_5_6_5, 201, 215, 220, 222, 656
UNSIGNED_SHORT_5_6_5_REV, 201, 215, 220, 222, 656
UPPER_LEFT, 537, 543, 556, 560, 569, 742
usampler1D, 117
usampler1DArray, 117
usampler2D, 117
usampler2DArray, 117
usampler2DMS, 117
usampler2DMSArray, 117
usampler2DRect, 117
usampler3D, 117
usamplerBuffer, 117
usamplerCube, 117
usamplerCubeArray, 117
UseProgram, **97**, 97, 98, 122, 140, 156, 464–466, 532, 859, 883, 891
UseProgramStages, 98, **122**, 123, 156, 168, 464, 532, 859, 871, 883
uvec2, 115, 443
uvec3, 115, 443
uvec4, 115, 332, 443

V2F, 427
V3F, 427
VALIDATE_STATUS, 166, 169, 465, 466, 770, 771
ValidateProgram, 166, **465**, 465, 702, 873
ValidateProgramPipeline, 169, **466**, 873
vec2, 115, 443
vec3, 115, 443
vec4, 115, 141, 145, 146, 332, 443
VENDOR, 711, 796
VERSION, 711, 796
Vertex, **433**, 433, 436, 576, 687
Vertex2, 437
VERTEX_ARRAY, 407, 428, 679, 732
VERTEX_ARRAY_BINDING, 398, 429, 430, 736
VERTEX_ARRAY_BUFFER_BINDING, 735
VERTEX_ARRAY_POINTER, 710, 732
VERTEX_ARRAY_SIZE, 732
VERTEX_ARRAY_STRIDE, 732
VERTEX_ARRAY_TYPE, 732
VERTEX_ATTRIB_ARRAY_BARRIER_BIT, 161
VERTEX_ATTRIB_ARRAY_BUFFER, 161
VERTEX_ATTRIB_ARRAY_-BUFFER_BINDING, 412, 429, 430, 735, 869, 899
VERTEX_ATTRIB_-ARRAY_DIVISOR, 429, 430, 734, 869
VERTEX_ATTRIB_ARRAY_ENABLED, 429, 430, 734
VERTEX_ATTRIB_-ARRAY_INTEGER, 429, 430, 734
VERTEX_ATTRIB_ARRAY_LONG, 429, 430, 734, 854, 859
VERTEX_ATTRIB_ARRAY_NORMALIZED, 429, 430, 734
VERTEX_ATTRIB_-ARRAY_POINTER, 404, 431, 734
VERTEX_ATTRIB_ARRAY_SIZE, 429, 430, 734
VERTEX_ATTRIB_ARRAY_STRIDE,

404, 429, 430, 734
VERTEX_ATTRIB_ARRAY_TYPE, 429, 430, 734
VERTEX_ATTRIB_BINDING, 430, 735
VERTEX_ATTRIB_RELATIVE_OFFSET, 429, 430, 735
VERTEX_BINDING_BUFFER, 429, 430, 735, 870, 899, 900
VERTEX_BINDING_DIVISOR, 429, 430, 735, 870, 899, 900
VERTEX_BINDING_OFFSET, 429, 735
VERTEX_BINDING_STRIDE, 404, 429, 735, 899
VERTEX_PROGRAM_TWO_SIDE, 515, 516, 782, 841
VERTEX_SHADER, 88, 154, 172, 770
VERTEX_SHADER_BIT, 123
VERTEX_SHADER_INVOCATIONS, 43, 49, 467
VERTEX_SUBROUTINE, 102, 154
VERTEX_SUBROUTINE_UNIFORM, 102, 107, 110, 111, 119, 154
VERTEX_TEXTURE, 717
VertexArrayAttribBinding, **403**, 403
VertexArrayAttribFormat, **398**
VertexArrayAttribIFormat, **398**
VertexArrayAttribLFormat, **398**
VertexArrayBindingDivisor, **408**, 408
VertexArrayElementBuffer, **397**
VertexArrayVertexBuffer, **400**, 400, 401
VertexArrayVertexBuffers, **401**, 401, 402
VertexAttrib, **390**, 436, 438
VertexAttrib4, **390**
VertexAttrib4N, **390**
VertexAttrib4Nub, **390**, 874
VertexAttribBinding, **402**, 403, 404, 412
VertexAttribDivisor, **408**, 414, 415, 419
VertexAttribFormat, **398**, 399
VertexAttribI, **390**
VertexAttribI1i, 443
VertexAttribI1ui, 443
VertexAttribI2i, 443
VertexAttribI2ui, 443
VertexAttribI3i, 443
VertexAttribI3ui, 443
VertexAttribI4, **390**
VertexAttribI4i, 443
VertexAttribI4ui, 443
VertexAttribIFormat, **398**, 399, 400
VertexAttribIPointer, **403**, 430, 701
VertexAttribL1d, 443
VertexAttribL2d, 443
VertexAttribL3d, 443
VertexAttribL3dv, 444
VertexAttribL4d, 443
VertexAttribL{1234}d, **390**
VertexAttribL{1234}dv, **390**
VertexAttribLFormat, **398**, 399, 400
VertexAttribLPointer, **403**, 435, 444, 701
VertexAttribP1ui, 390
VertexAttribP2ui, 390
VertexAttribP3ui, 390
VertexAttribP4ui, 390, 392
VertexAttribP4uiv, 392
VertexAttribPointer, 397, **403**, 412, 430, 435, 436, 701, 841
VertexBindingDivisor, **408**, 408
VertexId, 832
VertexP*uiv, 433
VertexP2ui, 433
VertexP3ui, 433
VertexP4ui, 433
VertexPointer, **405**, 406, 428, 436, 701, 840

vertices, 469
VERTICES_SUBMITTED, 43, 49, 439
VIEW_CLASS_128_BITS, 310
VIEW_CLASS_16_BITS, 310
VIEW_CLASS_24_BITS, 310
VIEW_CLASS_32_BITS, 310
VIEW_CLASS_48_BITS, 310
VIEW_CLASS_64_BITS, 310
VIEW_CLASS_8_BITS, 310
VIEW_CLASS_96_BITS, 310
VIEW_CLASS_BPTC_FLOAT, 310
VIEW_CLASS_BPTC_UNORM, 310
VIEW_CLASS_RGTC1_RED, 310
VIEW_CLASS_RGTC2_RG, 310
VIEW_COMPATIBILITY_CLASS, 724, 875
VIEWPORT, 738
Viewport, **545**, 546
VIEWPORT_BIT, 706
VIEWPORT_BOUNDS_RANGE, 546, 794
VIEWPORT_INDEX_PROVOKING_VERTEX, 497, 794
VIEWPORT_SUBPIXEL_BITS, 546, 794
ViewportArrayv, **545**, 545
ViewportIndex, 832
ViewportIndexedf, **545**, 545
ViewportIndexedfv, **545**, 545

WAIT_FAILED, 39
WaitSync, 37–39, **40**, 40, 41, 55, 56, 809
WGL_ARB_create_context, 671, 918
WGL_ARB_create_context_no_error, 941
WGL_ARB_create_context_profile, 839, 922
WGL_ARB_create_context_robustness, 926
WGL_ARB_framebuffer_sRGB, 917
WGL_ARB_pixel_format_float, 915
WGL_ARB_robustness_application_isolation, 932
WGL_ARB_robustness_share_group_isolation, 932
while, 473
Width, 597
WindowPos, **577**, 577, 579, 693
WindowPos2, 577
WindowPos3, 577
Workgroup, 828
WorkgroupId, 832
WorkgroupSize, 830, 832
WRITE_ONLY, 61, 75–77, 328

Xfb, 449, 452, 829
xfb_buffer, 449, 450, 452
xfb_offset, 114, 449, 450, 452
xfb_stride, 449, 452
XfbBuffer, 449, 453, 829
XfbStride, 449, 829, 830
XOR, 627

ZERO, 275, 600, 616, 621, 623, 625, 755
ZERO_TO_ONE, 537, 538, 543
ZOOM_X, 767
ZOOM_Y, 767