ARM® Guide to Unity

Version 2.1

Enhancing Your Mobile Games

ARM

ARM 100140_0201_00_en

ARM® Guide to Unity

Enhancing Your Mobile Games

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	05 September 2014	Non-Confidential	First release
0200-00	23 June 2015	Non-Confidential	First release
0201-00	28 July 2015	Non-Confidential	First release

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

Contents

ARM® Guide to Unity Enhancing Your Mobile Games

Preface

This preface introduces the *ARM® Guide to Unity Enhancing Your Mobile Games*.

It contains the following:

About this book

This book is designed to help you create applications and content that make the best use of Unity on mobile platforms, especially those with Mali® GPUs.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This book is for beginner to intermediate developers.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter introduces the ARM® Guide to Unity: Enhancing Your Mobile Games

Chapter 2 Optimizing Applications
This chapter describes how to optimize applications in Unity.

Chapter 3 Profiling Your Application
This chapter describes profiling your application.

Chapter 4 Optimization Lists
This chapter lists a number of optimizations for your Unity application.

Chapter 5 Global Illumination in Unity with Enlighten
This chapter describes global illumination in Unity with Enlighten.

Chapter 6 Advanced Graphics Techniques
This chapter lists a number of advanced graphics techniques that you can use.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`mono`space
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

Information published by ARM and by third parties.

See *http://infocenter.arm.com* for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

None.

Developer resources:

http://malideveloper.arm.com.

Other publications

Relevant documents published by third parties:

OpenGL ES 1.1 Specification at *http://www.khronos.org*.
OpenGL ES 2.0 Specification at *http://www.khronos.org*.
OpenGL ES 3.0 Specification at *http://www.khronos.org*.
OpenGL ES 3.1 Specification at *http://www.khronos.org*.
Unity Scripting Reference at *Unity*.
GPU Gems: Programming Techniques, Tips, and Tricks for Real-time Graphics by Randima Fernando (Series Editor).
GPU Pro: Advanced Rendering Techniques by Wolfgang Engel (Editor).

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *ARM® Guide to Unity Enhancing Your Mobile Games*.
- The number ARM 100140_0201_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter introduces the ARM® Guide to Unity: Enhancing Your Mobile Games

It contains the following sections:

1.1 About Unity

Unity is a software platform that enables you to create and distribute 2D games, 3D games, and other applications.

This book is intended to help you create applications and content that make the best use of Unity on mobile platforms, especially those with Mali GPUs. It describes techniques and best practices that you can use to improve the performance of your applications.

Note

Unless otherwise noted, the techniques described here also work on other platforms.

1.2 About Mali GPUs

Mali GPUs are designed for mobile or embedded devices. Mali GPUs are divided into the following families:

The Mali Utgard GPU family

The Mali Utgard family of GPUs have a vertex processor and one or more fragment processors. They are used for graphics only applications with OpenGL ES 1.1 and 2.0. The Mali Utgard family includes the following Mali GPUs:

- Mali-300.
- Mali-400 MP.
- Mali-450 MP.

The Mali Midgard GPU family

The Mali Midgard family of GPUs have unified shader cores that perform vertex, fragment, and compute processing. They are used for graphics and compute applications with OpenGL ES 1.1 to OpenGL ES 3.1, and OpenCL 1.1 Full Profile.

The Mali Midgard family includes the following Mali GPUs:

- Mali-T604.
- Mali-T622.
- Mali-T624.
- Mali-T628.
- Mali-T720.
- Mali-T760.
- Mali-T820.
- Mali-T830.
- Mali-T860.
- Mali-T880.

1.3 About optimization in Unity

Graphics is about making things look good. Optimization is about making things look good with the least computational effort. This is especially important for mobile devices that keep power consumption low by limiting computing power and memory bandwidth.

1.4 About the Ice Cave demo

The Ice Cave demo is a demonstration application created by ARM that uses a number of optimized techniques to produce high quality visual content for mobile devices.

This document describes the graphical techniques used in the Ice Cave demo, and solutions to problems that were encountered during the development of the project.

Chapter 2
Optimizing Applications

This chapter describes how to optimize applications in Unity.

It contains the following sections:

2.1 The optimization process

Optimization is the process of taking an application and making it more efficient. For graphical applications, this typically means modifying the application to make it faster.

For example, a game with a low frame rate means it appears jumpy. This gives a bad impression and can make a game difficult to play. You can use optimization to improve the frame rate of a game making it a better, smoother experience.

To optimize your code, use the optimization process. This is an iterative process that guides you through finding and removing performance problems.

The optimization process consists of the following steps:

1. Take measurements of your application with a profiler.
2. Analyze the data to locate the bottleneck.
3. Determine the relevant optimization to apply.
4. Verify that the optimization works.
5. If the performance is not acceptable return to step 1 and repeat the process.

The following is an example of the optimization process:

1. If you have a game that does not have the performance you require, you can use the Unity profiler to take measurements.
2. Use the Unity profiler to analyze the measurements so you can isolate and identify the source of the performance problem.
3. The problem with the game is, for example, rendering too many vertices.
4. Reduce the number of vertices in your code.
5. Execute the game again to ensure the optimization worked.

If you do this and the game still does not perform as expected, restart the process by profiling the application again to find out what else is causing problems.

Expect to repeat this process a number of times. Optimization is an iterative process where you might find performance problems in a number of different areas.

2.2 Unity quality settings

It is useful to know the Unity quality settings to ensure you select the correct settings for your application.

Unity has a number of options that alter the image quality of your game. Some of these options have a high computational cost and can have a negative impact on the performance of your game.

The following figure shows quality settings in the Inspector:

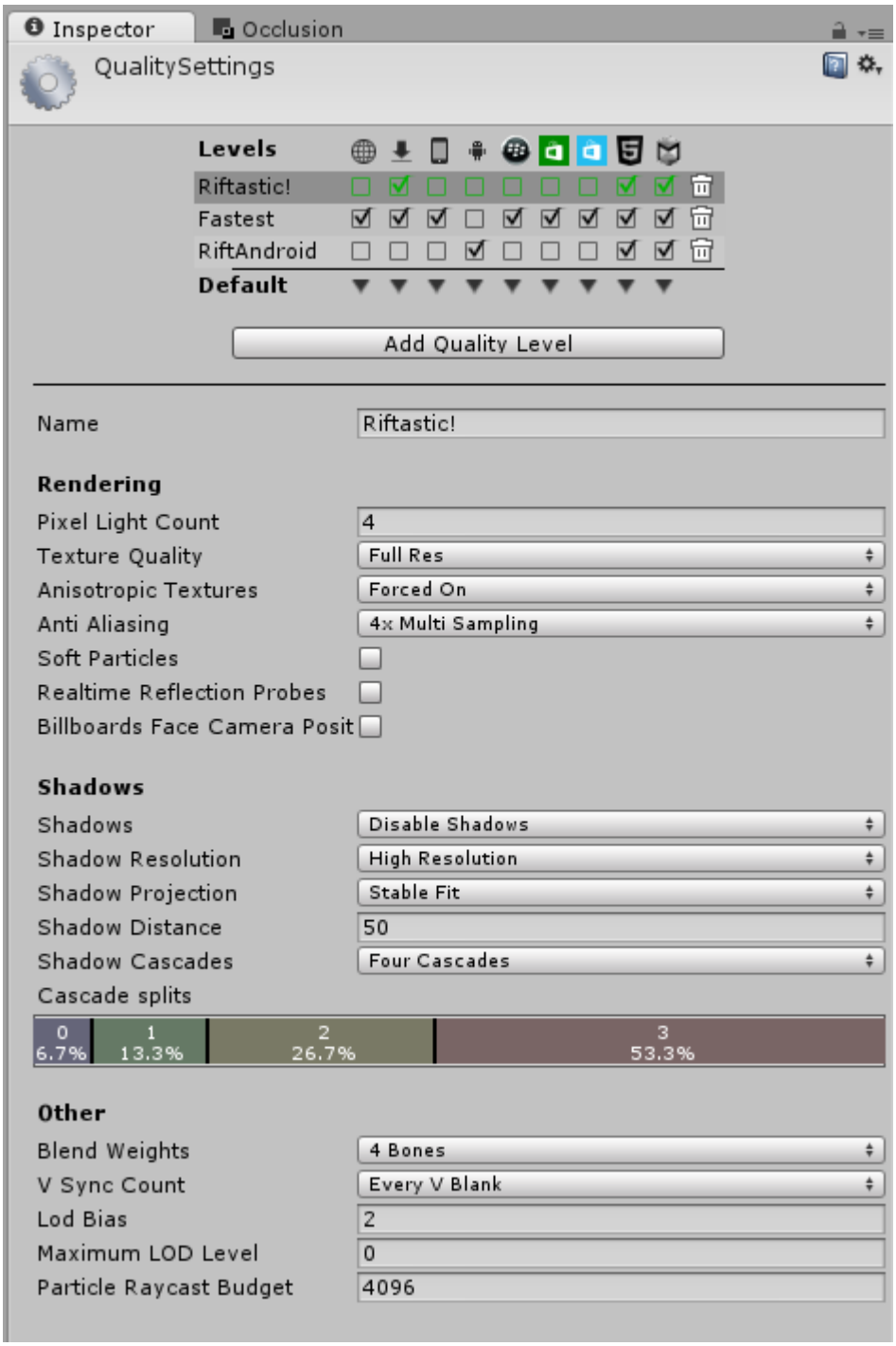

Figure 2-1 Quality settings

There are a number of options that can increase the image quality of your game with only a small trade-off in performance. For example, if the frame rate of your game is low, the GPU might be processing too much information when performing a complex graphical effect. You can perform less complex versions of graphical effects, such as shadows and lighting, for a relatively small impact on the graphical quality. Simpler effects can reduce the load on the GPU significantly, providing a higher frame rate.

The default settings for lighting can sometimes be too complex for a mobile device, so some games written for mobile platforms avoid complex techniques or use game-specific techniques. This might involve techniques such as pre-baking lighting into light maps or projecting textures instead of casting shadows.

In **Project Settings** > **Quality** there are a number of options that can have a large impact on the performance of your game:

Pixel light count

Pixel light count is the number of lights that can affect a given pixel. A high pixel light count requires a large number of calculations. Most games can use very few dynamic and real-time lights with minimal impact on image quality. Consider using techniques such as light maps and projected textures in your game, if lighting is causing performance problems.

Texture quality

Texture quality can load the GPU but it typically does not cause performance problems. Reducing texture quality can negatively impact the visual quality of your games, so only reduce the quality if you must. In the Ice Cave demo, **Texture quality** is set to full resolution.
If textures are causing performance problems, try using mipmapping. Mipmapping reduces compute and bandwidth requirements without impacting image quality.

Anti-Aliasing

Anti-Aliasing is an edge-smoothing technique that blends the pixels around triangle edges. This provides a noticeable improvement to the visual quality of your game. There are several methods of anti-aliasing, but in this case the toggle is for *Multi-Sampled Anti-Aliasing* (MSAA). 4x MSAA is very low cost operation on Mali GPUs, so always use it if possible.

Soft Particles

Soft Particles requires rendering to a depth texture or rendering in deferred mode. This increases the load on the GPU, but can be worth it in terms of achieving realistic visuals on your particles. On mobile platforms, rendering to and reading from a depth texture uses up valuable bandwidth, and rendering using a deferred path means you have no access to MSAA. Consider whether soft particles are important enough to your game to use them.

Anisotropic Textures

Anisotropic Textures is a technique that removes distortion from textures drawn at high gradients. This improves the image quality but it is an expensive technique. Avoid using this technique unless the distortion is especially noticeable.

Shadows

Shadows can be computationally intensive if they are high quality. If shadows cause performance problems, try simple shadows or switch them off. If shadows are important in your game, consider using simple dynamic shadowing techniques such as projected textures.

Real-time reflection probes

The **Real-time reflection probes** option can have a significant negative impact on the runtime performance.

When a reflection probe is rendered, every face of the cubemap is rendered separately by a camera at the origin of the probe. If inter-reflections are considered, this process takes place for every reflection bounce level. In the case of glossy reflections, the cubemap mipmaps are also used to apply a blurring process.

The following factors influence the rendering of the cubemap:

Cubemap resolution

Higher resolution cubemaps increase rendering time. Use the lowest resolution cubemap possible for the quality you require.

Culling Mask

Use the culling mask when rendering the cubemap to avoid rendering any geometry not relevant in the reflections.

Cubemap Updating

the **Refresh Mode** option defines the update frequency for a cubemap:

- The **Every Frame** option renders the cubemap every frame. This is the most computationally expensive option, so avoid using it unless you require it.
- The **On Awake** option renders the cubemap at runtime one time, when the scene starts.
- The **Via Scripting** option provides you with control over when the cubemap is updated. With this option, you can limit the use of runtime resources by specifying the conditions when an update takes place.

Chapter 3
Profiling Your Application

This chapter describes profiling your application.

It contains the following sections:

3.1 About profiling

You profile your application to find performance bottlenecks. When you have identified these, optimizing in these areas improves your application performance.

You can profile your Unity application with the following tools:

- Unity Profiler.
- Unity Frame Debugger.
- ARM Mali Graphics Debugger.
- ARM DS-5 Streamline.

3.2 Profiling with the Unity profiler

The Unity profiler provides detailed per-frame performance data in a series of charts to help you find the bottlenecks in your game.

If you click in a chart, you see a vertical slice and this selects a single frame. You can read information from this frame in the display panel at the bottom of the screen. If you click on a different chart without modifying the frame selection, the panel shows data from the profiler you have selected.

The following figure shows the Unity profiler:

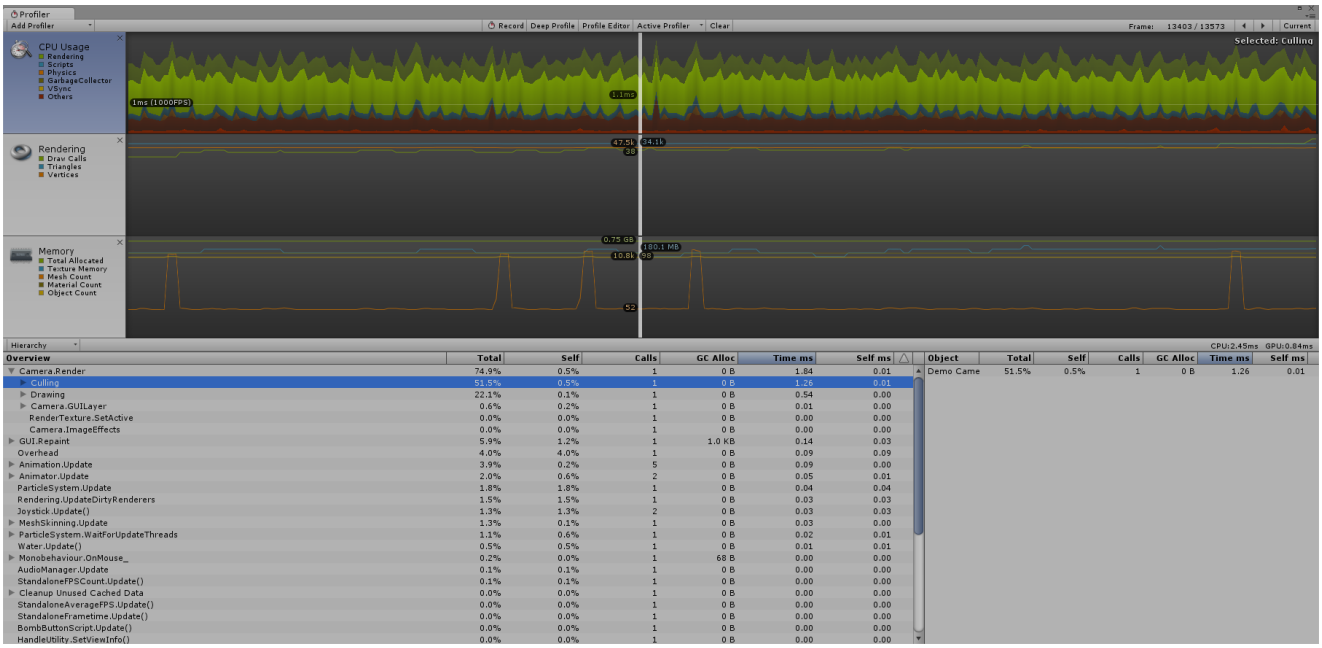

Figure 3-1 Unity profiler

The Unity profiler provides the following functions:

CPU Usage profiler

The CPU Usage profiler chart shows a breakdown of CPU utilization, highlighting different components, such as rendering, scripts, or physics. If you select a frame on the chart, the panel displays the functions that most contributed to that particular frame, in time taken, number of calls or memory allocations.

Concentrate on the functions that spend more time executing or allocate too much memory.

Note

In a multi-processor system, the values are averaged.

Rendering profiler

The Rendering profiler chart shows the number of draw calls, triangles, and vertices rendered in your scene. Selecting a frame on the chart displays more information about batching, textures, and memory consumption.

Look at the number of draw calls, triangles, and vertices that your scene renders. These are the most important numbers in mobile platforms.

Memory profiler

The Memory profiler chart displays the amount of memory allocated and the amount of resources used by the game, such as meshes or materials. Selecting a frame on the chart displays the memory consumption of your assets, the graphics and audio subsystems, or of profiler data itself.

Memory is limited on mobile platforms, so you must monitor the requirements of your game during its lifetime and check the number of resources in use. Some techniques, if applied incorrectly, can cause a large number of new objects to be created. For example, a texture atlas applied incorrectly can lead to the creation of a large number of new material objects.

The Add Profiler function

Add Profiler is an option on the drop-down menu in the top left corner of the profiler window. It enables you to add more charts to the profiler window, for example, CPU usage, rendering, or memory.

The Profiler.BeginSample() and Profiler.EndSample() methods

The Unity profiler enables you to use the `Profiler.BeginSample()` and `Profiler.EndSample()` methods. You can mark a region in your script, attach a custom label to it, and this region appears in the profiler hierarchy as a separate entry. Doing this enables you to obtain information about a specific piece of code without the compute and memory overhead of the `Deep Profile` option.

```
void Update()
{
    Profiler.BeginSample("ProfiledSection");
    [...]
    Profiler.EndSample();
}
```

The following figure shows a profiled section:

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	95.7%	95.7%	1	0 B	15.06	15.06
▼ ProfiledSectionTest.Update()	2.1%	0.0%	1	0 B	0.33	0.00
ProfiledSection	2.1%	2.1%	1	0 B	0.33	0.33

Figure 3-2 Profiled section

3.3 The Unity Frame Debugger

The Frame Debugger is a profiling tool that enables you to trace draw calls on a frame-by-frame basis.

The Frame Debugger is available from the **Window** menu.

The left pane shows the tree of draw calls issued in the frame.

In the right pane it shows additional information related to the selected draw call, such as geometry details and the shader that draws it.

The following figure shows a Phoenix object in the Frame Debugger:

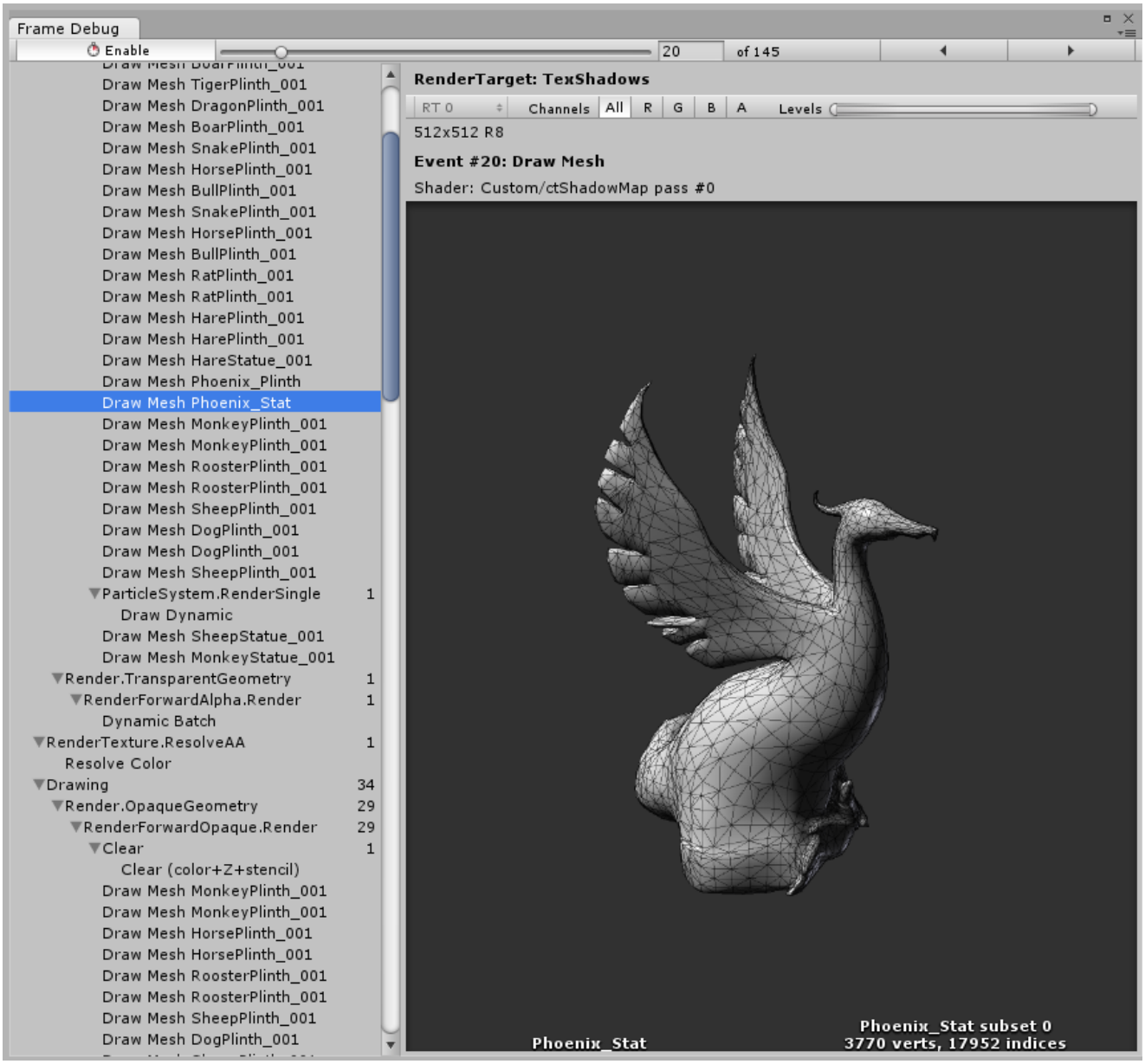

Figure 3-3 Frame Debugger

If the camera is rendering to a target at the selected draw call, you can visualize the rendered texture in the Game View.

The following figure shows a Phoenix object in the Game View:

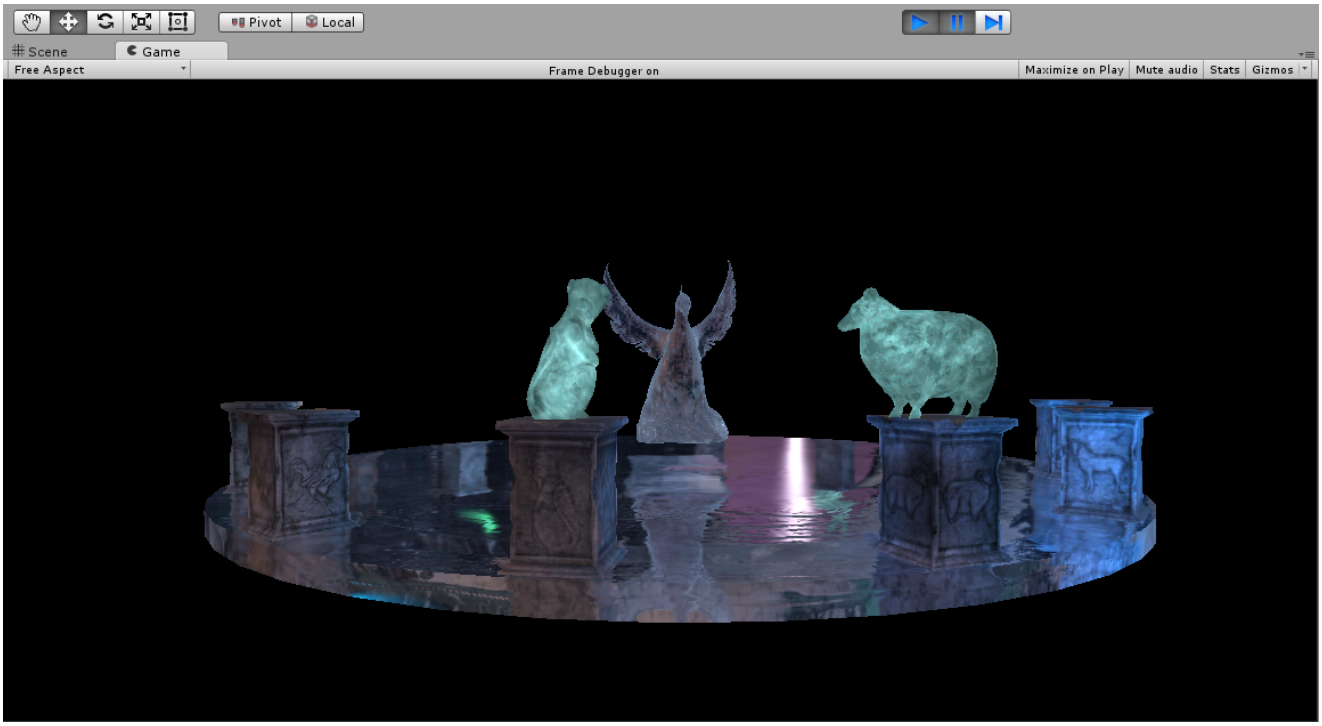

Figure 3-4 Frame Debugger Game View

For a description of how to increase the performance of a game by using the Frame Debugger to help optimize the rendering order of objects, see *4.2.9 Specify the rendering order* on page 4-50.

Chapter 4
Optimization Lists

This chapter lists a number of optimizations for your Unity application.

It contains the following sections:

4.1 Application processor optimizations

The following list describes application processor optimizations:

Use coroutines instead of Invoke()

The `Monobehaviour.Invoke()` method is a fast and convenient way to call a method in a class with a time delay, but it has the following limitations:

- It uses reflection in C# to find the method to call, this can be slower than calling the method directly.
- There are no compile-time checks on the method signature.
- You cannot supply additional parameters.

The following code shows the `Invoke()` function:

```
public void Function()
{
    [...]
}
Invoke("Function", 3.0f);
```

An alternative method is to use coroutines. A coroutine is a function of type `IEnumerator` that can return control to Unity with a special `yield return` statement. You can call the function again later and it resumes where it left off earlier.
You can call coroutines through the `MonoBehaviour.StartCoroutine()` method:

```
public IEnumerator Function(float delay)
{
    yield return new WaitForSeconds(delay);
    [...]
}
StartCoroutine(Function(3.0f));
```

Changing from the `Monobehaviour.Invoke()` method to using coroutines provides more flexibility over the parameters passed to the functions dealing with animation states.

Using coroutines for relaxed updates

If your game requires an action every specific time interval, try launching a coroutine in the `MonoBehaviour.Start()` callback, instead of performing an action every frame in the `MonoBehaviour.Update()` callback. For example:

```
void Update()
{
    // Perform an action every frame
}

IEnumerator Start()
{
    while(true)
    {
        // Do something every quarter of second
        yield return new WaitForSeconds(0.25f);
    }
}
```

Note

An example of this technique is to spawn enemies at an irregular interval. Use an infinite loop inside the coroutine that spawns an enemy and generates a random number. Pass the random number to the `WaitForSeconds()` function.

Avoid hard-coded strings for tags

Avoid hard-coded values for tags because they restrict the scalability and robustness of your game. For example, with tag names, if you refer to the names directly by strings, you cannot easily modify them and you are potentially exposed to spelling errors. For example:

```
if(gameObject.CompareTag("Player"))
{
    [...]
}
```

You can improve this by implementing a special class for tags that exposes public constant strings. For example:

```
public class Tags
{
    public const string Player ="Player";
    [...]
}

if(gameObject.CompareTag(Tags.Player))
{
    [...]
}
```

Note

You can use a Tags class with public constant strings to assist adding new tags in a consistent and scalable manner.

Reduce the number of physics calculations by changing the fixed time step

You can reduce the computational load of physics calculations by changing the fixed time step. Typically, most physics calculations take place at a fixed time step and you can increase or decrease the length of this step.

Increasing the time step decreases the load on the application processor but reduces the accuracy of physics calculations.

You can access the time manager from the main menu: **Edit** > **Project Settings** > **Time**.

The following figure shows the time manager:

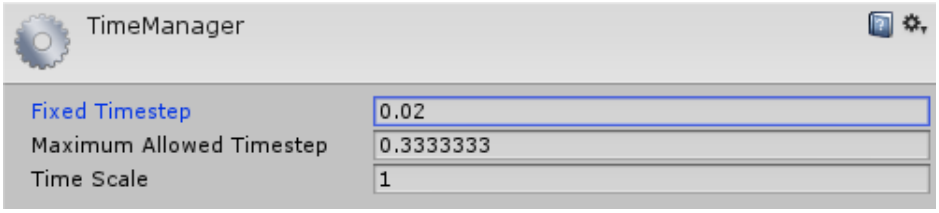

Figure 4-1 Fixed timestep settings

Remove empty callbacks

If your code includes empty definitions for functions such as `Awake()`, `Start()`, or `Update()`, remove them. There is an overhead associated with these because the engine still attempts to access them even though they are empty. For example:

```
// Remove the following empty definition

void Awake()
{

}
```

Avoid using GameObject.Find() in every frame.

`GameObject.Find()` is a function that iterates through every object in the scene. This can cause a significant increase in the main thread size if it is used in the incorrect part of your code. For example:

```
void Update()
{
    GameObject playerGO = GameObject.Find("Player");
    playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

A better technique is to call `GameObject.Find()` on startup and cache the result, for example in the `Start()` or `Awake()` function:

```
private GameObject _playerGO = null ;

void Start()
{
    _playerGO = GameObject.Find("Player");
}

void Update()
{
    _playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

Another alternative is to use `GameObject.FindWithTag()`:

```
void Update()
{
    GameObject playerGO = GameObject.FindWithTag("Player");
    playerGO.transform.Translate(Vector3.forward * Time.deltaTime);
}
```

Note

Use a dedicated class called `LocatorManager` that performs all the object retrievals immediately when the scene finishes loading. Other classes can use this as a service so objects are not retrieved multiple times.

Use the StringBuilder class to concatenate strings

When concatenating complex strings, use the `System.Text.StringBuilder` class. This is faster than the `string.Format()` method and uses less memory than concatenation with the plus operator:

```
// Concatenation with the plus operator
string str = "foo" + "bar";

// String.Format() method
string str = string.Format("{1}{2}", "foo", "bar");
```

The `System.Text.StringBuilder` class:

```
// StringBuilder class
using System.Text;

StringBuilder strBld = new StringBuilder();
strBld.Append("foo");
strBld.Append("bar");
string str = strBld.toString();
```

The following figure shows string concatenations:

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ StringConcatenationTest.Start()	99.8%	0.0%	1	1.12 GB	2488.86	0.33
▼ StringConcatenationTest.StringFormatMethod()	51.0%	0.2%	1	0.56 GB	1273.01	6.78
▶ String.Format()	50.8%	0.3%	10000	0.56 GB	1266.22	7.51
▼ StringConcatenationTest.PlusConcatenation()	48.4%	0.2%	1	0.56 GB	1208.21	6.53
▶ String.Concat()	48.2%	0.5%	10000	0.56 GB	1201.68	13.15
▼ StringConcatenationTest.StringBuilderClass()	0.2%	0.2%	1	256.2 KB	7.31	6.97
▶ StringBuilder.Append()	0.0%	0.0%	10000	256.1 KB	0.32	0.12
▶ StringBuilder..ctor()	0.0%	0.0%	1	0 B	0.00	0.00
▶ StringBuilder.ToString()	0.0%	0.0%	1	0 B	0.01	0.00

Figure 4-2 String concatenations

Use the CompareTag() method instead of the tag property

Use the `GameObject.CompareTag()` method instead of the `GameObject.tag` property. The `CompareTag()` method is faster and does not allocate extra memory:

```
GameObject mainCamera = GameObject.Find("Main Camera");

// Gameobject.tag property
if(mainCamera.tag == "MainCamera")
{
    // Perform an action
}

// Gameobject.CompareTag() method
if(mainCamera.CompareTag("MainCamera"))
{
    // Perform an action
}
```

The following figure shows the use of `CompareTag()`:

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ TagComparisonTest.Start()	97.2%	0.1%	1	3.6 MB	127.48	0.26
▼ TagComparisonTest.TagProperty()	68.0%	59.6%	1	3.6 MB	89.27	78.14
▼ GameObject.get_tag()	7.9%	1.0%	100000	3.6 MB	10.44	1.32
GC.Collect	6.9%	6.9%	4	0 B	9.12	9.12
▼ String.op_Equality()	0.5%	0.3%	100000	0 B	0.69	0.50
String.Equals()	0.1%	0.1%	100000	0 B	0.18	0.18
▶ TagComparisonTest.CompareTagMethod()	28.9%	28.4%	1	0 B	37.94	37.27
GameObject.Find()	0.0%	0.0%	1	0 B	0.00	0.00

Figure 4-3 Compare tag

Use object pools

If your game has many objects of the same kind that are created and destroyed at runtime, you can use the design pattern *object pool*. This design pattern avoids the performance penalty of allocating and freeing many objects dynamically.

If you know the total number of objects that you require, you can create them all immediately and disable the objects that are not immediately required. When a new object is required, search the pool for the first unused one and enable it.

When an object is not required anymore, you can return it to the pool. This means resetting the object to a default starting state and disabling it.

This technique can be used with objects such as enemies, projectiles, and particles. If you do not know the exact number of objects that you require, do a test to find how many are used and create a pool slightly bigger than the number you find.

Note

Use object pools for enemies and bombs. This restricts the allocation of those objects to the loading phase of the game.

Cache component retrievals

Cache the component instance returned by `GameObject.GetComponent<Type>()`. The function call involved is quite expensive.

Properties such as `GameObject.camera`, `GameObject.renderer` or `GameObject.transform` are shortcuts to the corresponding `GameObject.GetComponent<Camera>()`, `GameObject.GetComponent<Renderer>()`, and `GameObject.GetComponent<Transform>()`:

```
private Transform _transform = null;

void Start()
{
    _transform = GameObject.GetComponent<Transform>();
}

void Update()
{
    _transform.Translate(Vector3.forward * Time.deltaTime);
}
```

Consider caching the return value of `Transform.position`. Even if it is a C# getter property, there is overhead associated with an iteration over the transform hierarchy to calculate the global position.

Note

In Unity 5 and higher, the transform component is automatically cached.

Use OnBecameVisible() and OnBecameInvisible() callbacks

Callbacks such as `MonoBehaviour.OnBecameVisible()` and `MonoBehaviour.OnBecameInvisible()` notify your scripts if their associated game objects become visible or invisible on screen.

These calls enable you to, for example, disable computational heavy code routines or effects when a game object is not rendered on screen.

Use sqrMagnitude for comparing vector magnitudes

If your application requires the comparison of vector magnitudes, use `Vector3.sqrMagnitude` instead of `Vector3.Distance()` or `Vector3.magnitude`.

`Vector3.sqrMagnitude` sums the squared components without calculating the root, but this is useful for comparisons. The other calls use a computationally expensive square root.

The following code shows the three different techniques used in comparisons of two positions in space:

```
// Vector3.sqrMagnitude property
if ((_transform.position - targetPos).sqrMagnitude < maxDistance * maxDistance)
{
    // Perform an action
}

// Vector3.Distance() method
if (Vector3.Distance(transform.position, targetPos) < maxDistance)
{
    // Perform an action
}

// Vector3.magnitude property
if ((_transform.position - targetPos).magnitude < maxDistance)
{
    // Perform an action
}
```

Use built-in arrays

If you know the size of an array in advance, use the built-in arrays.

`ArrayList` and `List` classes have more flexibility because they grow in size the more elements you insert, but they are slower than the built-in arrays.

Use planes as collision targets

If your scene only requires particle collisions with planar objects such as floors or walls, change the particle system collision mode to `Planes`. Changing the setting to use planes reduces the computations required. In this mode you can provide Unity with a list of empty GameObjects to act as the collider planes.

The following figure shows collision settings:

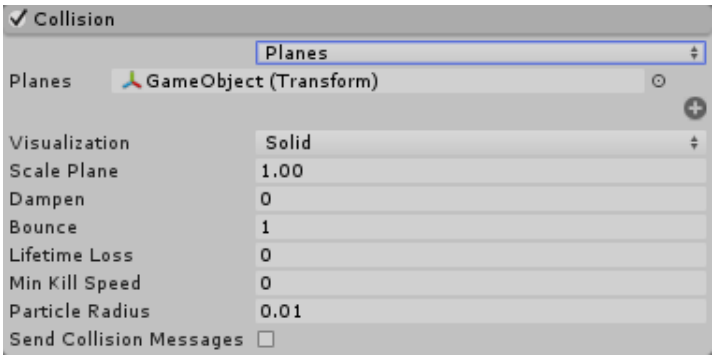

Figure 4-4 Collision settings

Use compound primitive colliders rather than mesh colliders

Mesh colliders are based on the real geometry of an object. These are very accurate for collision detection but are computationally expensive.

You can combine shapes such as boxes, capsules, or spheres into a compound collider that mimics the shape of the original mesh. This enables you to achieve similar results with much lower computational overhead.

4.2 GPU optimizations

This section lists GPU optimizations.

This section contains the following subsections:

4.2.1 Miscellaneous GPU optimizations

The following list describes miscellaneous GPU optimizations:

Use static batching

Static batching is a common optimization technique that reduces the number of draw calls and this, in turn, reduces application processor utilization.
Dynamic Batching is performed transparently by Unity, but it cannot be applied to objects made up of a large number of vertices, because the computational overhead becomes too large.
Static Batching can work on objects made up of a large number of vertices, but the batched objects must not move, rotate, or scale during rendering.
To enable Unity to group objects for static batching, mark them as static in the Inspector.
The following figure shows static batching settings:

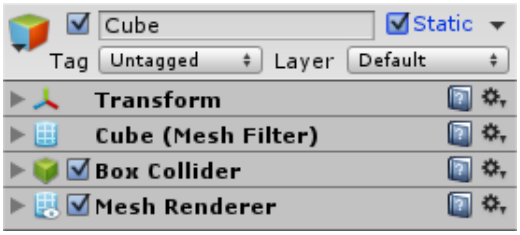

Figure 4-5 Static batching settings

Use 4x MSAA

ARM Mali GPUs can do 4x multi-sample anti-aliasing with very low computational overhead.
You can enable 4x MSAA in the Unity `Quality Settings`.
The following figure shows the MSAA settings:

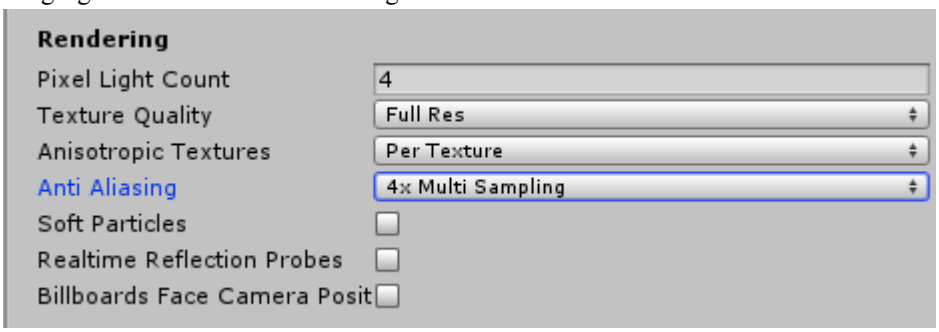

Figure 4-6 MSAA settings

Use level of detail

Level of detail (LOD) is a technique where the Unity engine renders different meshes for the same object depending on the distance from the camera.

Geometry is more detailed when the object is close to the camera. The detail level is reduced as the object moves away from the camera and at the furthest distance you can use a planar billboard.

You must set up LOD groups properly to manage the meshes to use and the associated distance ranges.

To access the setup of LOD groups, select: **Add Component** > **Rendering** > **LOD Group**.

In Unity 5 you can set a `Fade Mode` for each LOD level to blend to contiguous LODs. This smooths the transition between them. Unity calculates a blending factor according to the screen size of the object and passes it to your shader for blending. You must implement the geometry blending in a shader.

The following figure shows the LOD group settings:

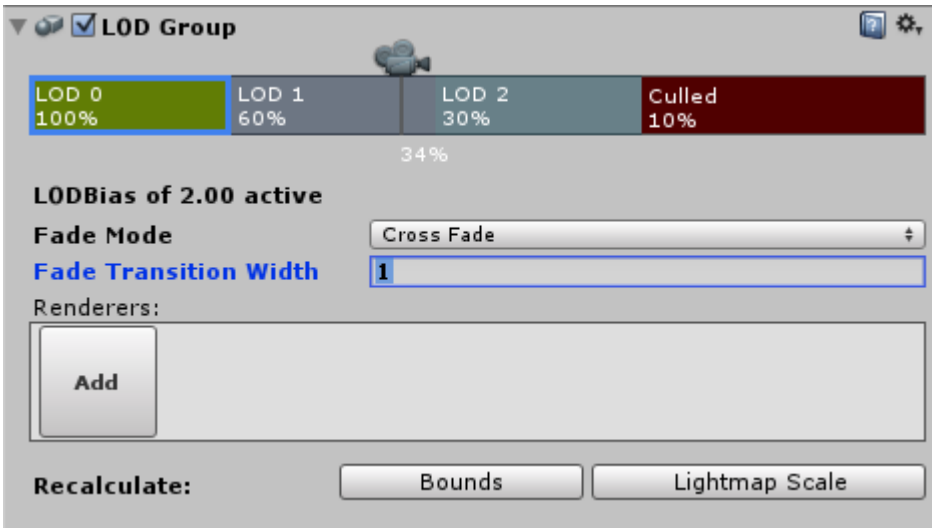

Figure 4-7 LOD group settings

Avoid mathematical functions in custom shaders

When you are writing custom shaders, try to avoid using expensive built-in mathematical functions such as:

- `pow()`.
- `exp()`.
- `log()`.
- `cos()`.
- `sin()`.
- `tan()`.

4.2.2 Lightmaps and light probes

Runtime lighting calculations are computationally expensive. A popular technique to reduce the computation requirements called lightmapping pre-computes the lighting calculations and bakes them into a texture called a lightmap.

This means you lose the flexibility of a fully dynamically lit environment, but you do get very high quality images without impacting performance.

To bake the resulting lighting in a static lightmap

- Set the geometry that receives the lighting to **static**.
- Set the **Baking** option in the light to **Baked** instead of **Realtime**.
- Tick the **Baked GI** option in Scene tab of the Lightmapping window.

To see the resulting lightmap:

- Select the geometry.
- Open the Lighting window by selecting **Window** > **Lighting**.
- Press the **Object** button.
- Select **Baked Intensity lightmap** in the preview options.

If the Continuous Baking option is selected Unity bakes the lightmap and updates the scene in the Editor in seconds.

A quick way to check that the lightmap has set up correctly is to run the game in the Editor and disable the light. If the lighting is still there, the lightmap has been created correctly and it is in use.

The following figure shows an intensity lightmap in the Lighting tab.

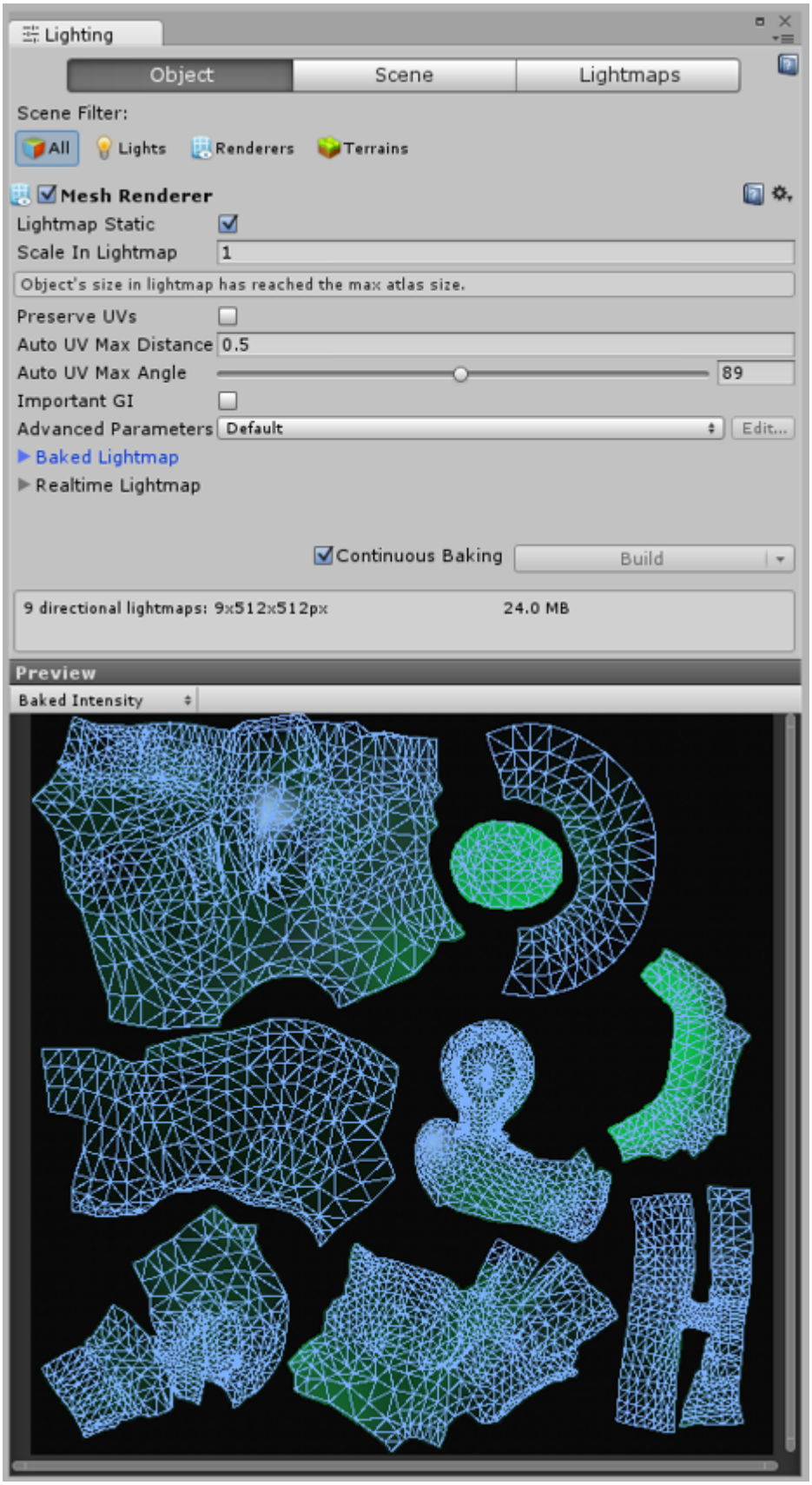

Figure 4-8 The intensity lightmap

The following figure shows the editor displaying lighting from a green light at the end of a cave. This lighting is generated with a static lightmap.

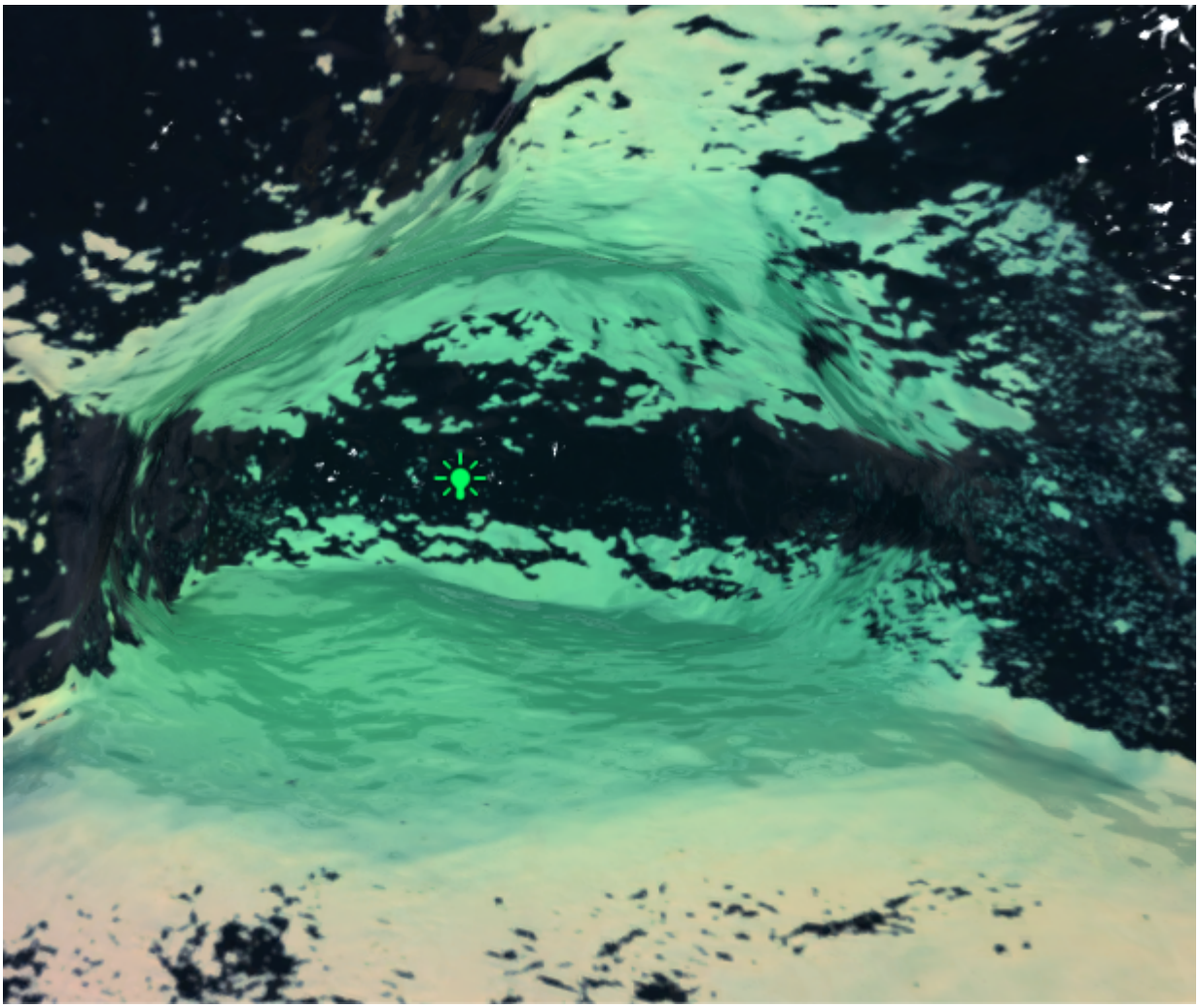

Figure 4-9 Adding a light to bake a static lightmap

The following figure shows the result of the static lightmap in the Ice Cave demo.

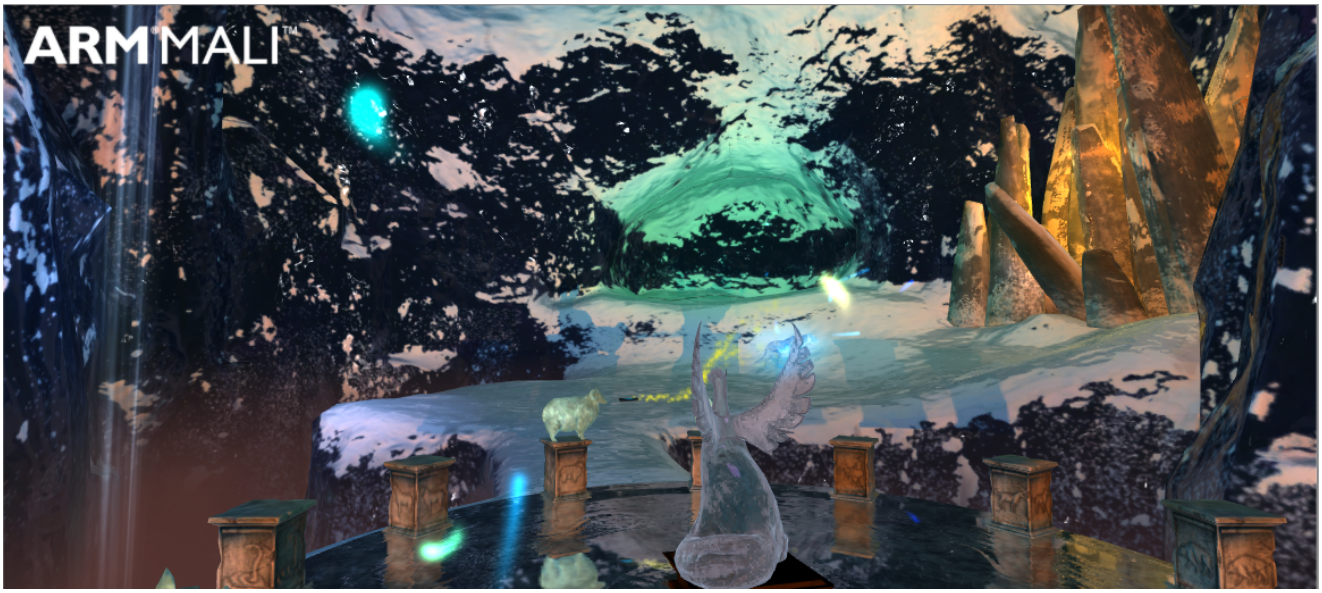

Figure 4-10 Lightmapped cave

Setting up lightmapping

To prepare an object for lightmapping you must have:

- A model in your scene with lightmap UVs.
- The model must be marked as **lightmap static**.
- There must be a light in range of the model.
- The **Baking type** of the light must be set to `Baked`.

Note

Only the static objects in your scene are lightmapped. They are not likely to be perfect, so experiment to see what works best for your game.

Objects that are not marked as static are not placed in the lightmap. Selecting a renderer provides you with a number of settings and enables you to set whether its lightmap is static or not.

Open the Lighting window from the main menu of the editor window and select **Window** and Lightmapping. There are three buttons:

- **Object**.
- **Scene**.
- **Lightmaps**.

The following figure shows lightmap options:

Figure 4-11 Lightmap options

Object

Clicking the **Object** button enables you to change settings related to lightmapping on the object you have selected in the hierarchy. This enables you to modify the object settings that impact the lightmapping process. Select a light, and you can change a number of options:

- **Baked Only** enables the light at baking time and disables it at runtime.
- **Baked** If **Baked GI** is selected, the light is baked.
- **Realtime** The light works for both pre-computed real-time GI and without GI.
- **Realtime Only** disables the light at baking time and enables it at runtime.
- **Mixed** The light is baked, but it is still present at runtime to give direct lighting to non-static objects.

Setting the majority of lights to **Baked** ensures the number of calculations at runtime is relatively low.

Scene

The **Scene** tab contains settings that apply to the whole scene. You can enable and disable the **Pre-computed Realtime GI** and **Baked GI** features in this tab.
In the **Environment Lighting** section, there are a number of options that allow you to define several factors influencing the environment lighting, such as the Skybox, the type of Ambient Source, and the Ambient Intensity:

- The **Reflection Bounces** option is the most important from the performance point of view. **Reflection Bounces** defines the number of inter-reflections between reflective objects, that is, the number of bake times for the probe that sees the objects. This option can have a large negative impact on performance if the reflection probes are updated at runtime. Only set the number of bounces higher than one if the reflective objects shall be clearly visible in the probes.
- In the **Precomputed Realtime GI** tab the **CPU Usage** option defines the amount of processor time that is spent evaluating GI at runtime. A higher value for **CPU Usage** results in faster reactions from the lighting, but might affect frame rate. The impact on the performance is lower in multiprocessor systems.
- The **Baked GI** tab contains an option where you can set the lightmap texture to be compressed. Compressing lightmap textures requires less storage space and less bandwidth at runtime but the compression process can add artifacts to the texture.
- In the **General GI** tab, be careful with the **Directional Mode** option. If you cannot use deferred lighting with dual lightmaps, another technique is to use directional lightmaps. These enable you to use normal mapping and specular lighting without real-time lights. Use directional lightmaps if normal mapping must be preserved but dual lightmaps are not available. This is typically the case on mobile devices. When **Directional Mode** is set to **Directional** an additional lightmap is created to store the dominant direction of incoming light. As a result this mode requires about twice as much storage space.
- In **Directional Specular** mode, additional data is stored for specular reflection and normal maps. In this case the storage requirements increase four times.
- The **Lightmaps** tab enables you to set and locate the lightmap asset file used for the scene. To access the Lightmap Snapshot box the **Continuous Baking** option must be unchecked.

The following figure shows lightmaps in the **Lighting** tab:

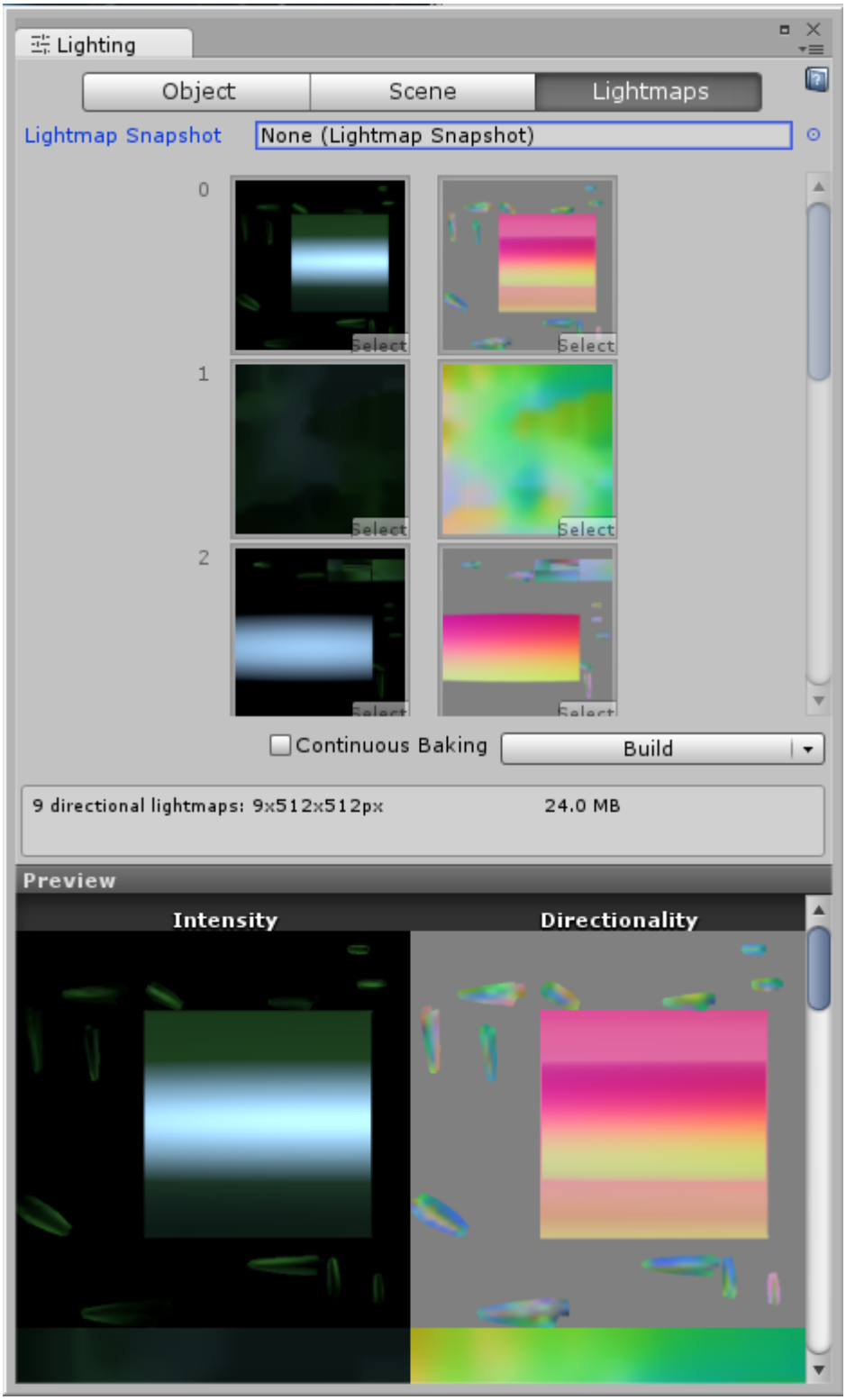

Figure 4-12 Lightmaps in the Lighting tab

Use directional lightmaps

If you cannot use deferred lighting with dual lightmaps, another technique is to use directional lightmaps. These enable you to use normal mapping and specular lighting without real-time lights.

Use directional lightmaps if normal mapping must be preserved, but dual lightmaps are not available. This is typically the case on mobile devices.

Note

This technique requires more video memory because it computes a second set of lightmaps to store directional information.

Use light probes for dynamic objects in your game

Light probes enable you to add some dynamic lighting to lightmapped scenes.

The following figure shows light probe settings:

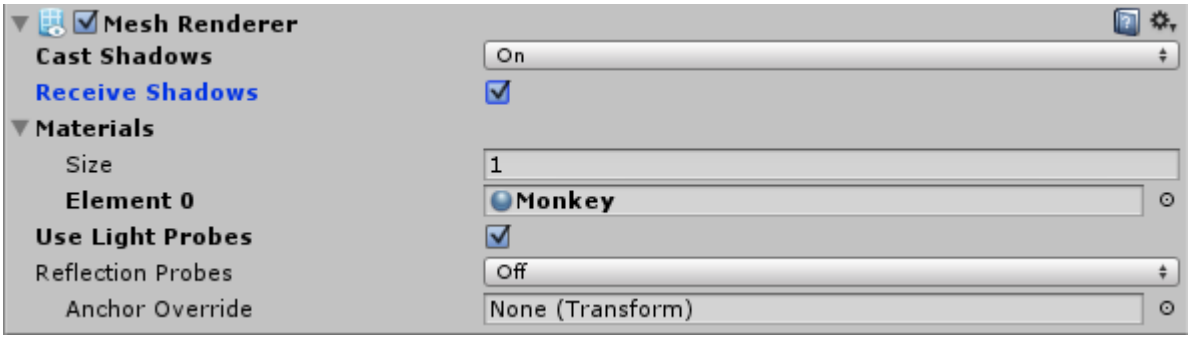

Figure 4-13 Light probe settings

Light probes take a sample, or probe, of the lighting in an area. If the probes form a volume, or cell, the lighting is interpolated between these probes depending on their position within the cell.

The following figure shows light probes:

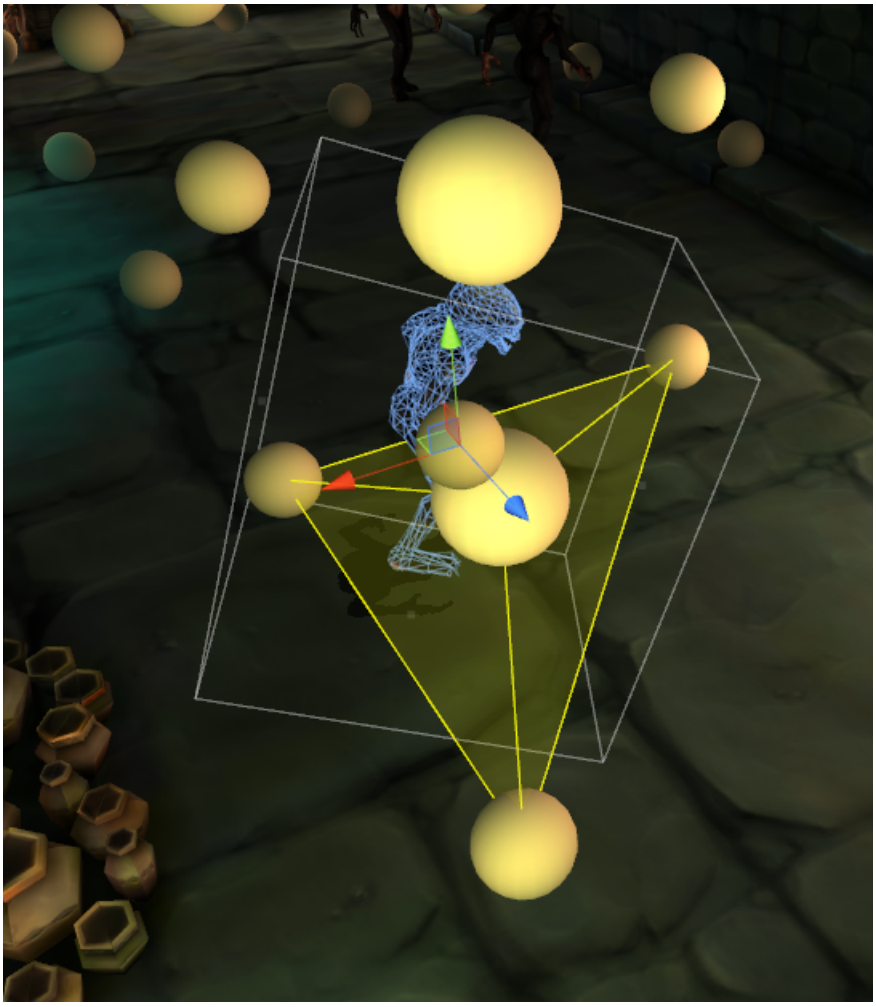

Figure 4-14 Light probes

The more probes there are, the more accurate the lighting is. You do not typically require many light probes because there is interpolation between probes. You require more light probes in areas where there are large changes in light color or intensity.

The lighting at any position can then be approximated by interpolating between the samples taken by the nearest probes.

Take care placing the light probes and mark the meshes you want to be influenced by them with the `Use Light Probes` option.

The following figure shows multiple light probes:

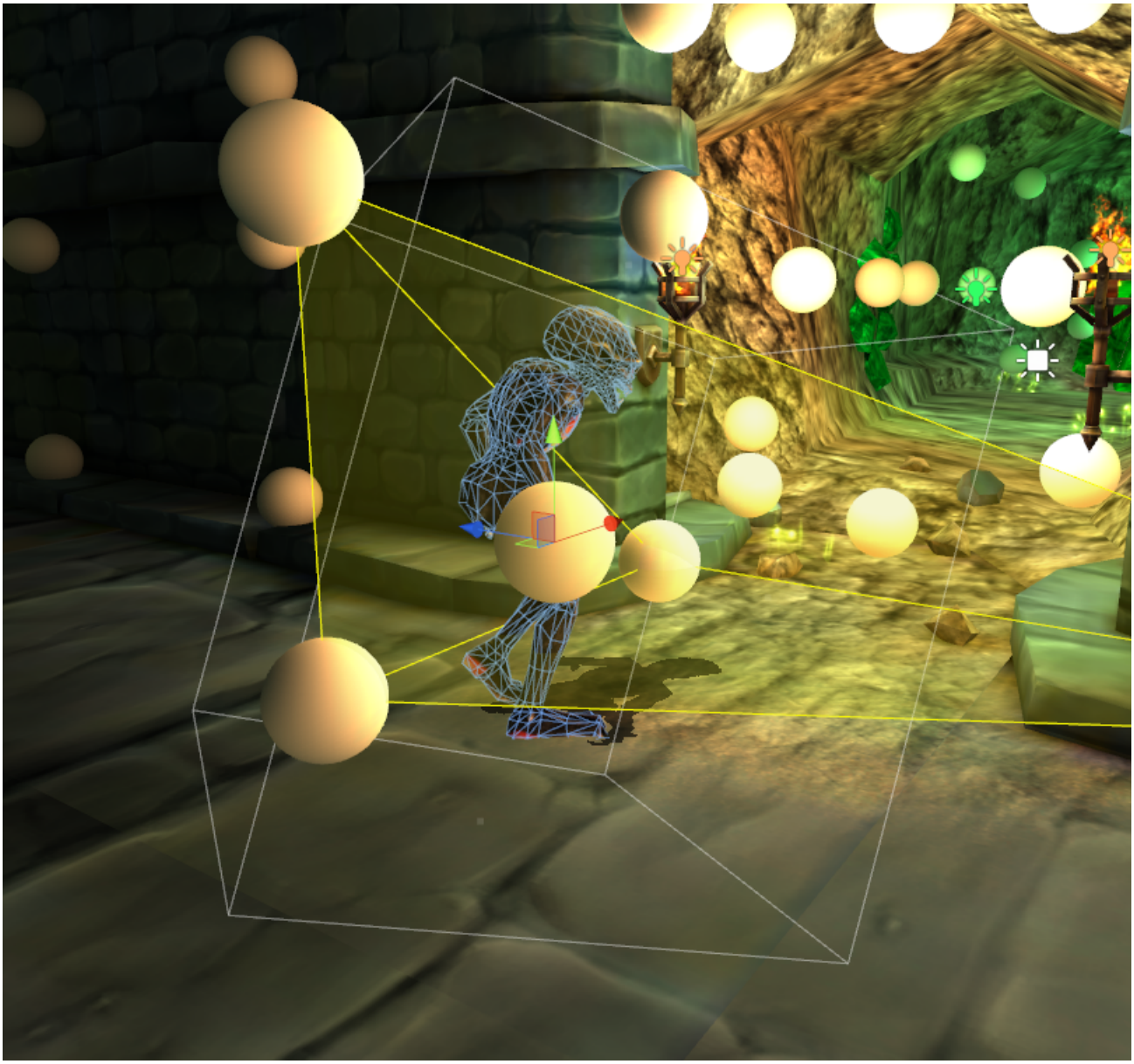

Figure 4-15 Multiple light probes

4.2.3 ASTC texture compression

ASTC texture compression is an official extension to the OpenGL and OpenGL ES graphics APIs. ASTC can reduce the memory required by your application and reduce the memory bandwidth required by the GPU.

ASTC offers texture compression with high quality, low bitrate and has many control options. It includes the following features:

- Bit rates range from 8 *bits per pixel* (bpp) to less than 1bpp. This enables you to fine-tune the trade-off of space against quality.
- Support for 1 to 4 color channels.
- Support for both *low dynamic range* (LDR) and *high dynamic range* (HDR) images.
- Support for 2D and 3D images.
- Support for selecting different combinations of features.

The following figure shows the ASTC settings window:

Figure 4-16 ASTC settings

There are a number of block size options available in the ASTC settings window. You can choose among these options and select the block size that best fits the assets. The larger block sizes provide higher compression. Select large block sizes for textures that are not shown in great detail, for example, objects far away from camera. Select smaller block sizes for textures that are show more detail, for example those closer to camera.

Note

- If your device supports ASTC, use it to compress the textures in your 3D content. If your device does not support ASTC, try using ETC2.
- You must differentiate between textures used in 3D content from textures used in the GUI elements. In some cases it might be best to leave the GUI textures uncompressed to avoid unwanted artifacts.

The following figure shows the block sizes available for different texture compression formats:

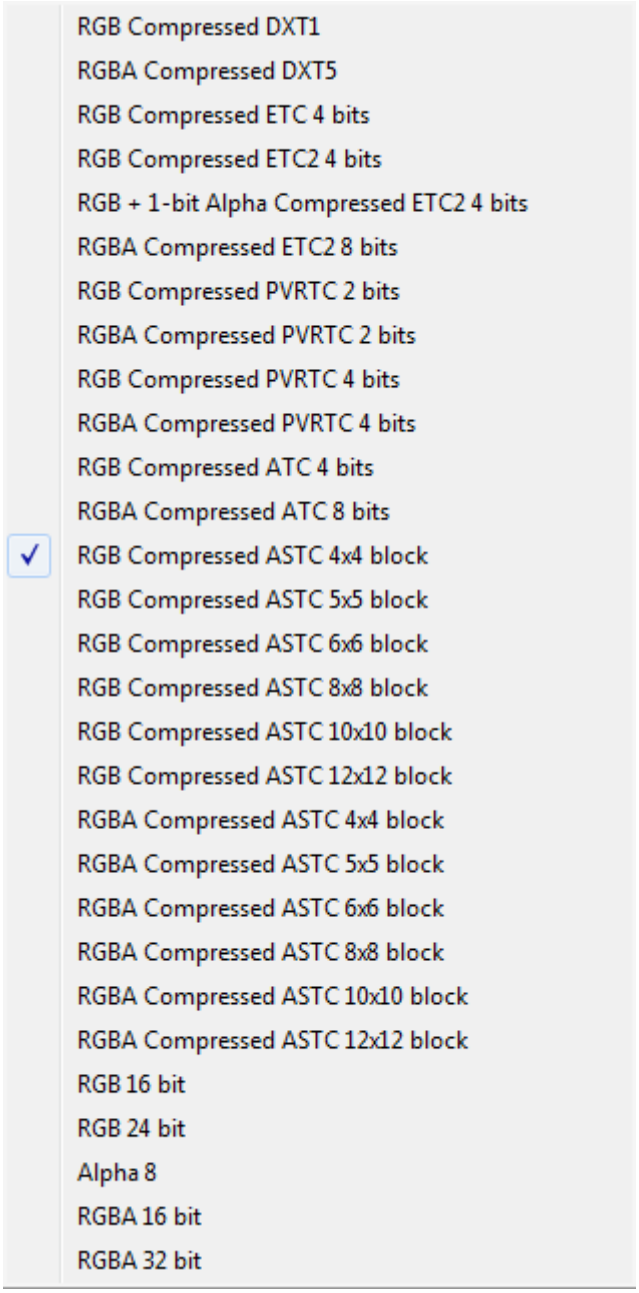

Figure 4-17 Texture compression block sizes

Selecting the correct format for ASTC textures

When compressing an ASTC texture, there are a number of options to choose from.

Texture compression algorithms have different channels formats, typically RGB and RGBA. ASTC supports several other formats, but these formats are not exposed within Unity. Each texture is typically used for a different purpose such as, standard texturing, normal mapping, specular, HDR, alpha, and look up textures. All of these texture types require a different compression format to achieve the best possible results.

The following figure shows texture settings:

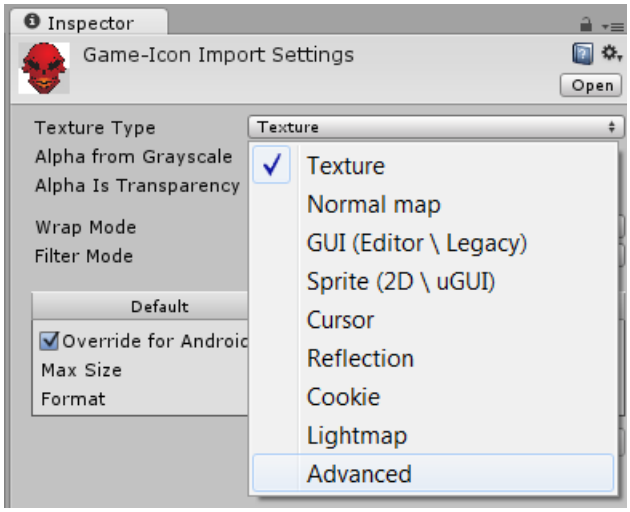

Figure 4-18 Texture settings

Do not compress all of your textures with one format in **Build Settings**. Keep texture compression as **Don't Override**.

Find your texture within the project hierarchy and bring it up in the Inspector. Unity typically imports your texture as the type **Texture**. This type only provides limited options for compression. Set the type to **Advanced** to show a larger choice of options.

The following diagram shows setting for a GUI texture with some transparency. The texture is for a GUI, so **sRGB** and **MipMaps** are disabled. To include transparency, you require the alpha channel. To enable this, tick the **Alpha Is Transparency** box and tick the **Override for Android** box.

The following figure shows advanced texture settings:

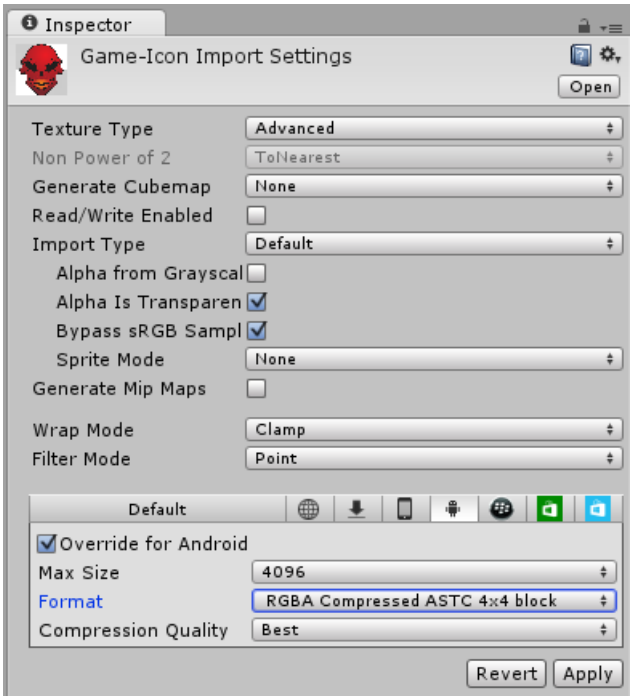

Figure 4-19 Advanced texture settings

There is an option to select a format and block size. RGBA includes the alpha channel and 4x4 is the smallest block size you can select. Set **Max texture size** to the maximum and set **Compression Quality**, this setting defines how much time is spent looking for accurate compression.

Selecting specific settings for all of your textures improves the visual quality of your project and avoids unnecessary texture data at compression time.

The following table shows the compression ratio for the available ASTC block sizes in Unity for an RGBA 8 bits per channel texture with a 1024x1024 pixel resolution at 4 MB in size.

Table 4-1 Compression ratios for the ASTC block sizes available in Unity

ASTC block size	Size	Compression ratio
4x4	1 MB	4.00
5x5	655 KB	6.25
6x6	455 KB	9.00
8x8	256 KB	16.00
10x10	164 KB	24.97
12x12	144 KB	35.93

4.2.4 Mipmapping

Mipmapping is a technique related to textures that can both enhance the visual quality and the performance of your game.

Mipmaps are pre-calculated versions of a texture at different sizes. Each texture generated is called a level, and it is half as wide and half as high as the preceding one. Unity can automatically generate the complete set of levels from the first level at the original size down to a 1x1 pixel version.

To generate the mipmaps do the following:
1. Select a texture in the **Project window**.
2. Change the **Texture Type** to **Advanced**.
3. Enable the **Generate Mip Maps** option in the **Inspector**.

The following figure shows mipmap settings:

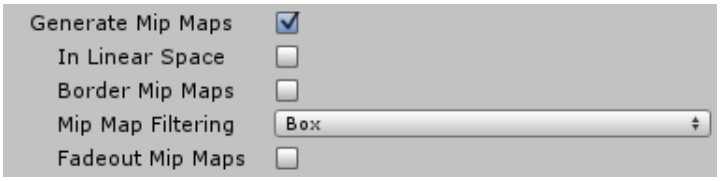

Figure 4-20 Mip map settings

If a texture does not have mipmap levels, when the area in pixels covered by a textured surface is smaller than the size of its texture, the GPU scales the texture down to fit the smaller area. However, some accuracy is lost in this process even with a filter to interpolate the pixel colors.

If a texture does have mipmap levels, the GPU fetches pixel data from the level closest to the object size to render the texture. This ensures both higher image quality and reduced bandwidth because the levels are scaled offline for better quality and only the texture data from the correct level is fetched by the GPU. The disadvantage of mipmapping is it requires 33% more memory to store the texture data.

Mipmaps and GUI textures

You do not usually require mipmapping for textures used in a 2D UI. UI textures are typically rendered on screen without scaling so they only use the first level in the mipmap chain.

To change this setting select a texture in the **Project** window then go in the **Inspector** and look at **Texture Type**. Set the type to **Editor GUI and Legacy GUI**, or set the type to **Advanced** and disable the **Generate Mip Maps** option.

4.2.5 Skyboxes in Unity

Skyboxes are often used in games and other applications, there are several methods to implement them.

You can draw a skybox by rendering the background of the camera using a single cubemap.

This requires one cubemap texture and one draw call. This uses less memory, memory bandwidth, and draw calls, compared to other methods.

To set up a skybox using this method:
1. Select the camera.
2. Ensure **Clear Flags** is set to **Skybox**.
3. Select or add a skybox component.

The skybox component has one spot for a material. This is the material that Unity uses to draw the background of your camera at the start of each frame.

The material you use is the one that contains all the information you require. Create a material using the **Mobile** > **Skybox** shader and fill the six images of the Skybox with the material. Your material preview displays an image.

When you have completed the material, drag it into the skybox component. Your skybox renders in the background correctly, without any obvious seams or unnecessary draw calls.

4.2.6 Shadows in Unity

Shadows help add perspective and realism to your scenes. Without them it can sometimes be difficult to tell the depth of objects, especially if they are similar to the surrounding objects.

Shadow algorithms can be very complex, especially when rendering high resolution accurate shadows. Ensure you select an appropriate level of complexity and resolution for the shadows in your game.

Unity supports transform feedback for calculating real-time shadows.

——— **Note** ———

The Ice Cave demo implements custom shadows. Shadows based on local cubemaps are combined with shadows rendered at runtime.

———————————

Unity has a number of options for shadows under **Edit** > **Project Settings** > **Quality** that can impact the performance of your game:

Hard/Hard and Soft Shadows
: Soft shadows look more realistic but take longer to calculate.

Shadow Distance
: The **Shadow Distance** option defines the distance from the camera that shadows appear in. Increasing the shadow distance increases the number of shadows visible, and this increases the computational load. Increasing the shadow distance also increases the number of texels available for the shadows in the shadow map, passively increasing the resolution of your shadows.

You can use hard shadows with a small shadow distance and a high resolution. This produces reasonable quality shadows within a good distance of the camera that are not too complex.

Light mapped objects do not produce realtime shadows, the more static shadows you can bake into the scene fewer less real-time calculations your GPU must do.

The following figure shows an alien character with a shadow:

Figure 4-21 Alien with shadow

Use real-time shadows sparingly

Real-time shadows can dramatically enhance the realism of a scene but they are computationally expensive.

On mobile devices, try to limit the number of lights that include real-time only shadows and try to use lightmapping instead.

Consider the mesh renderer component of the objects in your scene. If you do not intend to use them for casting or receiving shadows disable the **Cast Shadows** and **Receive Shadows** options accordingly. This reduces the computation cost of rendering shadows.

You can find more settings for shadows in the **Quality Settings** section such as:

- **Shadow Resolution** enables you to select the balance between quality and processing time.
- **Shadow Distance** enables you to limit shadow generation to objects close to the camera.
- **Shadow Cascades** enables you to select the balance between quality and processing time. You can set this to zero, two or four. Cascaded Shadow Maps are used for directional lights to achieve very good shadow quality, especially for long viewing distances. A higher number of cascades produces better quality, but increases processing overhead.

4.2.7 Occlusion Culling

Occlusion Culling is a process that disables the rendering of objects when they are obscured from the view of the camera. This saves GPU processing time by rendering fewer objects.

Unity automatically performs frustum culling when objects exit the camera frustum completely however, depending on your style of application, there might still be other objects that cannot be seen and do not have to be rendered.

Unity includes an occlusion culling system called Umbra. For more information on Umbra see occlusion culling in the Unity documentation.

The settings you use for occlusion culling depends on the style of your game. You must be careful picking settings because using occlusion culling in a scene with the incorrect settings can degrade performance.

4.2.8 Use OnBecameVisible()and OnBecomeInvisible() callbacks

If you use the callbacks `MonoBehaviour.OnBecameVisible()` and `MonoBehaviour.OnBecameInvisible()`, Unity notifies your scripts when their associated game objects moves in or out of a camera frustum. Your application can then act accordingly.

You can use `OnBecameVisible()` and `OnBecameInvisible()` to optimize the rendering process, for example, rendering reflections on a pool with a second camera and render targets.

This involves rendering geometry and combining textures off screen before rendering to the final screen surface. This technique is relatively expensive so it is only used when it is necessary. You are only required to render a reflection when it is visible. That is when:

- The reflection surface is within the camera frustum.
- Nothing opaque is in front of the surface.

These conditions are checked with the `OnBecameVisible()` and `OnBecameInvisible()` callbacks from the reflective surface:

```
void OnBecomeVisible()
{
    enabled = true;
}

void OnBecomeInvisible()
{
    enabled = false;
}
```

Even with these checks in place there can still be times when a reflection might be rendered off screen even though it is not visible onscreen. To avoid this you can add another condition:

For example, the camera must be inside the room of the reflective surface:

```
void OnBecomeVisible()
{
    if (inside == false)
    {
        return;
    }
    enabled = true;
}

void OnBecomeInvisible()
{
    if (inside == false)
    {
        return;
    }
    enabled = false;
}

void OnTriggerEnter()
{
    inside = true;
}

void OnTriggerExit()
{
    inside = false;
}
```

These conditions restrict the rendering of reflections to specific areas of the game. This means you can add effects in other, less compute intensive areas of the game.

4.2.9 Specify the rendering order

In a scene, the object rendering order is very important for performance.

If objects were rendered in random order, an object might be rendered and subsequently occluded by another object in front of it. This means all the computations required to render the occluded object are wasted.

Various software and hardware techniques exist to reduce the amount of wasted computation because of occluded objects, but you can guide this process because you have knowledge of how the scene is explored by a player.

One of the hardware techniques for reducing wasted computation that is available on the ARM Mali GPUs from the Mali-T600 series onwards, is Early-Z. Early-Z is a completely transparent system from your point of view that performs a Z-test before the fragment shader is actually processed. Z testing typically happens after the fragment shader, however usually this shader is the most computationally expensive so it should be avoided if the fragment is not visible in the scene. The Early-Z system checks that the depth of the pixel being processed is not already occupied by a nearer pixel. If it is occupied, it does not execute the fragment shader. This system provides performance benefits but it is automatically disabled in some cases, such as if the fragment shader modifies the depth by writing into the `gl_FragDepth` variable, the fragment shader calls `discard`, or if blending or alpha testing is enabled for objects such as transparent objects. To assist this system to achieve maximum efficiency, ensure that opaque objects are rendered from front to back. This helps to reduce the overdraw factor in scenes with only opaque objects.

Ordering the rendering of each frame front-to-back can be expensive and also incorrect if you render transparent objects in the same pass. ARM Mali GPUs from T620 onwards provide a mechanism called *Pixel Forward Kill* (PFK). Mali GPUs are pipelined so multiple threads can be concurrently executing for the same pixel. If a thread completes its execution, the PFK system stops all other threads for that pixel if the current one covers them. The effect is a reduction of wasted computations.

Unity provides you with **Queue** options inside the shaders, or in the material to specify the order of rendering. This can be set in the shader, so objects that have a material that uses that shader are rendered together. Inside this rendering group, the order of rendering is random except in some cases such as transparency.

By default, Unity provides some standard groups that are rendered from first to last in the following order:

Table 4-2 Queue values for specifying rendering order

Name	Value	Notes
Background	1000	-
Geometry	2000	Default, used for opaque geometry.
AlphaTest	3000	This is drawn after all opaque objects. For example, foliage.
Transparent	4000	This group is also rendered in back to front order to provide the correct results.
Overlay	5000	Overlay effects such as user interface, lens flares, dirty lens.

The integer values can be used instead of their string names. These are not the only values available. You can specify other queues using an integer value between those shown. The higher the number, the later it is rendered.

For example, you can use one of the following instructions to render a shader after the Geometry queue, but before the AlphaTest queue:

```
Tags { "Queue" = "Geometry+1" }
```

```
Tags { "Queue" = "2001" }
```

Using the rendering order to increase performance

In the Ice Cave demo, the cave covers large part of the screen and its shaders are expensive. Avoiding rendering parts of it when possible can increase performance.

Rendering order optimization was included after looking at the composition of the framebuffers using the Unity Frame Debugger and other tools such as the ARM Mali Graphics Debugger. These enable you to see the rendering order.

To open the Unity Frame Debugger, select the menu option **Window** > **Frame Debugger**. This is useful because there might be things that appear to be correct in editor mode, but might not work correctly when you execute them. This can be the case if for example, you have run time only settings, or if you are rendering to a texture from another camera. After starting the demo in play mode and positioning the camera, you can enable the Frame Debugger and get the sequence of drawings executed by Unity.

In the Ice Cave demo, scrolling down the draw calls shows that the cave is rendered first. The objects are then rendered into the scene occluding parts of the cave that are already rendered. Another example is the reflective crystals that in some scenes are occluded by the cave. In these cases, setting a higher rendering order results in a reduction in computations because fragment shaders are not executed for the occluded crystals.

4.2.10 Use depth pre-pass

Setting the rendering order for objects to avoid overdraw is useful, but it is not always possible to specify the rendering order for each object.

For example, if you have a set of objects with computationally expensive shaders and the camera can rotate around them freely, some objects that were at the back can move to the front. In this case, if there is a static rendering order set for these objects, some might be drawn last, even if they are occluded. This can also happen if an object can cover parts of itself.

In these cases, you can use a depth-prepass to reduce overdraw. The depth-prepass renders the geometry without writing colors in the framebuffer. This initializes the depth buffer for each pixel with the depth of the nearest visible object. After this pre-pass the geometry is rendered as usual but using the Early-Z technique, only the objects that contribute to the final scene are actually rendered. Extra vertex shader computations are required for this technique because the vertex shader is computed two times for each object, once for filling the depth buffer and another for the actual rendering. This technique is very useful if your game is fragment bound and you have spare capacity in the vertex shader.

In Unity, to do a rendering pre-pass for objects with custom shaders, add an extra pass to your shaders:

```
// extra pass that renders to depth buffer only
 Pass {
     ZWrite On
     ColorMask 0
 }
```

After adding this pass, the frame debugger shows that the objects are rendered twice. The first time they are rendered there are no changes in the color buffer.

Note

You can see the depth buffer by choosing it in the top left menu of the frame debugger.

4.3 Asset optimizations

The following list describes asset optimizations:

Disable Read/Write for static textures

If you do not dynamically modify a texture, ensure the **Read/Write Enabled** option in the Inspector is disabled.

Combine meshes to reduce draw calls

To reduce the number of draw calls required for rendering you can combine several meshes into one with the `Mesh.CombineMeshes()` method. If the meshes all share the same material, set the `mergeSubMeshes` argument to `true` so it generates a single submesh out of each mesh in the combine group.

Combining several meshes into a single larger mesh helps you:

- Create more effective occluders.
- Turn tile-based assets into a single large seamless solid asset.

The mesh combine script can be useful for performance optimization but this depends on the makeup of your scene. Large meshes tend to stay in view longer than smaller meshes, so experiment to get the correct size.

One way to apply this technique is to create an empty game object in the hierarchy, make it the parent of all the meshes that you want to combine and then attach it to a script.

For more information on the mesh combine script, see the Unity documentation at *http://unity3d.com*.

Do not import animations data on FBX mesh models that do not animate

When importing an FBX mesh that does not contain any animation data, you can set the **Animation Type** to **None** in the Rig tab of the import settings. If this is set, placing your mesh into the hierarchy Unity does not generate an unused animator component.

Avoid Read/Write Meshes

If your model is modified at runtime, Unity keeps a second copy of the mesh data in memory to modify while preserving the original.

If your model is not modified at run-time, even to be scaled, disable the **Read/Write Enabled** option from the Model tab of the import settings. A second copy is no longer required so this saves memory.

Use texture atlases

You can use a texture atlas to reduce the number of draw calls required for a set of objects.
A texture atlas is a group of textures combined into one large texture. Multiple objects can reuse this texture with an appropriate set of coordinates. This helps with the automatic batching that Unity applies to objects sharing the same material.
When setting the UV texture coordinates of an object, avoid changing the `mainTextureScale` and `mainTextureOffset` properties of its material. This creates a new unique material that does not work with batching. Instead, access the mesh data through the `MeshFilter` component and change the coordinates per vertex using the `Mesh.uv` property.
The following figure shows a texture atlas:

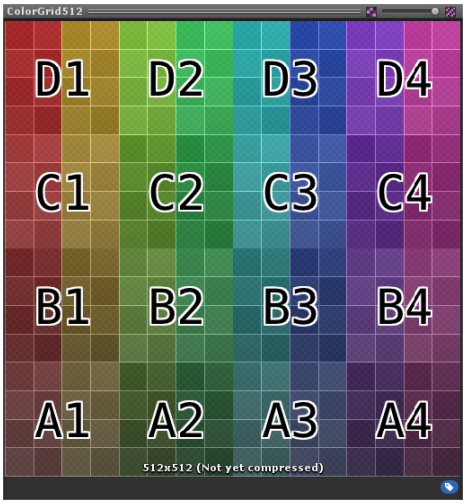

Figure 4-22 Texture atlas

4.4 Optimizing with the Mali Offline Shader Compiler

The Mali Offline Shader Compiler is a tool that enables you to compile vertex, fragment, and compute shaders into a binary form. You can also use the compiler as a profiling tool.

This section contains the following subsections:

4.4.1 About the Mali Offline Shader Compiler

The shaders in your application run on the GPU. This requires the GPU to spend time computing the final result of your shader, such as the vertex position or the color of the pixel.

The Mali Offline Shader Compiler provides information about the number of cycles shaders require to execute in each pipeline of the Mali GPU.

The cycle values produced are tailored for a specific GPU. You select the GPU as an option on the command line. Ensure you choose GPUs that correspond to that range of devices that you want your application to target. This ensures that the statistics you get from the tool are realistic and match your typical use case scenario.

4.4.2 Measuring Unity shaders with the Mali Offline Shader Compiler

You write Unity shaders in the programming language *C for Graphics* (Cg). Cg is based on the C programming language with some modifications to make it more suitable for GPU programming.

Unity translates the Cg to OpenGL, OpenGL ES, or DirectX during the build process.

To retrieve the OpenGL ES shader code:

1. Select the shader you want to analyze in Unity.
2. Choose **OpenGLES30** or **OpenGLES20** as the Custom Platform you want to build for.
3. Click the **Compile and show** button.

The result is displayed in your development environment.

Note

- The Mali Offline Shader Compiler only supports OpenGL ES shaders.
- If your build platform is set to Android, Unity builds **OpenGLES30** shaders by default.

Vertex and fragment sessions are typically delimited by `#ifdef VERTEX` or `#ifdef FRAGMENT`. If you use an option such as `#pragma multi_compile <FEATURE_OFF> <FEATURE_ON>`, multiple shader variants are built in the file.

Typically there are multiple `VERTEX` and `FRAGMENT` sections. Each variant is compiled separately by Unity starts your application. When you enable a feature, the relevant variant is selected.

Because the code has been translated into OpenGL ES, you can copy the vertex and fragment shader code into two separate files and compile them with the Mali Offline Shader Compiler.

Compile the shaders with one of the following options:

- `-v` for vertex shaders.
- `-f` for fragment shaders.

The following figure shows the output of the Mali Offline Shader Compiler:

```
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.


8 work registers used, 13 uniform registers used, spilling not used.

               A       L/S     T       Total   Bound
Cycles:        43      11      13      67      A
Shortest Path: 16      11      13      40      A
Longest Path:  16      11      13      40      A

Note: The cycles counts do not include possible stalls due to cache misses.

Compilation succeeded.
```

Figure 4-23 Output of the Mali Offline Shader Compiler

4.4.3 Analyzing statistics from the Mali Offline Shader Compiler

The statistics produced by the Mali Offline Shader Compiler provide a measurement of how many cycles per vertex or pixel the shader requires.

The result is subdivided into three lines:

- Total.
- Shortest path.
- Longest path.

The shortest and longest paths are measured by looking at the effect of taking or not taking branches in your code. This provides an estimate of the smallest and largest number of cycles of execution.

For Arithmetic, the measurement in the first line is divided by the number of Arithmetic pipelines. This is one, two or three, depending on the Mali GPU.

The second and third lines are for the Load/Store and Texture pipelines. These do not take into account cache misses, so it is best to multiply those numbers by 1.5 to get more realistic estimates.

4.4.4 About the Mali GPU pipelines

Mali GPUs contain three types of processing pipeline:

- Arithmetic pipeline.
- The Load/Store pipeline.
- The Texture pipeline.

The pipelines all run in parallel. Your shaders typically use all three types of pipeline.

The Mali Offline Shader Compiler provides numbers of cycles used in each pipeline. The shader is slowest in the pipeline with the highest number of cycles. Optimize your shader with optimizations that target that pipeline it is slowest in.

The Arithmetic pipeline

All arithmetic operations consume cycles in the Arithmetic pipeline.

The following are a number of ways you can reduce the Arithmetic pipeline usage:

- Avoid using complex arithmetic such as:
  - The inverse matrix function.
  - Modulo operators.
  - Division.
  - Determinant.

  - Sine.
  - Cosine.
- For integer operands, use operations such as shifts to compute divisions, modulo, and multiplications.
- Use transpose instead of inverse for orthogonal matrices.
- To avoid computing the transpose, switch the order of operands in a matrix-vector or matrix-matrix multiplication if one of the matrices is transposed. For example:

```
Transpose(A)*Vector == Vector * A.
```

You can also reduce load on the Arithmetic pipe by moving load to the other pipelines:

- Pass matrices as uniforms instead of computing them. This uses the Load/Store pipeline.
- Use a texture to store a set of precomputed values that represent a function such as sine or cosine. This moves the load to the Texture pipeline.

The Load/Store pipeline

The Load/Store pipeline is used for reading uniforms, writing varyings, and accessing buffers in the shaders such as Uniform Buffer Objects or Shader Storage Buffer Objects.

If your application is Load/Store pipeline bound, try the following techniques:

- Use a texture instead of a buffer object to read data in the shader.
- Compute data using arithmetic operations.
- Compress or reduce uniforms and varyings.

The Texture pipeline

Texture accesses use cycles in the Texture pipeline and use memory bandwidth. Using large textures can be detrimental because cache misses are more likely and this can cause multiple threads to stall while waiting for data.

To improve the performance of the Texture pipeline try the following:

Use mipmaps
: Mipmaps increase the cache hit rate because it selects the best resolution of the texture to use based on the variation of texture coordinates.

Use texture compression
: This is also good for reducing the memory bandwidth and increasing the cache hit rate. Each compressed block contains more than one texel, so accessing it makes it more cacheable.

Avoid trilinear or anisotropic filtering
: Trilinear and anisotropic filtering increase the number of operations required to fetch texels. Avoid using these techniques unless you absolutely require them.

4.4.5 Additional techniques for reducing pipeline cycles

There are a number of additional techniques you can use to reduce the cycles used in each pipeline.

Avoid register spilling

The Mali Offline Shader Compiler indicates if your shader spills registers. Register spilling is typically caused in a thread by a high number of input uniforms that cannot fit entirely in the register set.

Register spilling forces the Mali GPU to read some uniforms from memory, this increases the load on the Load/Store unit and reduces performance. To solve this issue, try to reduce the number and the precision of the uniforms you supply to the shader.

In the Ice Cave demo, some of the shaders suffered from register spilling, for example:

```
8 work registers used, 16 uniform registers used, spilling used.
```

Figure 4-24 Shader with register spilling.

Reducing the number of uniforms permitted solves this problem, and the result is an increase in performance, for example:

```
8 work registers used, 13 uniform registers used, spilling not used.
```

Figure 4-25 Shader with no register spilling.

Reduce the precision of varying and uniforms

When you write custom shaders, you can specify the floating point precision of uniforms and varyings using 32-bit floats or 16-bit half-floats. The precision determines the minimum and maximum values that the variable can represent.

There are several advantages of using half-floats:

- Bandwidth usage is reduced.
- The cycles used in the Arithmetic pipeline are reduced because the shader compiler can optimize your code to use more parallelization.
- The number of uniform registers required is reduced and this in turn reduces the risk of register spilling.

The following code provides examples of a simple fragment shader variant from the Ice Cave demo. The shader is compiled with the Mali Offline Shader Compiler twice.

The first code example is compiled with floats:

```
$ malisc -f -V Compiled-CaveMaliStandardFloat.frag
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.


8 work registers used, 13 uniform registers used, spilling used.

                A       L/S     T       Total   Bound
Cycles:         47      15      10      72      A
Shortest Path:  15      13      9       37      A
Longest Path:   16      15      10      41      A

Note: The cycles counts do not include possible stalls due to cache misses.

Compilation succeeded.
```

Figure 4-26 Shader compiled with Floats

The second code example is compiled with half-floats:

```
$ malisc -f -V Compiled-CaveMaliStandardHalf.frag
ARM Mali Offline Compiler v4.6.0
(C) Copyright 2007-2015 ARM Limited.
All rights reserved.

No driver specified, using "offline_api" as default.

No core specified, using "Mali-T880" as default.

No core revision specified, using "r2p0" as default.


7 work registers used, 7 uniform registers used, spilling not used.

                A       L/S     T       Total   Bound
Cycles:         47      11      10      68      A
Shortest Path:  15      9       9       33      A
Longest Path:   15      11      10      36      A

Note: The cycles counts do not include possible stalls due to cache misses.

Compilation succeeded.
```

Figure 4-27 Shader compiled with Half floats

The number of Load/Store instructions is reduced in the half-float version. The number of work and uniform registers used is reduced and there is no register spilling.

The code generated with half-floats is also smaller than code generated with floats. This improves the cache hit rate on the Mali GPU increasing performance.

Use world space normal maps for static objects

You can use Tangent space normal maps to increase the details of a model without increasing the geometric detail. You can use tangent space normal maps on animated objects without modifying them because of their locality to each triangle of the mesh.

Unfortunately these require more arithmetic operations to be performed in the shaders to achieve the correct result. For static objects, these calculations are typically unnecessary.

You can alternatively use local space normal maps or world space normal maps. Using local space normal maps reduces the number of calculations performed in the shaders but transformations on the model must be applied to the sampled normal. World space normal maps do not require any transformations but these are static and the objects cannot move. In the Ice Cave demo, the cave and other high quality objects are static and using world space normal maps reduces the number of ALU operations required by the shaders considerably. Most common 3D modeling tools can create world space normal maps or you can generate them by code in an offline process.

Chapter 5
Global Illumination in Unity with Enlighten

This chapter describes global illumination in Unity with Enlighten.

It contains the following sections:

5.1 About Enlighten

Unity 5 includes Enlighten, a real-time global illumination solution from Geomerics.

You can use Enlighten for baking light maps, light probes, and for real-time indirect lighting. This works both in real-time in a game, and also in the Unity 5 editor on many of the platforms supported by Unity. These include Windows, OS X, iOS and Android.

The core of Enlighten is a real-time radiosity engine. It generates a light map containing only the indirect lighting, that is, the light that is bounced off surfaces back into the scene. The following figure shows this:

Figure 5-1 Arches

The left scene is illuminated by direct lighting, with no bounce light and no added ambient light term. Any surfaces in shadow are black.

The middle image shows the same scene and lighting setup with only the indirect light contribution and no albedo textures applied. This is the output Enlighten generates in real-time.

The right image shows the final composition that you normally see in a game.

Enlighten calculates the indirect light. The results are stored in a light map or in light probes, and applied in the shader code. The standard Unity materials all make full use of Enlighten and this applies fully to custom material shaders. The shaders and the Unity rendering engine also ensure the direct light is properly rendered, so that the final game looks like the final composition.

5.2 The structure of Enlighten

Enlighten components are not explicitly exposed in Unity. They are referenced in the user interface so it is useful to know what they are and how they work together. This section describes the components.

This section contains the following subsections:

5.2.1 Pre compute

The pre-compute stage is divided into a number of steps including packing, clustering, and computing the light transport.

The following figure shows the status of the running tasks in the progress bar at the bottom of Unity.

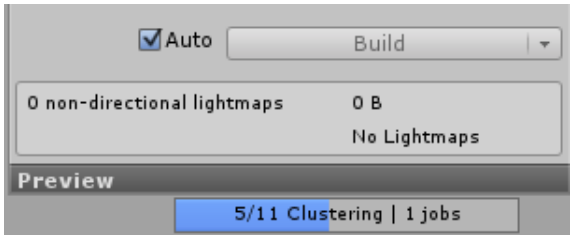

Figure 5-2 Progress

To illustrate the individual steps, consider the scene in the following figure, consisting of a floor, a wall with a corner and a TV hanging on the wall:

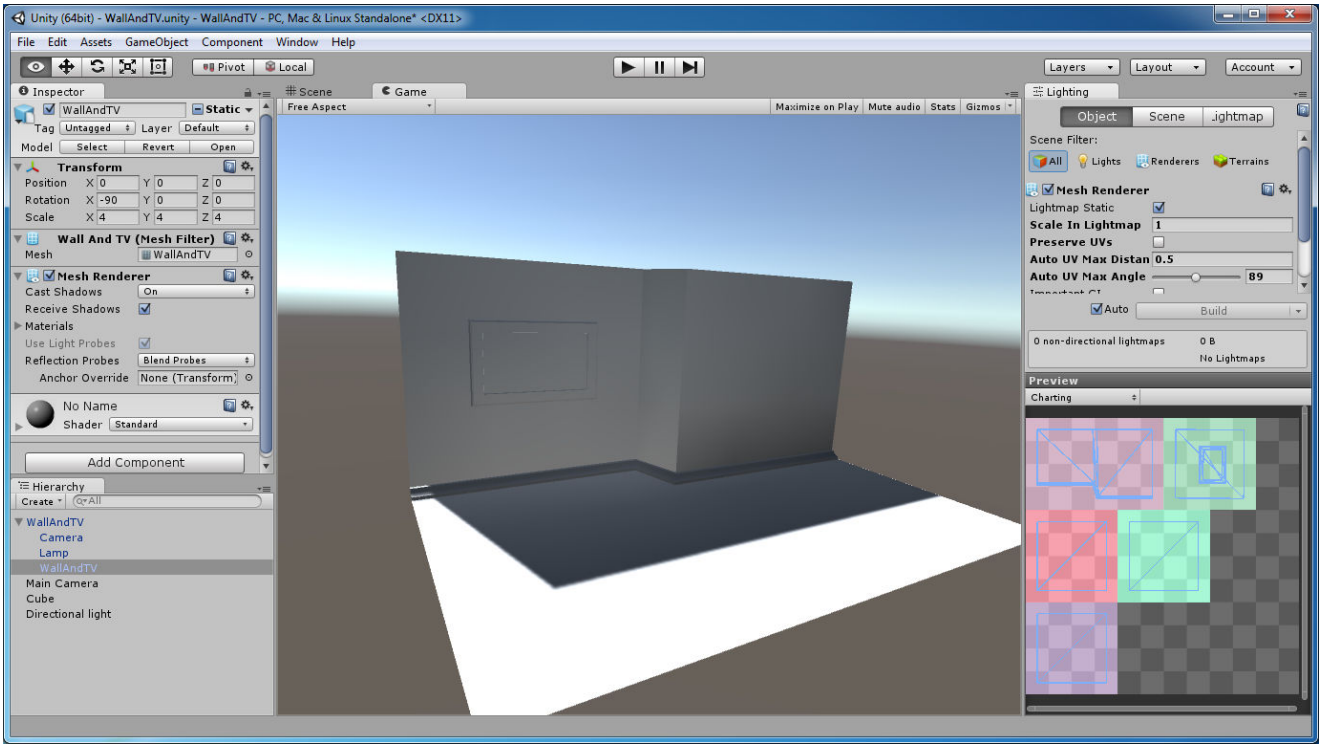

Figure 5-3 Wall and TV scene

Packing

Packing takes the existing UVs or the UVs Unity created for you when you select Generate Lightmap UVs in the Import settings, identifies charts, and packs the charts into the light map used for the real-time lighting.

Charts are groups of triangles that share the same vertices. Enlighten repacks these UVs to ensure that there is no light leaking between charts while also ensuring that the UVs are packed as tightly as possible. You can see the results of the packing in the **UV Charts** mode and also in the Charting preview, shown in the previous screenshot.

The precompute, run-time memory usage, and the run-time performance of Enlighten all depend on the number of light map pixels. You can use Enlighten to generate simplified UVs that deal with complex geometry more efficiently. Simplified UVs are on by default, but if you carefully crafted your own UVs, you can disable this by ticking **Preserve UVs**. You can see the results in the **UV Charts** render mode.

The following figure shows this mode with **Preserve UVs** enabled.

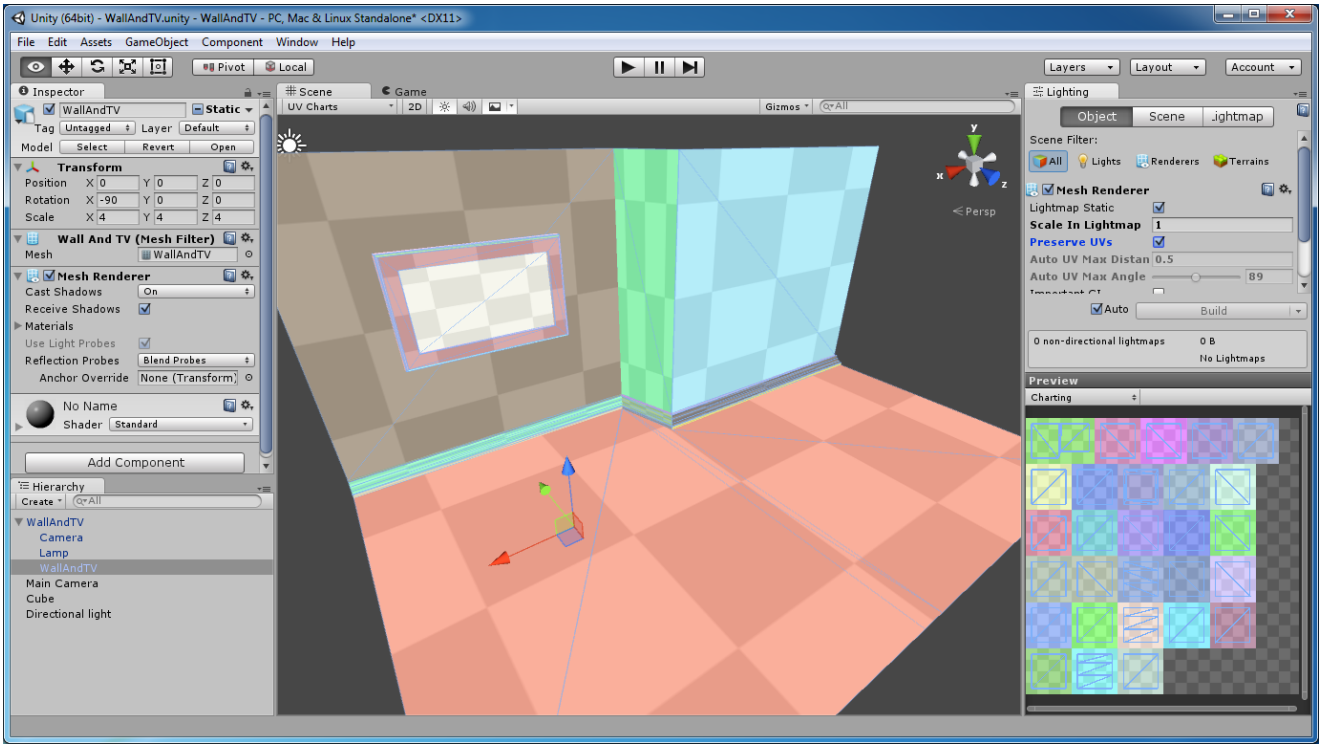

Figure 5-4 Wall and TV scene with Preserve UVs

The original UV layout and its charts are all shown in a different color. Parts of the TV, the wall, and the skirting board all consist of multiple charts, leading to an overall larger number of charts.

Only enable **Preserve UVs** if you have carefully authored UVs that you want to preserve in Unity. In most cases it is best to not enable this flag.

If **Preserve UVs** is disabled and the **Auto UV Max Distance** set to a small value, such as `0.01` the packing results look like the following figure:

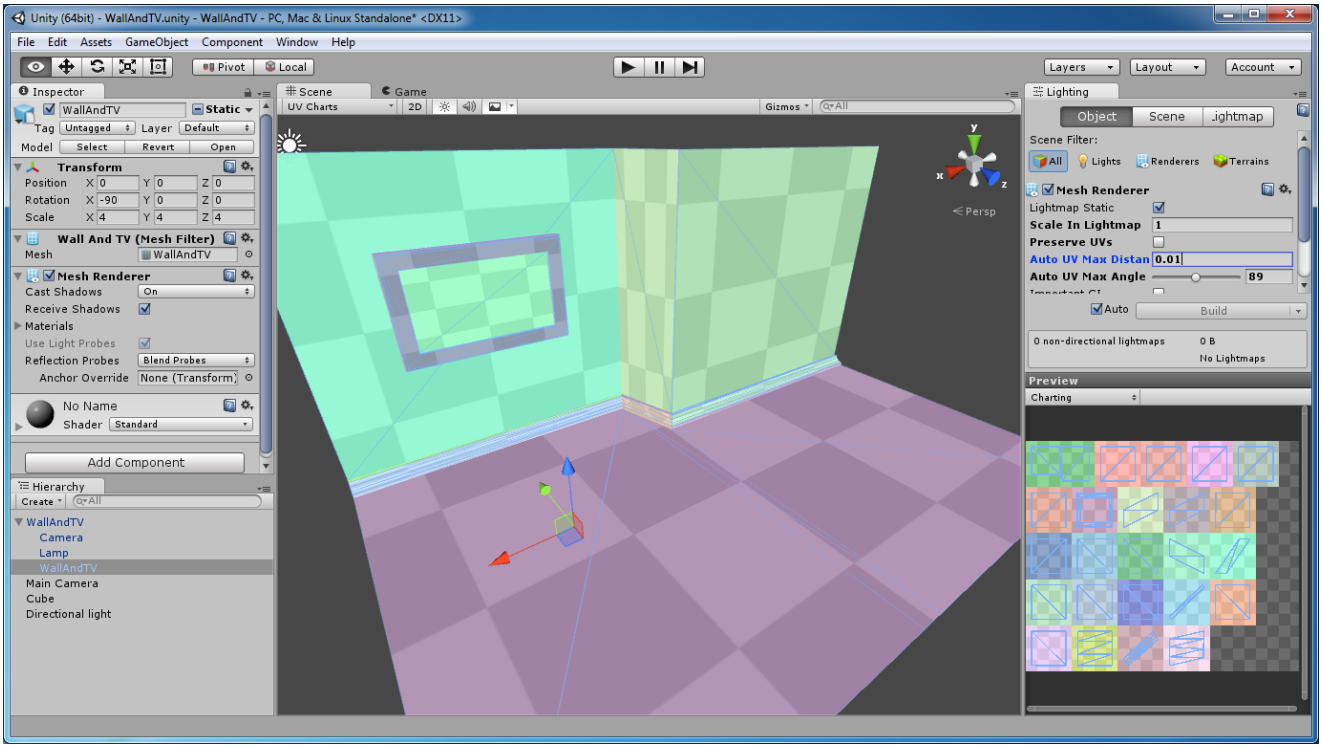

Figure 5-5 Wall and TV scene with Auto UV Max Distance at 0.01

Except for the colors of the visualization, the packing is very similar. It has roughly the same number of charts, but the unwrapping is slightly different.

The Auto UV Max Distance tells Enlighten what details can be omitted from the illumination. Enlighten tries to merge charts by creating a UV layout for the combined charts, but it does not split input charts. The new UV layout is created by projecting the vertices of all charts considered onto an optimal plane in world space. Only charts whose world space vertices are not more than the given distance above the plane are considered for merging.

With the chosen light map resolution, only a few light map pixels are used for the wall and the floor. This is enough to capture the coarse global illumination of the scene. You can expect that the fine details are filled in by screen-space ambient occlusion. The TV and the skirting board on the other hand are so small that they barely require one pixel, but were allocated far more, because they consist of several charts, and charts have a minimum size of 4 pixels by 4 pixels.

The following figure shows the result of increasing the distance value to 0.5:

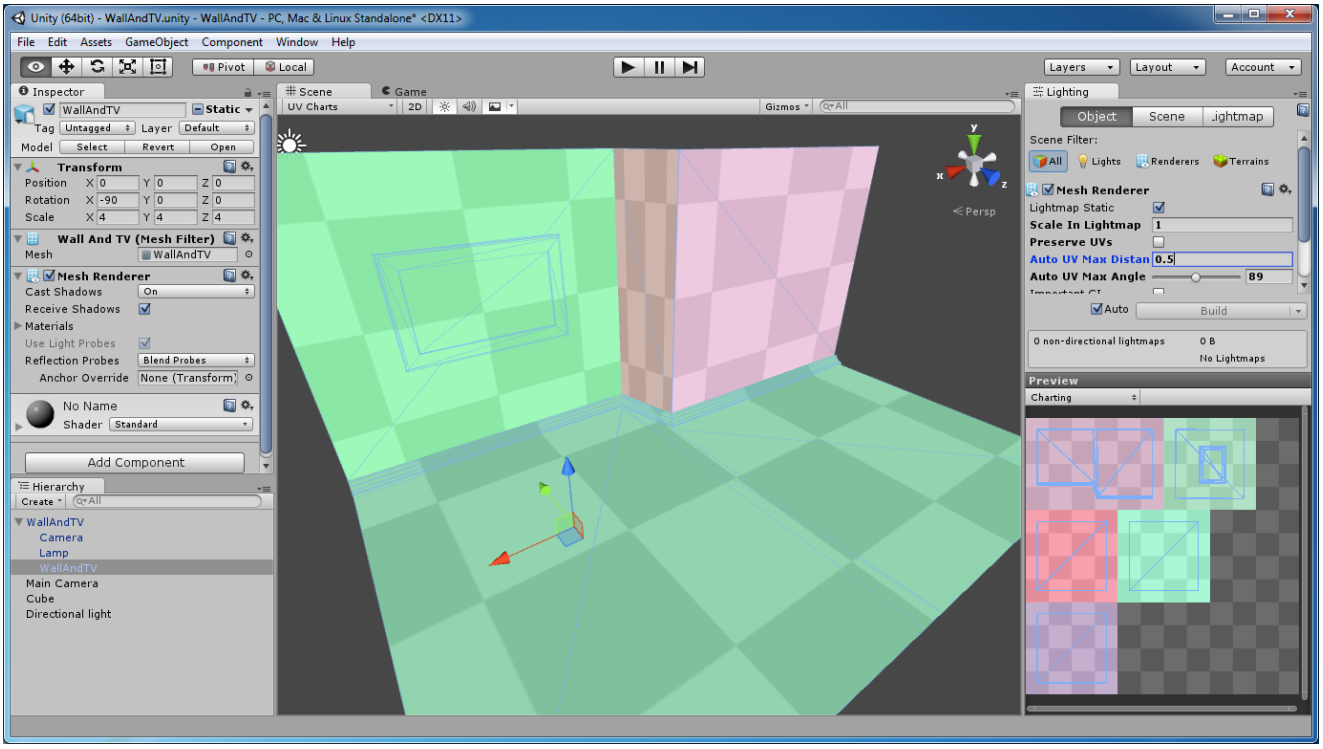

Figure 5-6 Wall and TV scene with Auto UV Max Distance at 0.5

Enlighten projects the UVs of the TV and the skirting board onto the wall and floor respectively. They therefore do not require any additional pixels in the light map. You can clearly see this in the Charting preview. Using even larger values lead to more details being ignored.

The following figure shows the results of a distance of 2:

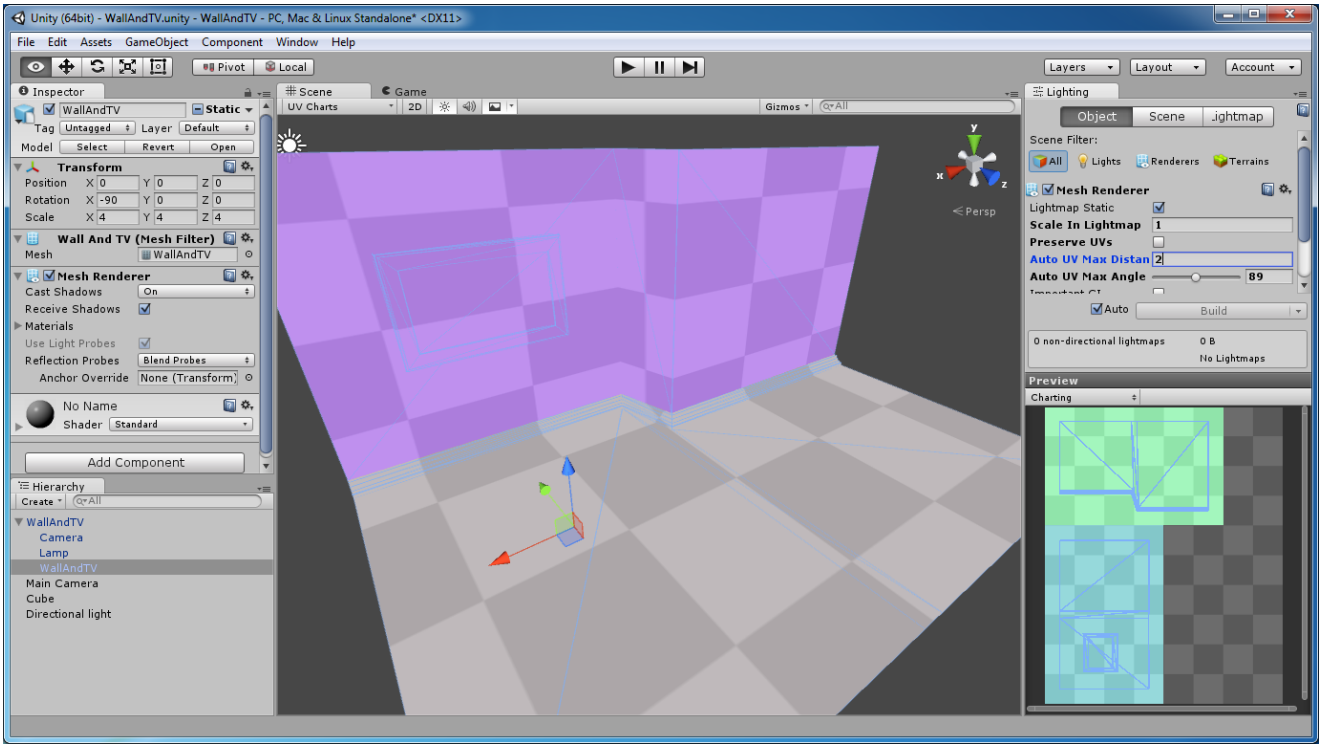

Figure 5-7 Wall and TV scene with Auto UV Max Distance at 2

While the light map is even smaller, Enlighten considers the different parts of the wall, including the corner, to be one segment, with pixels spanning the different parts. Similar to Auto UV Max Distance, you can define what charts Enlighten considers for merging based on the angle between triangles. The default value of 89, in degrees, ensures that Enlighten does not merge charts that are set at a right angle to each other in world space.

Clustering and light transport

To model the light transport in a scene, Enlighten splits the scene geometry into clusters.

The clusters are separate from the light map pixels and generated only from the position and orientation of the triangles. Materials or UV coordinates are not used. Unity does not have a visualization for clusters.

The following figure shows the clusters in the demo scene:

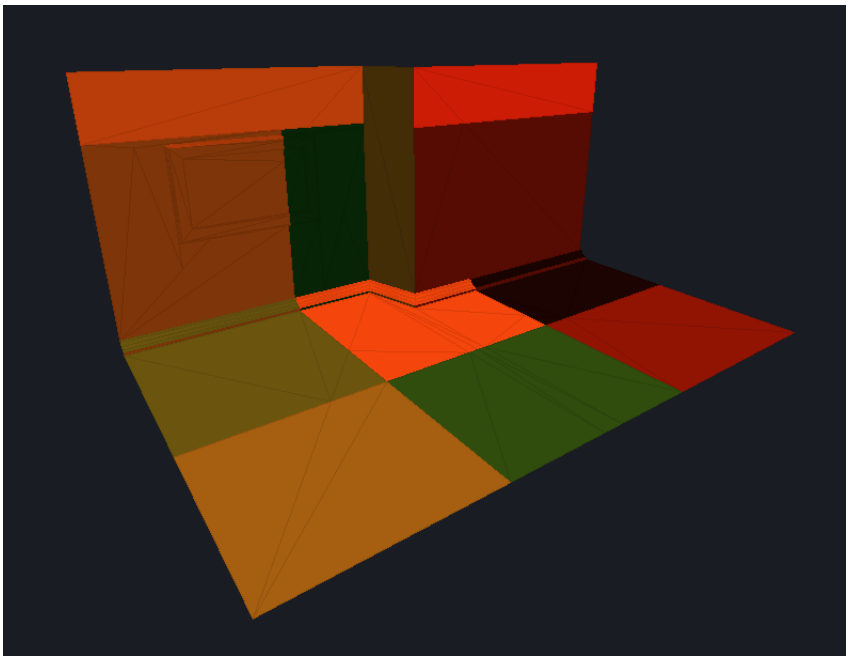

Figure 5-8 Wall and TV Clusters

Every cluster receives input lighting from lights placed in the scene, emits light itself, or can both receive and emit light. For every pixel in a light map, and for every light probe, Enlighten casts rays in all directions of its hemisphere or in all directions for probes. It calculates the visibility of the cluster from the pixel or probe. Enlighten assumes that the scene only consists of diffuse surfaces, so that the visibility is directly proportional to the light that the pixel receives from the clusters. The visibility is often called form-factor in computer graphics. When the form-factors have been calculated, they are compressed to reduce memory usage and performance requirements.

The following figure shows a simplified view of this process. The figure shows a pixel `X` that sees the three clusters `A`, `B` and `C` with visibility values of `0.35`, `0.3` and `0.2`. At run-time, Enlighten can evaluate the lighting that this pixel receives by evaluating

```
0.35*A+0.3*B+0.2*C
```

After the form-factors are calculated, they are compressed to reduce memory and performance costs.

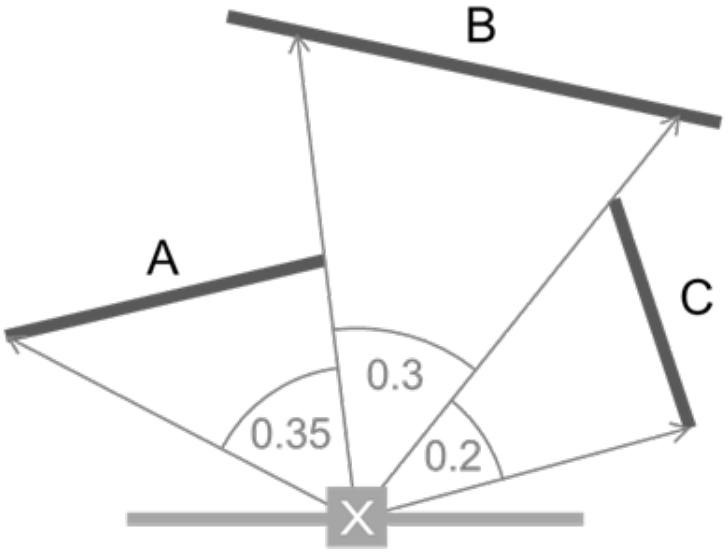

Figure 5-9 Form Factors

The pre-computed data must be updated whenever the geometry is changed, because the clusters depend on the geometry of the scene. Unity can automatically trigger the pre-compute when it is required, or you can turn the continuous build off, and only trigger it when you are finished with the scene changes. You must pre-compute a scene before you can use real-time lighting or bake lights.

Unity exposes the **Cluster Resolution** setting that controls the size of a cluster in relation to the light map resolution. The default value of 0.5 means that a cluster is twice the size of a light map pixel. There is rarely any reason to change this.

The light transport calculates the form-factors. Enlighten has two important parameters that affect this step:

- The `Irradiance Budget` is the number of form-factors that Enlighten stores. Enlighten merges form-factors as much as required to stay within a given budget, but a too small budget produces oversimplified illumination. On the other hand, a larger budget consumes more run-time memory and the run-time solving cost is higher. Normally the default value of 128 is a good choice, but values as low as 64 can also give good results. Increasing this value does not increase the light transport time and therefore also not the pre-compute time.
- The `Irradiance Quality` is the number of rays to cast per pixel. Increasing this value results in longer pre-compute times. The default of 8192 normally produces good results, but if you have chosen a high light map resolution, ARM recommends you also increase this value as well, because the indirect light might contain noise. This value does not affect run-time memory or performance.

To speed development, leave this option set to the default or a value that produces reasonable results when developing your game. Increase the value before shipping to get the maximum quality.

5.2.2 Real-time solver

The real-time module is present in the editor and the game. It generates light maps and probes for direct feedback of indirect lighting.

The real-time solver is split into the following stages:

- The input lighting stage calculates how lights illuminate the clusters, including dynamic lights. The light values are added together for all lights, and the final value is multiplied by the albedo of the cluster. This is based on the material of the geometry that the cluster is made of. Realistic rendering of lights always requires shadows and this applies to the input lighting as well. Directional lights can correctly handle shadowing, that is, clusters that are in the shadow area of the light do not receive any light. Enlighten supports shadowed spot lights and point lights as well, but this is not implemented in

Unity 5. For this reason, set an appropriate range for both spot and point lights to avoid light leaking of the indirect light.

- The solve stage sums up the cluster value multiplied by the stored form-factors, and stores the results in the light map. The run-time performance is therefore directly linked to the number of form-factors stored per pixel or probe.
- The bounce stage reads back the values from the light maps and bounces light values back to the clusters. This way Enlighten properly simulates multiple light bounces. In real-life the bounce happens at the speed of light, but it is limited to the light map update rate in Enlighten. This is typically limited by the render update rate.

Lights, materials and the pre-compute data are inputs to this step and some of these can be changed at run-time. This enables you to fine-tune the lighting and the appearance of the scene with instant feedback.

5.2.3 Baking light maps

Enlighten can generate baked light maps that contain direct lighting, indirect lighting and ambient occlusion. The indirect light map is generated based on the real-time results and these are up-sampled and filtered to produce high quality output.

You can also enable the final gather stage that uses the baked light maps as input, and Enlighten calculates another, final light bounce, with the precision bound by the bake pixel resolution. This step uses the standard baked light maps as input, so only consider switching to final gather when you are happy with the lighting in your scene.

Both the real-time and baked lighting can be combined. In Unity this is exposed on a per-light setting. Both types of lighting can contribute to the indirect lighting, making them blend together well.

The following figure shows light map baking settings:

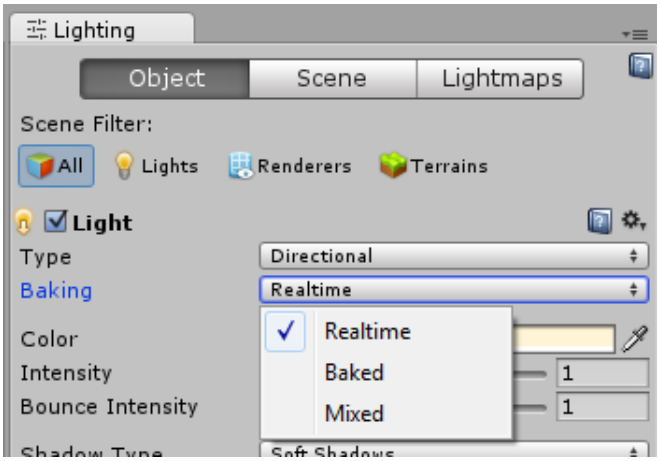

Figure 5-10 Light map baking settings

5.3 Setting up a scene with Enlighten

Start setting up your scene as if you are only using real-time global illumination. Do this even if you only plan on using baked light maps.

The advantage of this approach is that you get real-time feedback on your lighting, enabling a faster iteration time.

When you are happy with your lighting, you can turn on light map baking for all the required lights. Enlighten is responsible for both the real-time lighting and the light map baking, so the baked light maps match the real-time rendered results, with differences only visible in soft shadows and area lights.

Another advantage of this approach is that you can easily make use of more powerful target platforms by enabling real-time lighting.

The most important setup parameter to get right is the Enlighten light map resolution. In a real-world scene, with human sized characters, you typically set the texture resolution to be 1 pixel per meter. Smaller scale details are best handled with screen space ambient occlusion. Unity includes a rendering mode called **UV Charts**. This is probably the best mode to use for setting up a scene for Enlighten.

The meshes are only rendered in this mode when a pre-compute is started. However, the necessary calculations are the very first performed. If you notice that it takes a very long time for meshes to appear in this render mode, then the resolution might be too high. Try decreasing it and re-run the pre-compute. The pre-compute is triggered automatically, unless you have deactivated it.

When you have adjusted the resolution to your requirements, wait for the pre-compute to finish. If you later notice lighting artifacts such as light leaking, use the **UV Charts** render mode again. Enlighten does not split input charts, and charts that go through a wall can leak. For example, a chart on the floor. Consider creating smaller charts by splitting and separating the input UVs.

With the pre-compute done, you can add lights and get instant feedback about the overall scene appearance. This is not limited to light sources and their position. You can also change material properties, such as surface color, texture, or emissive settings. With Enlighten, any surfaces can be set up to emit light and so be turned into an area light. These area lights have the benefit of having no associated render cost because all their lighting is done by Enlighten. This can be especially useful for lower-end mobile devices, where the number of dynamic light sources is very limited. Use **Important GI** for emissive surfaces, especially if they are small, because this ensures that the clusters for this light geometry stay as compact as required.

Consider lighting smaller objects in your scene using probes instead of light maps. You can do this by not clicking the **Lightmap static** box. Smaller objects typically do not contribute much to the global illumination and setting them to probe lit removes them from the pre-compute stage, making it faster. Smaller objects are often also more difficult to generate UVs for. In some cases you can also merge smaller objects with larger objects, such as the TV and the wall in the example.

Chapter 6
Advanced Graphics Techniques

This chapter lists a number of advanced graphics techniques that you can use.

It contains the following sections:

6.1 Custom shaders

This section describes custom shaders.

This section contains the following subsections:

6.1.1 About custom shaders

Unity 5 and higher includes a Physically Based Shading (*PBS*) model that simulates the interactions between material and light. This provides a high level of realism and makes it possible to achieve a consistent look under different lighting conditions.

You can easily use PBS with the standard shader. If you create your own material, it is automatically assigned the standard shader.

You can easily access the Standard Shader. If you create your own material the Standard Shader is assigned to it. There are a number of other built-in shaders that are very useful for beginners. You can see all the available built-in shaders divided into families by clicking on the **shader** drop down menu in the Inspector.

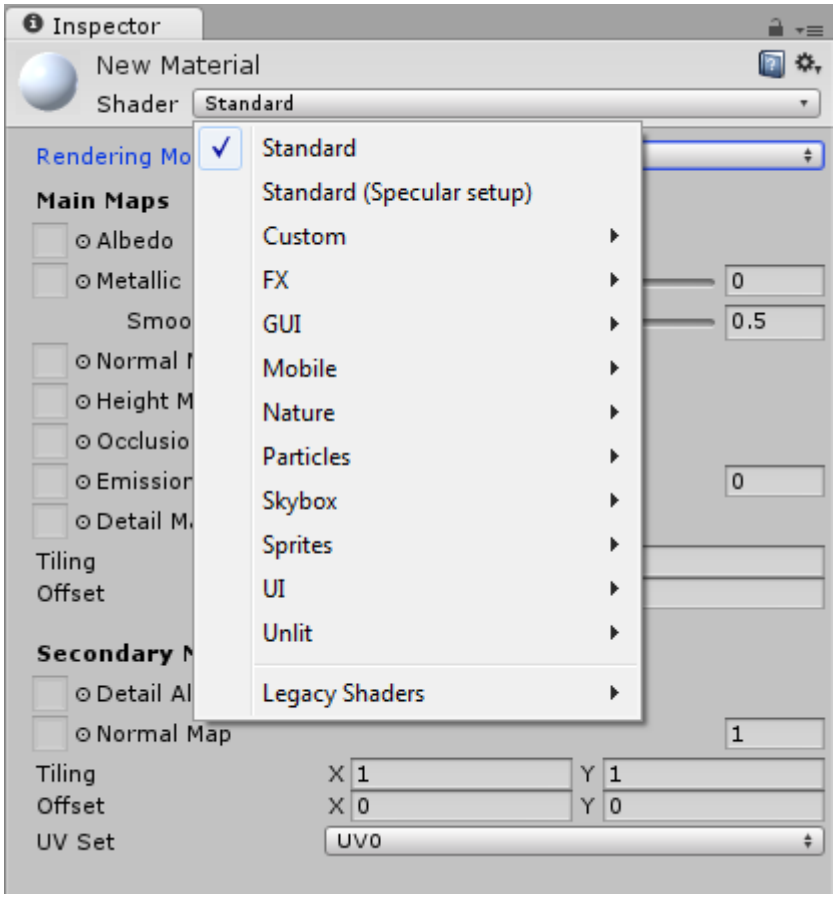

Figure 6-1 Unity built-in shaders

The source code of built-in shaders is available in the Unity download archive *http://unity3d.com/* that contains more than 120 shaders. You can learn a lot from reading and trying to understand the code of these shaders.

In addition to these, there are many effects that cannot be achieved by using existing shaders. For example, shaders that implement reflections based on local cubemaps. For more information see *6.2 Implementing reflections with a local cubemap* on page 6-84.

In Unity there are typically two ways of writing shaders:

Surface shaders
: These are commonly used when shaders are affected by lights and shadows. Unity does the work related to the lighting model for you, enabling you to write more compact shaders.

Vertex and fragment shaders
: These are the most flexible shaders but you must implement everything. The Unity ShaderLab does more than vertex and fragment shaders but these are in the main programmable part of the graphics pipeline where all the shading is done so it is important to know how to write custom vertex and fragment shaders.

6.1.2 Shader structure

The following code shows a very simple vertex and fragment shader that contains most of the elements required in vertex or fragment shader.

The shader example is written in Cg. Unity also supports the HLSL language for shader snippets.

```
Shader "Custom/ctTextured"
{
    Properties
    {
```

```
        _AmbientColor ("Ambient Color", Color) = (0.2,0.2,0.2,1.0)
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }

    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma target 3.0
            #pragma glsl
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            // User-specified uniforms
            uniform float4 _AmbientColor;
            uniform sampler2D _MainTex;

            struct vertexInput
            {
                float4 vertex : POSITION;
                float4 texCoord : TEXCOORD0;
            };
            struct vertexOutput
            {
                float4 pos : SV_POSITION;
                float4 tex : TEXCOORD0;
            };

            // Vertex shader.
            vertexOutput vert(vertexInput input)
            {
                vertexOutput output;

                output.tex = input.texCoord;
                output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
                return output;
            }

            // Fragment shader.
            float4 frag(vertexOutput input) : COLOR
            {
                float4 texColor = tex2D(_MainTex, float2(input.tex));
                return _AmbientColor + texColor;
            }

            ENDCG
        }
    }
    Fallback "Diffuse"
}
```

The first key word is `Shader` followed by the *`path/name`* of the shader. The path defines the category where the shader is displayed in the drop down menu when you are setting a material. The shader from the example is displayed under the category of **Custom** shaders in the drop down menu.

The `Properties{}` block lists the shader parameters that are visible in the inspector and what parameters you can interact with.

Each shader in Unity consists of a list of `subshaders`. When Unity renders a mesh, it looks for the shader to use, and selects the first `subshader` that can run on the graphics card. This way shaders are executed correctly on different graphics cards that support different shader models. This feature is important because GPU hardware and APIs are constantly evolving. For example, you can write your main shader targeting a Mali Midgard GPU to make use of the latest features of OpenGL ES 3.0, while in a separate subshader, write a replacement shader for graphics cards supporting OpenGL ES 2.0 and below.

The `Pass` block causes the geometry of an object to be rendered one time. A shader can contain one or more passes. You can use multiple passes on old hardware, or to achieve special effects.

If Unity cannot find a `subshader` in the body of the shader that can render the geometry correctly it rolls back to another shader defined after the `Fallback` statement. In the example this is the Diffuse built-in shader.

Cg program snippets are written between `CGPROGRAM` and `ENDCG`.

6.1.3 Compilation Directives

You pass compilation directives as `#pragma` statements. The compilation directives indicate the shader functions to be compiled.

Each compilation directive must contain at least the directives to compile the vertex and the fragment shader: `#pragma vertex name, #pragma fragment name`.

By default, Unity compiles shaders into shader model 2.0. The directive `#pragma target` enables shaders to be compiled into other capability levels. If the shader becomes large you get an error of the following type:

```
Shader error in 'Custom/MyShader': Arithmetic instruction limit of 64 exceeded; 83
arithmetic instructions needed to compile program;
```

If this is the case you must change from shader model 2.0 to shader model 3.0 by adding the `#pragma target 3.0` statement. Shader model 3.0 has a much higher instruction limit.

If you pass several varyings from vertex shader to fragment shader you might get the following error:

```
Shader error in 'Custom/MyShader': Too many interpolators used (maybe you want
#pragma glsl?) at line 75.
```

If this is the case add the compilation directive `#pragma glsl`. This directive converts Cg or HLSL code into GLSL.

The `#pragma only_renderers` directive.

Unity supports several rendering platforms such as `gles`, `gles3`, `opengl`, `d3d11`, `d3d11_9x`, `xbox360`, `ps3` and `flash`. By default, shaders are compiled to all these platforms unless you explicitly limit this number using the `#pragma only_renderers` followed by the render APIs you want leaving a blank space between them.

If you are targeting mobile devices only limit shader compilations to `gles` and `gles3`. You must also add the `opengl` and `d3d11` renderers used by Unity Editor:

```
#pragma only_renderers gles gles3 [opengl, d3d11]
```

6.1.4 Includes

It is possible to add include files in the shader to make use of Unity predefined variables and helper functions.

You can see the available includes in `C:\Program Files \Unity\Editor\Data\CGIncludes`. For example, in the include `UnityCG.cginc` you can find several useful helper functions and macros used in many standard shaders. To use them declare the include in your shader.

A number of Unity built-in variables are available to shaders. They are located in the include `UnityShaderVariables.cginc`. You are not required to include this file in your shader because Unity does this automatically. Several useful transformation matrices and magnitudes are directly available in the shaders. It is important to know all of these to avoid duplicating the work. For example, before considering how to pass a matrix to the shader, camera position or projection parameters or light parameters, check if an include already provides this.

To improve performance, it is sometimes preferable to execute an operation in the CPU and pass the result to the GPU instead of executing it in the vertex shader for every vertex. For example, this is the case of multiplications of matrix uniforms. This is the reason why Unity made available for us as built-in uniforms several compound matrices. Some of the important Unity shader built-in values are shown in the following table:

Table 6-1 Important Unity shader built-in values

Built-in Uniform	Description
UNITY_MATRIX_V	Current view matrix
UNITY_MATRIX_P	Current projection matrix
Object2World	Current model matrix
_World2Object	Inverse of current world matrix
UNITY_MATRIX_VP	Current view * projection matrix
UNITY_MATRIX_MV	Current model * view matrix
UNITY_MATRIX_MVP	Current model * view * projection matrix
UNITY_MATRIX_IT_MV	Invert transpose of current model * view matrix
_WorldSpaceCameraPos	Camera position in world space
_ProjectionParams	Near and far planes and 1/farPlane as components of a vector
_Time	Current time and fractions in a vector (t/20, t, t*2, t*3)

6.1.5 The OpenGL ES 3.0 graphics pipeline

It is important to know where in the graphic pipeline the programmable vertex and fragment shaders are located.

The following figure shows a schematic view of the OpenGL ES 3.0 graphic pipeline flow. OpenGL ES 3.0 is a major step in the evolution of embedded graphics and is derived from the OpenGL 3.3 specification.

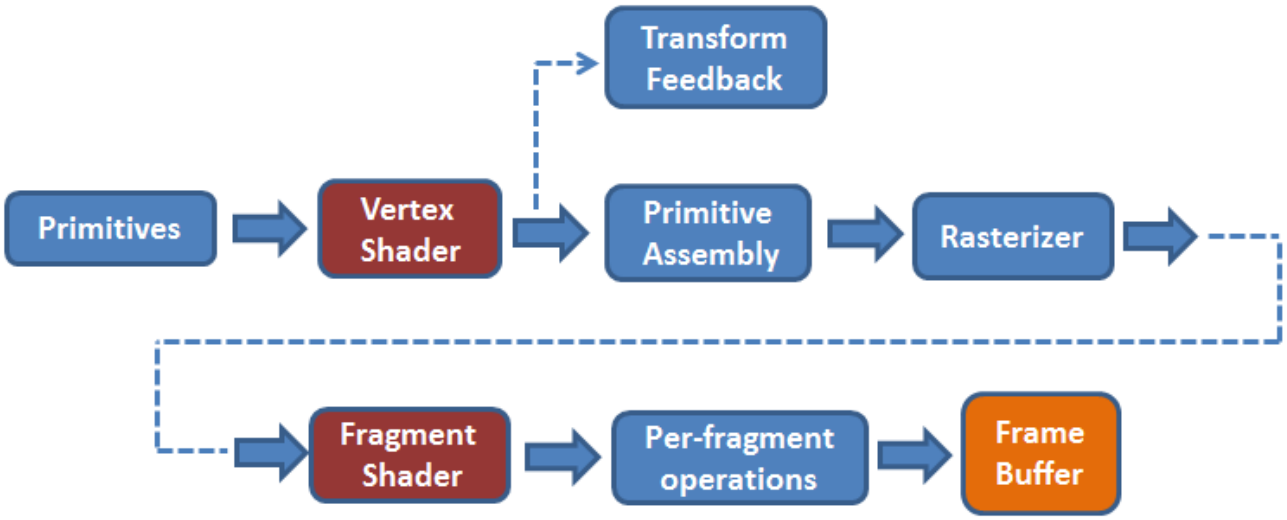

Figure 6-2 OpenGL ES 3.0 Programmable Pipeline

Primitives

In the primitives stage the pipeline operates on the geometric primitives described by vertices, points, lines and polygons.

Vertex Shader

The vertex shader implements a general-purpose programmable method for operating on vertices. The vertex shader transforms and lights vertices.

Primitive assembly
In primitive assembly the vertices are assembled into geometric primitives. The resulting primitives are clipped to a clipping volume and sent to the rasterizer.

Rasterization
Output values from the vertex shader are calculated for every generated fragment. This process is known as interpolation. During rasterization, the primitives are converted into a set of two-dimensional fragments that are then sent to the fragment shader.

Transform feedback
Transform feedback, enables writing selective writing to an output buffer that the vertex shader outputs and is later sent back to the vertex shader. This feature is not exposed by Unity but it is used internally, for example, to optimize the skinning of characters.

Fragment shader
The fragment shader implements a general-purpose programmable method for operating on fragments before they are sent to the next stage.

Per-fragment operations
In Per-fragment operations several functions and tests are applied on each fragment: pixel ownership test, scissor test, stencil and depth tests, blending and dithering. As a result of this per-fragment stage either the fragment is discarded or the fragment color, depth or stencil value is written to the frame buffer in screen coordinates.

6.1.6 Vertex shaders

The vertex shader example runs once for each vertex of the geometry. The purpose of the vertex shader is to transform the 3D position of each vertex, given in the local coordinates of the object, to the projected 2D position in screen space and calculate the depth value for the Z-buffer.

The vertex shader example sample code is in *6.1.2 Shader structure* on page 6-72.

The transformed position is expected in the output of the vertex shader. If the vertex shader does not return a value the console displays the following error:

```
Shader error in 'Custom/ctTextured': '' : function does not return a value: vert at
line 36
```

In the example, the vertex shader receives as input, the vertex coordinates in local space and the texture coordinates. Vertex coordinates are transformed from local to screen space using the Model View Projection matrix `UNITY_MATRIX_MVP` that is a Unity built-in value:

```
output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
```

Texture coordinates are passed to fragment shaders as a varying but this it does not mean that they are not transformed.

Normals are transformed from object space to world space in a different manner. To guarantee that the normal is still normal to the triangle after a non-uniform scaling operation, it must be multiplied by the transpose of the inverse of the transformation matrix. To apply the transpose operation you flip the order of factors in the multiplication. The inverse of the local to world matrix is the built-in `World2Object` Unity matrix. It is a 4x4 matrix so you must build a 4 component vector from the 3 component normal input vector.

```
float4 normalWorld = mul(float4(input.normal, 0.0), _World2Object);
```

When building the four component vector you add a zero as the fourth component. This is necessary to handle vector transformation correctly in the fourth dimensional space while for coordinates the fourth component must be a unit.

You can skip the process of normal transformation if normals are supplied already in world coordinates. This saves work in the vertex shader. Avoid this hint if the object mesh could potentially be handled by any Unity built in shader because in this case normals are expected in object coordinates.

Most of the graphics effects are implemented in the fragment shader but you can also do some effects in the vertex shader. Vertex Displacement Mapping, also known as Displacement Mapping is a well-known

technique enabling you to deform a polygonal mesh using a texture to add surface detail, for example, in terrain generation using height maps. To have access in the vertex shader to this texture, also known as displacement map, you must add the pragma directive `#pragma target 3.0` because it is only available in shader model 3.0. According to the shader model 3.0 at least 4 texture units must be accessible inside the vertex shader. If you force the editor to use the OpenGL renderer then you must also add the `#pragma glsl` directive. If you do not declare this directive the error message produced suggests it:

```
Shader error in 'Custom/ctTextured': function "tex2D" not supported in this profile
(maybe you want #pragma glsl?) at line 57
```

In the vertex shader you also can animate vertices using “procedural animation” techniques. You can use the time variable in shaders enabling you to modify the vertex coordinates as a function of time. Mesh skinning is another type of functionality implemented in the vertex shader. Unity uses this to animate the vertices of the meshes associated with character skeletons.

6.1.7 Vertex shader input

The input and output of the vertex shader are defined by means of structures. In the input structure of the example you declare only the vertex attributes position and texture coordinates.

You can define more attributes as input, for example a second set of texture coordinates, normals in object coordinates, colors and tangents, using the following semantics.

```
struct vertexInput
{
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 texcoord1 : TEXCOORD1;
    fixed4 color : COLOR;
};
```

A semantic is a string attached to a shader input or output that provides information about the use of a parameter. You must specify a semantic for all variables passed between shader stages.

If you use incorrect semantics such as `float3 tangent2 : TANGENTIAL`, you get an error of the following type:

```
Shader error in 'Custom/ctTextured': unknown semantics "TANGENTIAL" specified for
"tangent2" at line 32
```

For performance, only specify the parameters in the input structure that you strictly require. Unity has some predefined input structures for the most common cases of input parameter combinations: `appdata_base`, `appdata_tan` and `appdata_full`. These are described in the `UnityCG.cginc` include file. The previous vertex input structure example corresponds to `appdata_full`. In this case you are not required to declare the structure, only declare the include file.

6.1.8 Vertex shader output and varyings

Vertex shader output is defined in an output structure that must contain the vertex transformed coordinates. In the following example, the output structure is very simple but you can add other magnitudes.

The following code lists the semantics supported by Unity:

```
struct vertexOutput
{
    float4 pos : SV_POSITION;
    float4 tex : TEXCOORD0;
    float4 texSpecular : TEXCOORD1;
    float3 vertexInWorld : TEXCOORD2;
    float3 viewDirInWorld : TEXCOORD3;
    float3 normalInWorld : TEXCOORD4;
    float3 vertexToLightInWorld : TEXCOORD5;
    float4 vertexInScreenCoords :  TEXCOORD6;
    float4 shadowsVertexInScreenCoords :  TEXCOORD7;
};
```

The transformed vertex coordinates are defined with the semantic `SV_POSITION`. Two textures, several vectors, and coordinates in different spaces calling the semantic `TEXCOORDn` are also passed to the fragment shader.

`TEXCOORD0` is typically reserved for UVs and `TEXCOORD1` for lightmap UVs, but technically you can send anything from `TEXCOORD0` to `TEXCOORD7` to the fragment shader. It is important to notice that each interpolator, that is each semantic, can only process a maximum of 4 floats. Put larger variables such as matrices into multiple interpolators. This means that if you define a matrix to be passed as a varying: `float4x4 myMatrix : TEXCOORD2` , Unity uses the interpolators from `TEXCOORD2` to `TEXCOORD5`.

Everything you send from the vertex shader to the fragment shader is linearly interpolated by default. For every pixel in the triangle defined by the vertices `V1`, `V2` and `V3` the rasterizer, located in the graphic pipeline between vertex and fragment shaders, calculates the pixel coordinates as a linear interpolation of the vertices coordinates using the barycentric coordinates λ1, λ2 and λ3.

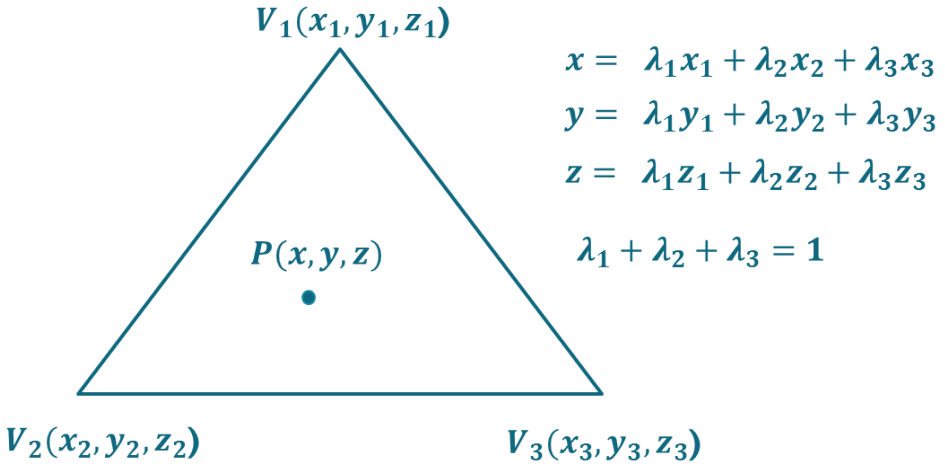

Figure 6-3 Linear interpolation using barycentric coordinates

The following diagram shows the results of color interpolation in a triangle with vertex colors red, green and blue.

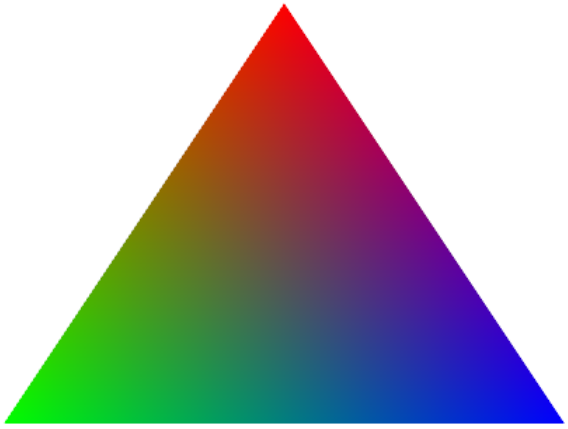

Figure 6-4 Color Interpolation

The same interpolation is applied to any varying passed from the vertex to the fragment shader. This is a very powerful tool because there is a hardware linear interpolator. For example, if you have a plane and you want to apply a color as a function of the distance to the center C, you pass the coordinate of the center C to the vertex shader, calculate the squared distance from the vertex to C and pass that magnitude to the fragment shader. The value of the distance is automatically interpolated for you in every pixel of every triangle.

Values are linearly interpolated so it is possible to perform per-vertex computations and reuse them in the fragment shader, that is, a value that can be linearly interpolated in the fragment shader can be calculated in the vertex shader instead. This can provide a substantial performance boost because the vertex shader runs on a much smaller data set than the fragment shader.

You must be careful with the use of varyings especially in mobiles where performance and memory bandwidth consumption are critical to the success of many games. The more varyings there are, the more vertex accesses and fragment shader varying reads bandwidth. Aim for a reasonable balance when using varyings.

6.1.9 Fragment Shaders

The fragment shader is the graphics pipeline stage after primitive rasterization.

For each sample of the pixels covered by a primitive, a fragment is generated. The fragment shader code is executed for each generated fragment. There are many more fragments than vertices so you must take care about the number of operations performed in the fragment shader.

In the fragment shader you can access the fragment coordinates in the windows space among other values that contains all interpolated per-vertex output values from the vertex shader.

In the shader example in *6.1.2 Shader structure* on page 6-72, the fragment shader receives the interpolated texture coordinates from the vertex shader and performs a texture lookup to obtain the color at these coordinates. It combines this color with the ambient color to produce the final output color. From the declaration of the fragment shader `float4 frag(vertexOutput input) : COLOR` it is clear that it is expected to produce the fragment color. The fragment shader is where you do the operations to achieve the required effect. This ultimately consists of assigning the correct color to a fragment.

6.1.10 Providing data to shaders

Any data declared as a uniform in the Pass block is available to both vertex and fragment shaders.

A uniform can be considered as a type of global constant variable because it cannot be modified inside the shader.

You can supply this uniform to the shader in the following ways:

- Using the `Properties` block.
- Programmatically from a script.

The `Properties` block enables you to define uniforms interactively in the **Inspector**. Any variable declared in the `Properties` block appears in the material inspector listed with the variable name.

The following code shows the `Properties` block of the shader example associated to material `ctSphereMat`:

```
Properties
{
    _AmbientColor ("Ambient Color", Color) = (0.2,0.2,0.2,1.0)
    _MainTex ("Base (RGB)", 2D) = "white" {}
}
```

The variables `_AmbientColor` and `_MainTex` declared in the `Properties` block with the names `Ambient Color` and `Base (RGB)` respectively appear in the **Material Inspector** with those names.

The following figure shows Properties in Material Inspector:

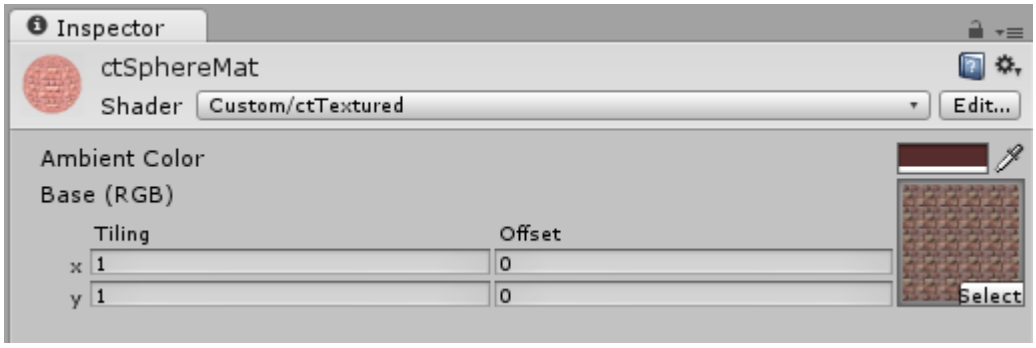

Figure 6-5 Properties in Material Inspector

Passing data to the shader by means of the `Properties` block is very useful especially when you are in the development stage of the shader because you can change the data interactively and see the effect at run time.

You can put the following types of variables in the `Properties` block:

- Float.
- Color.
- Texture 2D.
- Cubemap.
- Rectangle.
- Vector.

The `Properties` block is not a useful way of passing data if for example, data is required from a previous calculation or data is required to be passed at specific point in time.

An alternative method of passing data to the shaders is programmatically from a script.

The material class exposes several methods that you can use to pass data associated with a material to a shader. The following table lists the most common methods:

Table 6-2 Common methods for passing data associated with a material to a shader

Method
`SetColor (propertyName: string, color: Color);`
`SetFloat (propertyName: string, value: float);`
`SetInt (propertyName: string, value: int);`
`SetMatrix (propertyName: string, matrix: Matrix4x4);`
`SetVector (propertyName: string, vector: Vector4);`
`SetTexture (propertyName: string, texture: Texture);`

In the following code, immediately before the main camera renders the scene, a secondary camera `shwCam` renders the shadows to a texture to be combined with the main camera render pass.

For the shadow texture projection process the vertices must be transformed in a convenient manner. The shadow camera projection matrix (`shwCam.projectionMatrix`), world to local transformation matrix (`shwCam.transform.worldToLocalMatrix`), and the rendered shadow texture (`m_ShadowsTexture`) are passed to the shader.

These values are available in the shader as uniforms with the names `_ShwCamProjMat`, `_ShwCamViewMat` and `m_ShadowsTexture`.

The following code shows how matrices and textures are sent to the shader by means of materials contained in the list `shwMats`.

```
// Called before object is rendered.

public void OnWillRenderObject()
{
    // Perform different checks.
    ...
    CreateShadowsTexture();
    // Set up shadows camera shwCam.
    ...
    // Pass matrices to the shader
    for(int i = 0; i < shwMats.Count; i++)
    {
        shwMats[i].SetMatrix("_ShwCamProjMat", shwCam.projectionMatrix);
        shwMats[i].SetMatrix("_ShwCamViewMat", shwCam.transform.worldToLocalMatrix);
    }
    // Render shadows texture
    shwCam.Render();
    for(int i = 0; i < shwMats.Count; i++)
    {
        shwMats[i].SetTexture( "_ShadowsTex", m_ShadowsTexture );
    }
    s_RenderingShadows = false;
}
```

6.1.11 Debugging shaders in Unity

In Unity it is not possible to debug shaders in the same way as you do with traditional code. You can however use the output of the fragment shader to visualize the values you want to debug. You then have to interpret the image produced.

The following figure shows the output of the shader `ctReflLocalCubemap.shader` applied to the reflective surface of the floor from *6.2 Implementing reflections with a local cubemap* on page 6-84:

Figure 6-6 Chess room with reflections

In following fragment shader, the output color has been replaced by the normalized local corrected reflected vector:

```
return float4(normalize(localCorrReflDirWS), 1.0);
```

Instead of the reflected image, it visualizes the components of the reflected vector normalized as colors.

The reddish color zone in the floor indicates that the reflected vector has a strong X component, that is, it is mostly oriented to toward the X Axis. The reddish part shows the reflection coming from that direction, that is, from the windowed wall.

The blueish zone indicates a predominance of reflected vectors oriented to Z axis, that is, the reflection from the right wall.

In the black zone the vectors are mainly oriented to –Z but the colors can only have positive components because the negative components are clamped to 0.

The following figure shows the result of replacing the output color of the fragment by the normalized local reflected vector:

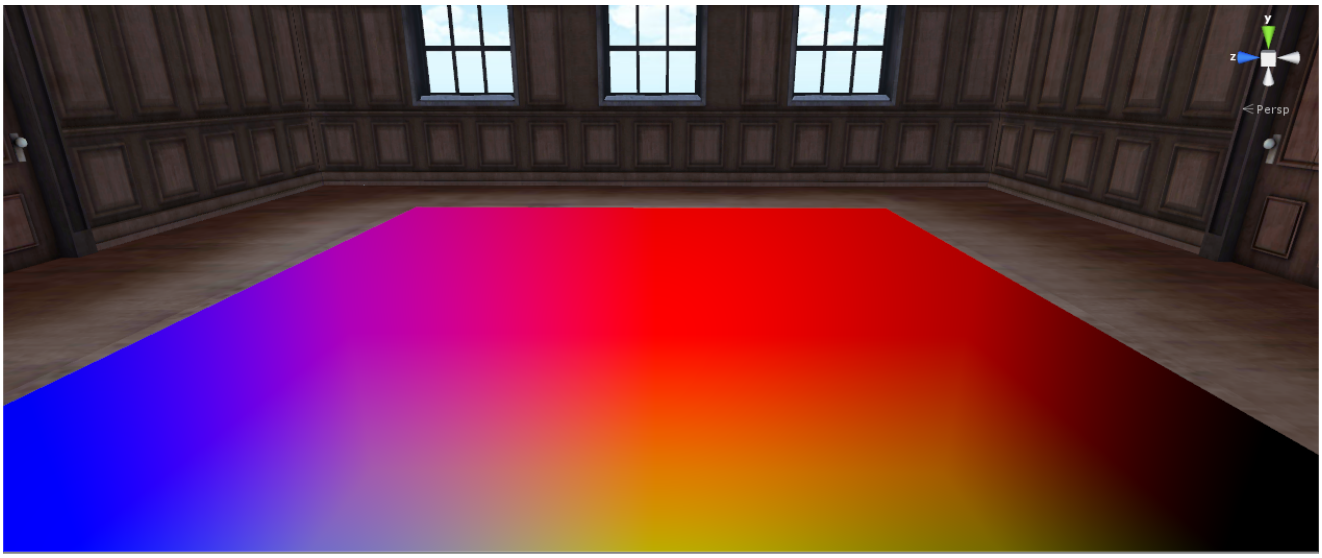

Figure 6-7 Shader debugging with multiple colors

It might initially be difficult to interpret the meaning of the colors while debugging so try to focus on a single color component. For example, you can return only the Y component of the normalized local corrected reflected vector:

```
float3 normLocalCorrReflDirWS = normalize(localCorrReflDirWS);
return float4(0, normLocalCorrReflDirWS.y, 0, 1);
```

In this case, the output is only the reflections coming mainly from the roof above the camera. That is, the part of the room oriented to the Y axis. The reflections from the walls of the room are coming from X, Z and –Z directions, so they are rendered in black.

The following figure shows shader debugging with a single color:

Figure 6-8 Shader debugging with a single color

Check that the magnitude you are debugging with color, is between 0 and 1 because any other value is automatically clamped. Any negative value is assigned zero and any value greater than 1 is assigned 1.

6.2 Implementing reflections with a local cubemap

Reflections based on a local cubemap is a useful technique for rendering reflections on mobile devices.

Unity version 5 and higher implements reflections based on local cubemaps as `Reflection Probes`. You can combine these with other types of reflections, such as reflections rendered at runtime in your own custom shader.

This section contains the following subsections:

6.2.1 History of reflection implementations

Graphics developers have always tried to find computationally cheap methods to implement reflections.

One of the first solutions is spherical mapping. This technique simulates reflections or lighting on objects without using expensive ray-tracing or lighting calculations.

The following figure shows an environment map on a sphere:

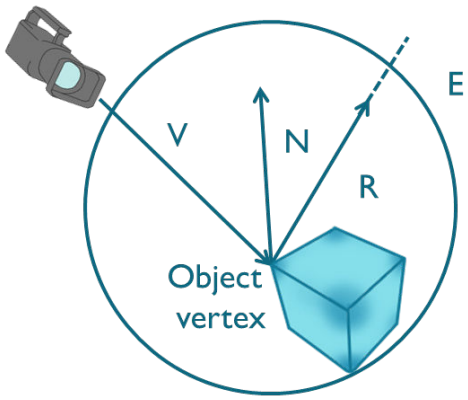

Figure 6-9 Environment map on a sphere

The following figure shows the equation for mapping a spherical surface into two dimensions:

The spherical surface is mapped into 2D:

$$u = \frac{R_x}{m} + \frac{1}{2}$$

$$v = \frac{R_y}{m} + \frac{1}{2}$$

$$m = 2 \cdot \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2}$$

Figure 6-10 Spherical surface 2D mapping equation

This approach has several disadvantages, but the main problem is the distortions that occur when mapping a picture onto a sphere. In 1999, it became possible to use cubemaps with hardware acceleration. Cubemaps solved the problems of image distortions, viewpoint dependency and computational inefficiency related to spherical mapping.

The following figure shows an unfolded cube:

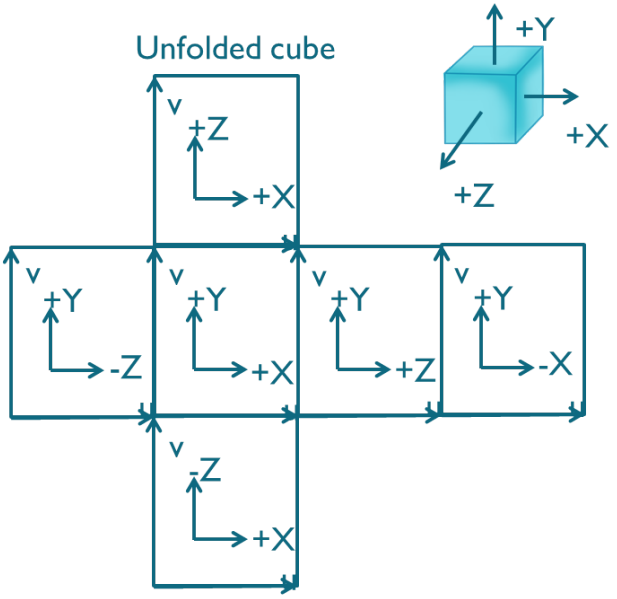

Figure 6-11 Unfolded cube

Cubemapping uses the six faces of a cube as the map shape. The environment is projected onto each side of a cube and stored as six square textures, or unfolded into six regions of a single texture. The cubemap is generated by rendering the scene from a given position with six different camera orientations with a 90 degree view frustum representing each a cube face. Source images are sampled directly. No distortion is introduced by resampling into an intermediate environment map.

The following figure shows infinite reflections:

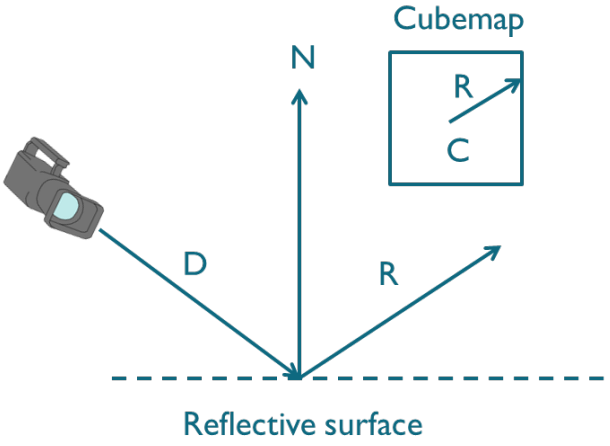

Figure 6-12 Infinite reflections

To implement reflections based on cubemaps, evaluate the reflected vector `R` and use it to fetch the texel from the cubemap `_Cubemap` using the available texture lookup function `texCUBE()`:

```
float4 color = texCUBE(_Cubemap, R);
```

The normal `N` and view vector `D` are passed to fragment shader from the vertex shader. The fragment shader fetches the texture color from the cubemap:

```
float3 R = reflect(D, N);
float4 color = texCUBE(_Cubemap, R);
```

This approach can only reproduce reflections correctly from a distant environment where the cubemap position is not relevant. This simple and effective technique is mainly used in outdoor lighting, for example, to add reflections of the sky.

The following figure shows incorrect reflections:

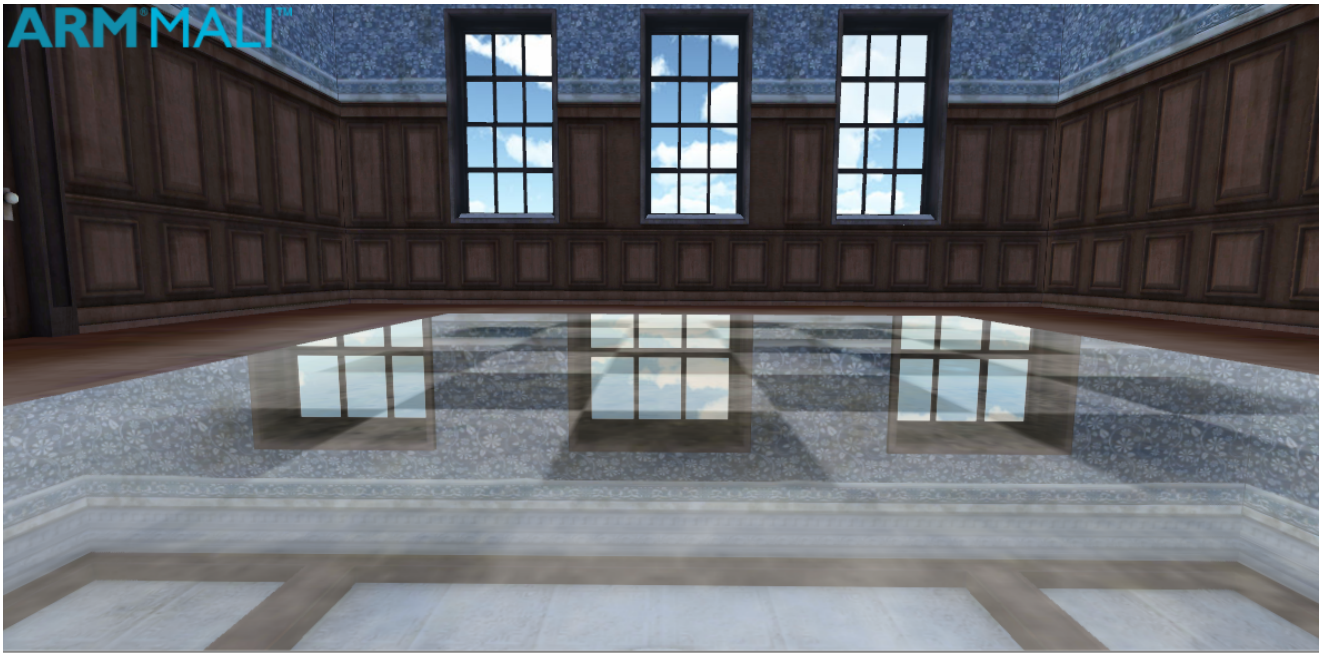

Figure 6-13 Incorrect reflections

If you use this technique in a local environment it produces inaccurate reflections. The reason why the reflections are incorrect is that in the expression `float4 color = texCUBE(_Cubemap, R);` there is no binding to the local geometry. For example, if you walk across a reflective floor looking at it from the same angle you always see the same reflection. The reflected vector is always the same and the expression always produces the same result. This is because the direction of the view vector does not change. In the real world reflections depend on both viewing angle and viewing position.

6.2.2 Generating correct reflections with a local cubemap

A solution to this problem involves binding to the local geometry in the procedure to calculate the reflection.

This solution is described in *GPU Gems: Programming Techniques, Tips, and Tricks for Real-time Graphics* by Randima Fernando (Series Editor).

The following figure shows a local correction using a bounding sphere:

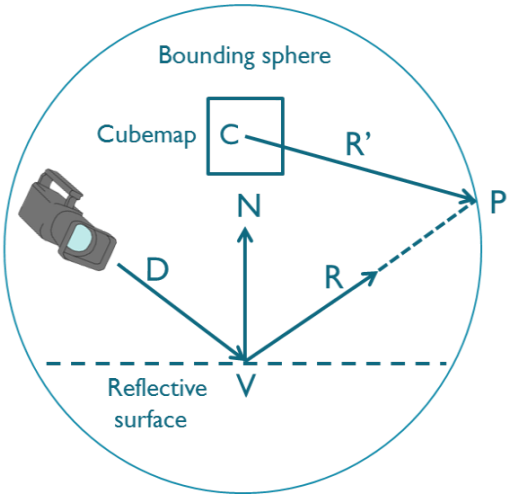

Figure 6-14 Local correction using a bounding sphere

A bounding sphere is used as a proxy volume that delimits the scene to be reflected. Instead of using the reflected vector `R` to fetch the texture from the cubemap a new vector `R'` is used. To build this new vector you find the intersection point `P` in the bounding sphere of the ray from the local point `V` in the direction of the reflected vector `R`. Create a new vector `R'` from the center of the cubemap `C`, where the cubemap was generated, to the intersection point `P`. Use this vector to fetch the texture from the cubemap.

```
float3 R = reflect(D, N);
Find intersection point P
Find vector R' = CP
float4 col = texCUBE(_Cubemap, R');
```

This approach produces good results in the surfaces of objects with a near spherical shape but reflections in plane reflective surfaces are deformed. Another drawback of this method is that the algorithm to calculate the intersection point with the bounding sphere solves a second degree equation and this is relatively complex.

In 2010 a developer proposed a better solution in a forum at *http://www.gamedev.net*. This approach replaces the previous bounding sphere by a box and solves the deformations and complexity problems of the previous method. For more information see: *http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/?&p=4637262*.

The following figure shows a local correction using a bounding box:

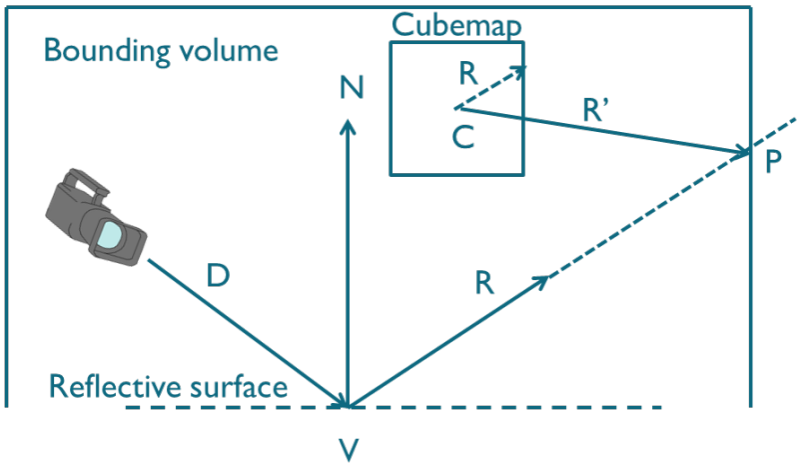

Figure 6-15 Local correction using a bounding box

A more recent work in 2012 by Sebastien Lagarde uses this new approach to simulate more complex ambient specular lighting using several cubemaps and uses an algorithm to evaluate the contribution of each cubemap and efficiently blend on the GPU. See *http://seblagarde.wordpress.com*

Table 6-3 Differences between infinite and local cubemaps

Infinite Cubemaps	Local Cubemaps
• Mainly used outdoors to represent the lighting from a distant environment. • Cubemap position is not relevant.	• Represents the lighting from a finite local environment. • Cubemap position is relevant. • The lighting from these cubemaps is right only at the location where the cubemap was created. • Local correction must be applied to adapt the intrinsic infinite nature of cubemaps to local environment.

The following figure shows the scene with correct reflections generated with local cubemaps.

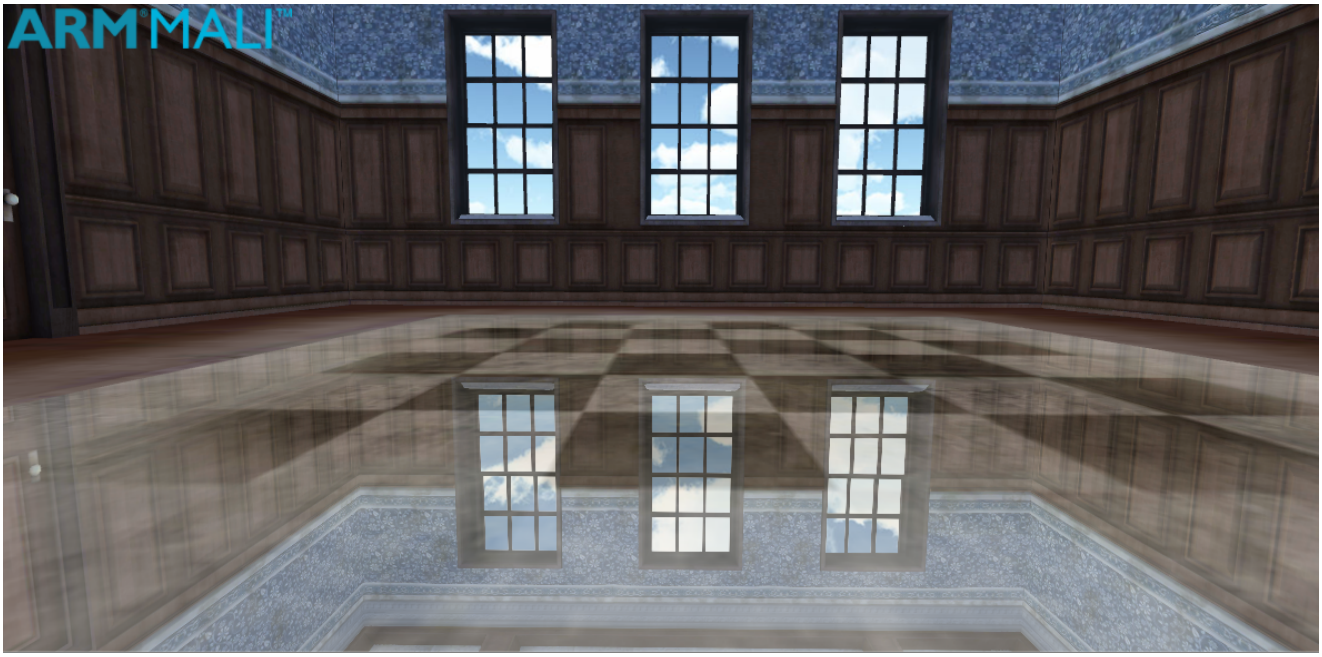

Figure 6-16 Correct reflections

6.2.3 Shader Implementation

This section describes shaders that implement reflections using local cubemaps.

The vertex shader calculates three magnitudes that are passed to the fragment shader as interpolated values:

- The vertex position.
- The view direction.
- The normal.

These values are in world coordinates.

The following code shows a shader implementation of reflections using local cubemaps, for Unity.

```
vertexOutput vert(vertexInput input)
{
    vertexOutput output;
    output.tex = input.texcoord;
    // Transform vertex coordinates from local to world.
    float4 vertexWorld = mul(_Object2World, input.vertex);
    // Transform normal to world coordinates.
    float4 normalWorld = mul(float4(input.normal,0.0), _World2Object);
```

```
    // Final vertex output position.  output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
    // ----------- Local correction ------------
    output.vertexInWorld = vertexWorld.xyz;
    output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
    output.normalInWorld = normalWorld.xyz;
    return output;
}
```

The intersection point in the volume box and the reflected vector are computed in the fragment shader. You build new local corrected reflection vector and use it to fetch the reflection texture from the local cubemap. You then combine the texture and reflection to produce the output color:

```
float4 frag(vertexOutput input) : COLOR
{
    float4 reflColor = float4(1, 1, 0, 0);
    // Find reflected vector in WS.
    float3 viewDirWS = normalize(input.viewDirInWorld);
    float3 normalWS = normalize(input.normalInWorld);
    float3 reflDirWS = reflect(viewDirWS, normalWS);
    // Working in World Coordinate System.
    float3 localPosWS = input.vertexInWorld;
    float3 intersectMaxPointPlanes = (_BBoxMax - localPosWS) / reflDirWS;
    float3 intersectMinPointPlanes = (_BBoxMin - localPosWS) / reflDirWS;
    // Looking only for intersections in the forward direction of the ray.
    float3 largestParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);
    // Smallest value of the ray parameters gives us the intersection.
    float distToIntersect = min(min(largestParams.x, largestParams.y), largestParams.z);
    // Find the position of the intersection point.
    float3 intersectPositionWS = localPosWS + reflDirWS * distToIntersect;
    // Get local corrected reflection vector.
    Float3 localCorrReflDirWS = intersectPositionWS - _EnviCubeMapPos;
    // Lookup the environment reflection texture with the right vector.
    reflColor = texCUBE(_Cube, localCorrReflDirWS);
    // Lookup the texture color.
    float4 texColor = tex2D(_MainTex, float2(input.tex));
    return _AmbientColor + texColor * _ReflAmount * reflColor;
}
```

In the previous code for the fragment shader, the magnitudes `_BBoxMax` and `_BBoxMin` are the maximum and minimum points of the bounding volume. The variable `_EnviCubeMapPos` is the position where the cubemap was created. Pass these values to the shader from the following script:

```
[ExecuteInEditMode]
public class InfoToReflMaterial : MonoBehaviour
{
    // The proxy volume used for local reflection calculations.
    public GameObject boundingBox;

    void Start()
    {
        Vector3 bboxLenght = boundingBox.transform.localScale;
        Vector3 centerBBox = boundingBox.transform.position;

        // Min and max BBox points in world coordinates
        Vector3 BMin = centerBBox - bboxLenght/2;
        Vector3 BMax = centerBBox + bboxLenght/2;

        // Pass the values to the material.
        gameObject.renderer.sharedMaterial.SetVector("_BBoxMin", BMin);
        gameObject.renderer.sharedMaterial.SetVector("_BBoxMax", BMax);
        gameObject.renderer.sharedMaterial.SetVector("_EnviCubeMapPos", centerBBox);
    }
}
```

Pass the values for `_AmbientColor`, `_ReflAmount`, the main texture, and cubemap texture to the shader as uniforms from the properties block:

```
Shader "Custom/ctReflLocalCubemap"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" { }
        _Cube("Reflection Map", Cube) = "" {}
        _AmbientColor("Ambient Color", Color) = (1, 1, 1, 1)
        _ReflAmount("Reflection Amount", Float) = 0.5
    }
```

```
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma glsl
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            // User-specified uniforms
            uniform sampler2D _MainTex;
            uniform samplerCUBE _Cube;
            uniform float4 _AmbientColor;
            uniform float _ReflAmount;
            uniform float _ToggleLocalCorrection;
            // ----Passed from script InfoRoReflmaterial.cs --------
            uniform float3 _BBoxMin;
            uniform float3 _BBoxMax;
            uniform float3 _EnviCubeMapPos;

            struct vertexInput
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 texcoord : TEXCOORD0;
            };
            struct vertexOutput
            {
                float4 pos : SV_POSITION;
                float4 tex : TEXCOORD0;
                float3 vertexInWorld : TEXCOORD1;
                float3 viewDirInWorld : TEXCOORD2;
                float3 normalInWorld : TEXCOORD3;
            };

        Vertex shader    { }
        Fragment shader    { }

        ENDCG
        }
    }
}
```

The algorithm to calculate the intersection point in the bounding volume is based on the use of the parametric representation of the reflected ray from the local position or fragment. For a description of the ray-box intersection algorithm, see *6.2.5 Ray-box intersection algorithm* on page 6-94.

6.2.4 Filtering cubemaps

One of the advantages of implementing reflections using local cubemaps is the fact that the cubemap is static. That is, it is generated during development rather than at run-time. This provides an opportunity to apply any filtering to the cubemap images to achieve an effect.

CubeMapGen is a tool by AMD that applies filtering to cubemaps. You can obtain CubeMapGen from the AMD developer web site at: *http://developer.amd.com*.

To export cubemap images from Unity to CubeMapGen you must save each cubemap image separately. For the source code of a tool that saves the images, see *6.2.6 Source code for editor script to generate cubemaps* on page 6-97. This tool can create a cubemap and can optionally save each cubemap image separately.

You must place the script for this tool in a folder called `Editor` in the Asset directory.

To use the cubemap editor tool:

1. Create the cubemap.
2. Launch the Bake CubeMap Tool from GameObject menu.
3. Provide the cubemap and the camera render position.
4. Optionally save individual images if you plan to apply filtering to the cubemap.

The following figure shows the Bake CubeMap tool interface:

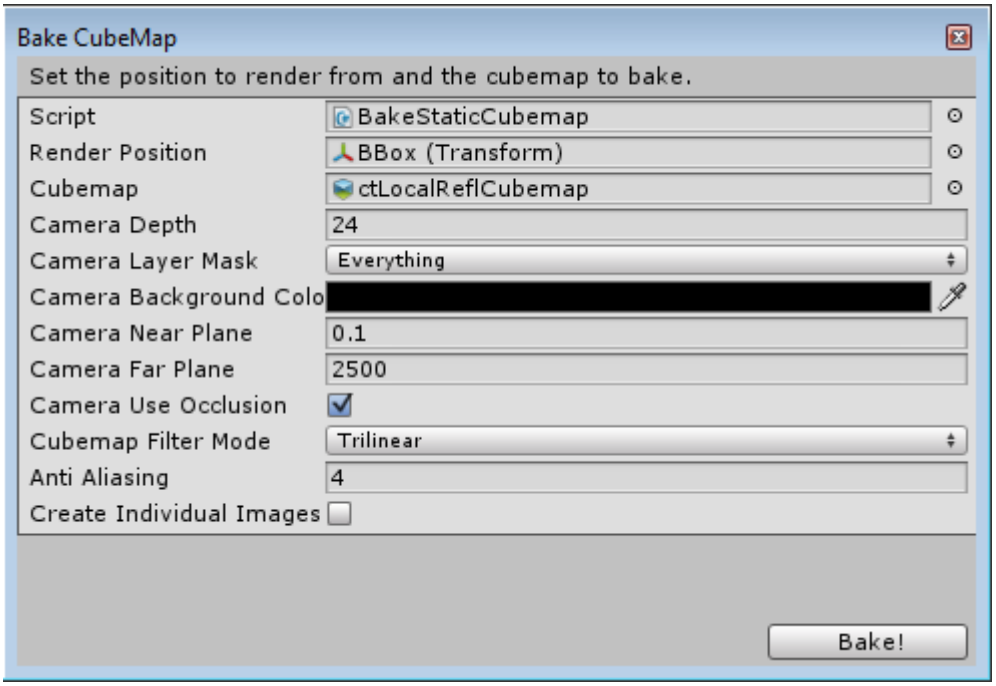

Figure 6-17 Bake CubeMap tool interface

You can load each of the images for the cubemap separately with CubeMapGen.

Select what face to load from the **Select Cube Face** drop down menu and then press **Load Cubemap Face** button. When all faces have been loaded it is possible to rotate the cubemap and check that it is correct.

CubeMapGen has a number of different filtering options in the **Filter Type** drop down menu. Select the filter settings you require and press **Filter Cubemap** to apply the filter. The filtering can take up to several minutes depending on the size of the cubemap. There is no undo option so save the cubemap as a single image before applying any filtering. If the result of the filtering is not what you expect you can reload the cubemap and try adjusting the parameters.

Use the following procedure for importing cubemap images into CubeMapGen:

1. Check the box to save individual images when baking the cubemap.
2. Launch the CubeMapGen tool and load cubemap images following the relations shown in the following table.
3. Save the cubemap as a single dds or cube cross image. Undo is not available so this enables you to reload the cubemap if you are experimenting with filters.
4. Apply filters to cubemap as required until the results are satisfactory.
5. Save the cubemap as individual images.

The following table shows the equivalence of cubemap face index between CubeMapGen and Unity.

Table 6-4 Equivalence of cubemap face index between CubeMapGen and Unity

AMD CubeMapGen	Unity
X+	-X
X-	+X
Y+	+Y
Y-	-Y
Z+	+Z
Z-	-Z

The following figure shows CubeMapGen after loading the six cubemap images:

Figure 6-18 CubeMapGen

The following figure shows the result of CubeMapGen applying a Gaussian filtering to achieve a *frosty* effect:

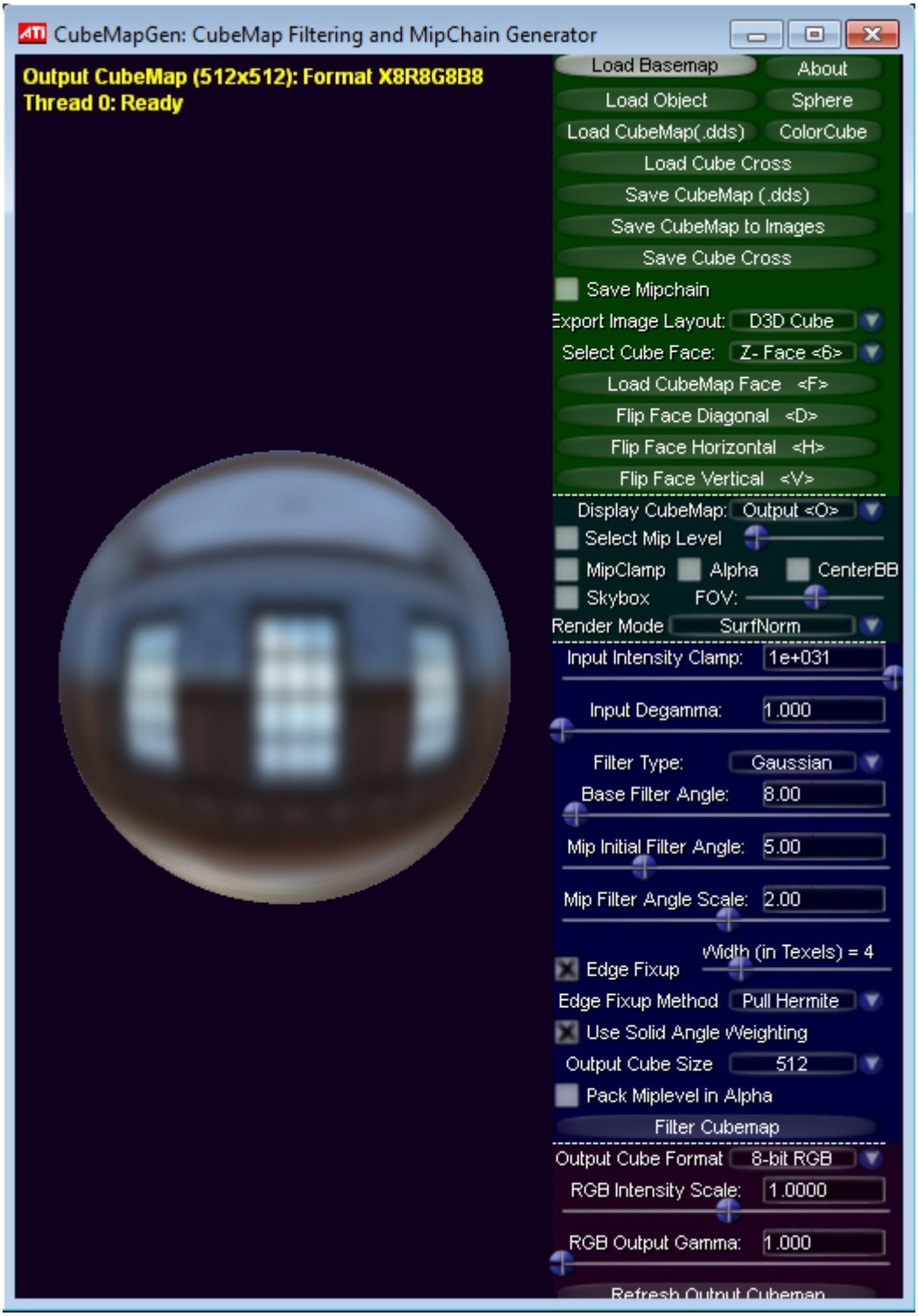

Figure 6-19 CubeMapGen showing frosty effect

The following table shows the filter parameters used with the Gaussian filter to achieve the frosty effect.

Table 6-5 Parameters used in CubeMapGen to produce a frosty effect in the reflections.

Filter settings	Value
Type	Gaussian
Base Filter Angle	8
Mip Initial Filter Angle	5
Mip Filter Angle Scale	2.0
Edge Fixup	Checked
Edge Fixup Width	4

The following figure shows a reflection generated with a cubemap with a frosty effect:

Figure 6-20 Reflection with frosty effect

The following figure summarizes the work flow to apply filtering to Unity cubemaps with the CubeMapGen tool.

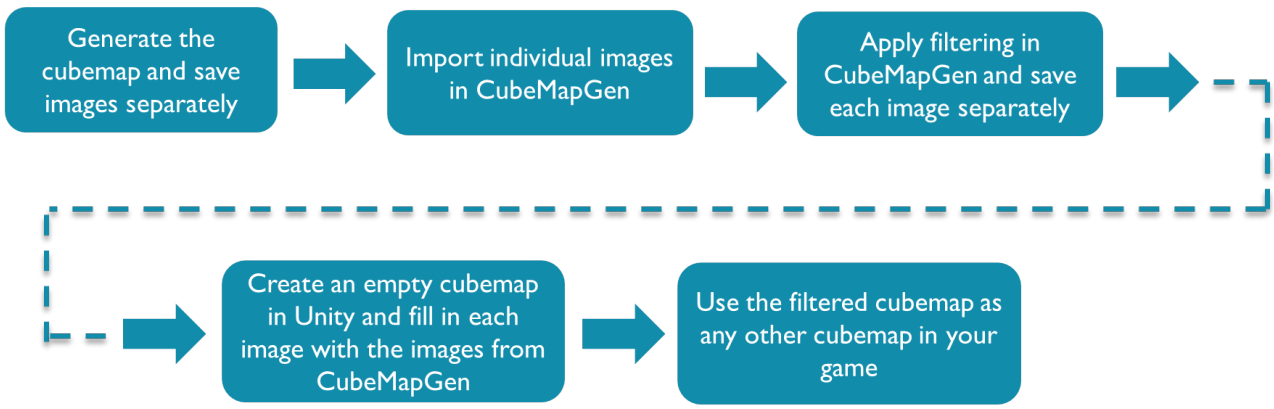

Figure 6-21 Cubemap filtering workflow

6.2.5 Ray-box intersection algorithm

This section describes the ray-box intersection algorithm.

The following figure shows a graph with line equations:

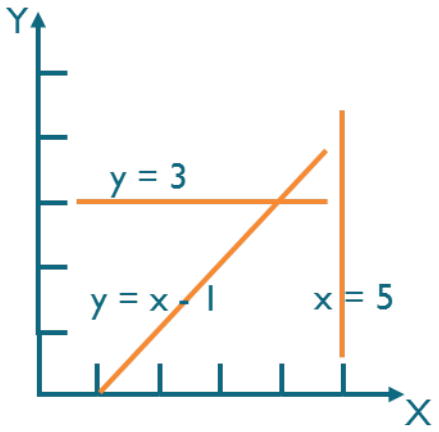

Figure 6-22 Graph with line equations

Equation of a line

```
y = mx + b
```

The vector form of this equation is:

```
r = O + t*D
```

Where:

`O` is the origin point

`D` is the direction vector

`t` is the parameter

The following figure shows an axis aligned bounding box:

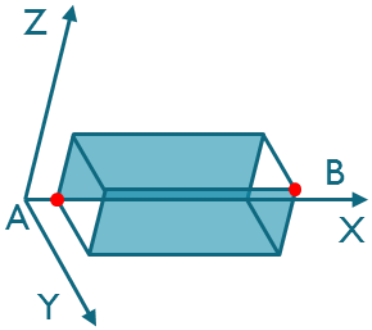

Figure 6-23 Axis aligned bounding box

An axis aligned bounding box `AABB` can be defined by its min and max points `A` and `B`

`AABB` defines a set of lines parallel to coordinate axis. Each component of line can be defined by the following equation:

```
x = Ax; y = Ay; z = Az
x = Bx; y = By; z = Bz
```

To find where a ray intersects one of those lines, equal both equations. For example:

```
Ox + tx*Dx = Ax
```

You can write the solution as:

```
tAx = (Ax - Ox) / Dx
```

Obtain the solution for all components of both intersection points in the same manner:

```
tAx = (Ax - Ox) / Dx
tAy = (Ay - Oy) / Dy
tAz = (Az - Oz) / Dz
tBx = (Bx - Ox) / Dx
tBy = (By - Oy) / Dy
tBz = (Bz - Oz) / Dz
```

In vector form these are:

```
tA = (A - O) / D
tB = (B - O) / D
```

This finds where the line intersects the planes defined by the faces of the cube but it does not guarantee that the intersections lie on the cube.

The following figure shows a 2D representation of ray-box intersection:

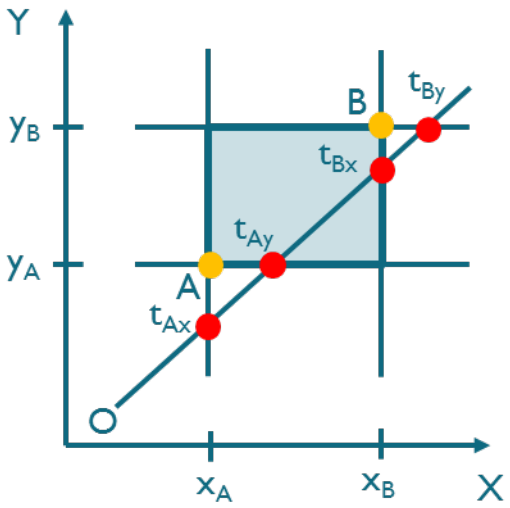

Figure 6-24 Ray-box intersection 2D representation

To find what solution is really an intersection with the box, you require the greater value of the `t` parameter for the intersection at the min plane.

```
tmin = (tAx> tAy) ? tAx: tAy
```

You require the smaller value of the parameter `t` for the intersection at the max plane.

```
tmin = (tAx> tAy) ? tAx: tAy
```

You also must consider those cases when you get no intersections.

The following figure shows a ray-box with no intersection:

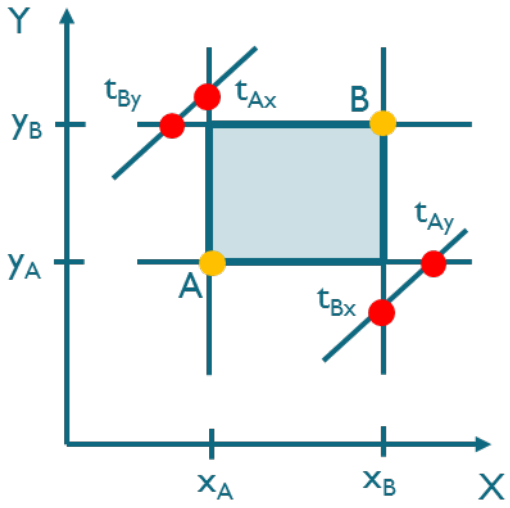

Figure 6-25 Ray-box with no intersection

If you guarantee that the reflective surface is enclosed by the BBox, that is, the origin of the reflected ray is inside the BBox, then there are always two intersections with the box, and the handling of different cases is simplified.

The following figure shows a ray-box intersection in BBox:

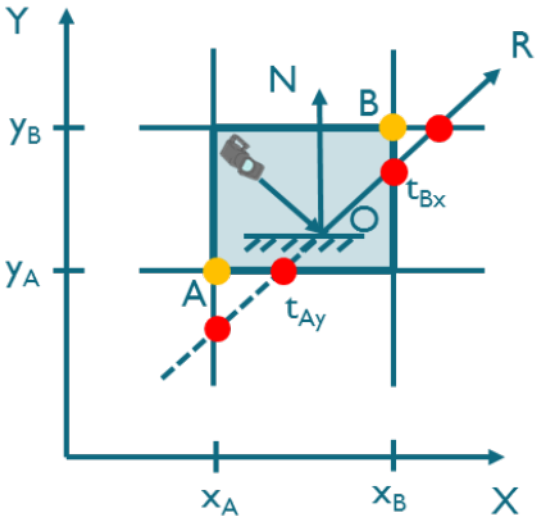

Figure 6-26 Ray-box intersection in BBox

6.2.6 Source code for editor script to generate cubemaps

This section provides the source code for editor script to generate cubemaps.

```
/*
* This confidential and proprietary software may be used only as
* authorised by a licensing agreement from ARM Limited
* (C) COPYRIGHT 2014 ARM Limited
* ALL RIGHTS RESERVED
* The entire notice above must be reproduced on all authorised
* copies and copies may only be made to the extent permitted
* by a licensing agreement from ARM Limited.
*/

using UnityEngine;
using UnityEditor;
using System.IO;

/**
```

```
* This script must be placed in the Editor folder.
* The script renders the scene into a cubemap and optionally
* saves each cubemap image individually.
* The script is available in the Editor mode from the
 * Game Object menu as "Bake Cubemap" option.
* Be sure the camera far plane is enough to render the scene.
*/

public class BakeStaticCubemap : ScriptableWizard
{

    public Transform renderPosition;
    public Cubemap cubemap;
    // Camera settings.
    public int cameraDepth = 24;
    public LayerMask cameraLayerMask = -1;
    public Color cameraBackgroundColor;
    public float cameraNearPlane = 0.1f;
    public float cameraFarPlane = 2500.0f;
    public bool cameraUseOcclusion = true;
    // Cubemap settings.
    public FilterMode cubemapFilterMode = FilterMode.Trilinear;
    // Quality settings.
    public int antiAliasing = 4;

    public bool createIndividualImages = false;

    // The folder where individual cubemap images will be saved
    static string imageDirectory = "Assets/CubemapImages";
    static string[] cubemapImage
         = new string[]{"front+Z", "right+X", "back-Z", "left-X", "top+Y", "bottom-Y"};
           static Vector3[] eulerAngles = new Vector3[]{new Vector3(0.0f,0.0f,0.0f),
           new Vector3(0.0f,-90.0f,0.0f), new Vector3(0.0f,180.0f,0.0f),
           new Vector3(0.0f,90.0f,0.0f), new Vector3(-90.0f,0.0f,0.0f),
           new Vector3(90.0f,0.0f,0.0f)};

    void OnWizardUpdate()
    {
        helpString = "Set the position to render from and the cubemap to bake.";
        if(renderPosition != null && cubemap != null)
        {
            isValid = true;
        }
        else
        {
            isValid = false;
        }
    }

    void OnWizardCreate ()
    {

        // Create temporary camera for rendering.
        GameObject go = new GameObject( "CubemapCam", typeof(Camera) );
        // Camera settings.
        go.camera.depth = cameraDepth;
        go.camera.backgroundColor = cameraBackgroundColor;
        go.camera.cullingMask = cameraLayerMask;
        go.camera.nearClipPlane = cameraNearPlane;
        go.camera.farClipPlane = cameraFarPlane;
        go.camera.useOcclusionCulling = cameraUseOcclusion;
        // Cubemap settings
        cubemap.filterMode = cubemapFilterMode;
        // Set antialiasing
        QualitySettings.antiAliasing = antiAliasing;

        // Place the camera on the render position.
        go.transform.position = renderPosition.position;
        go.transform.rotation = Quaternion.identity;

        // Bake the cubemap
        go.camera.RenderToCubemap(cubemap);

        // Rendering individual images
        if(createIndividualImages)
        {
            if (!Directory.Exists(imageDirectory))
            {
                Directory.CreateDirectory(imageDirectory);
            }

            RenderIndividualCubemapImages(go);
        }
```

```
        // Destroy the camera after rendering.
        DestroyImmediate(go);
    }

    void RenderIndividualCubemapImages(GameObject go)
    {
        go.camera.backgroundColor = Color.black;
        go.camera.clearFlags = CameraClearFlags.Skybox;
        go.camera.fieldOfView = 90;
        go.camera.aspect = 1.0f;

        go.transform.rotation = Quaternion.identity;

        //Render individual images
        for (int camOrientation = 0; camOrientation < eulerAngles.Length ; camOrientation++)
        {
            string imageName = Path.Combine(imageDirectory, cubemap.name + "_"
                    + cubemapImage[camOrientation] + ".png");
            go.camera.transform.eulerAngles = eulerAngles[camOrientation];
            RenderTexture renderTex = new RenderTexture(cubemap.height,
                    cubemap.height, cameraDepth);
            go.camera.targetTexture = renderTex;
            go.camera.Render();
            RenderTexture.active = renderTex;

            Texture2D img = new Texture2D(cubemap.height, cubemap.height,
                    TextureFormat.RGB24, false);
            img.ReadPixels(new Rect(0, 0, cubemap.height, cubemap.height), 0, 0);

            RenderTexture.active = null;
            GameObject.DestroyImmediate(renderTex);

            byte[] imgBytes = img.EncodeToPNG();
            File.WriteAllBytes(imageName, imgBytes);

            AssetDatabase.ImportAsset(imageName, ImportAssetOptions.ForceUpdate);
        }

        AssetDatabase.Refresh();
    }

    [MenuItem("GameObject/Bake Cubemap")]
    static void RenderCubemap ()
    {
        ScriptableWizard.DisplayWizard("Bake CubeMap", typeof(BakeStaticCubemap),"Bake!");
    }
}
```

6.3 Dynamic soft shadows based on local cubemaps

This technique uses a local cubemap to hold a texture that represents transparency of the static the environment. This is a very efficient technique that generates high-quality soft shadows.

This section contains the following subsections:

6.3.1 About dynamic soft shadows based on local cubemaps

In your scene there are moving objects and static environments such as rooms. By using this technique, you are not required to render static geometry to a shadow map every frame. This enables you to use a texture to represent the shadows.

Cubemaps can be a good approximation of many kinds of a static local environment including irregular shapes such as the cave in the Ice Cave demo. The alpha channel can also represent the amount of light entering the room.

The objects that move are typically everything except the room. Objects such as:

- The sun.
- The camera.
- Dynamic objects.

With the whole room represented by a cube texture you can access arbitrary texels of the environment within a fragment shader. For example, this means the sun can be in any arbitrary position and you can calculate the amount of light reaching a fragment based on the value fetched from the cubemap.

The alpha channel or transparency, represents the amount of light entering the local environment. In your scene, attach the cubemap texture to the fragment shaders that render the static and dynamic objects that you want to add shadows to.

6.3.2 Generating shadow cubemaps

Start with a local environment to which you want to apply shadows from light sources outside of the environment. For example, a room, a cave, or a cage.

This technique is similar to reflections based on a local cubemap. For more information see *6.2 Implementing reflections with a local cubemap* on page 6-84.

Create the shadow cubemap in the same manner as the reflection cubemap but you must also add an alpha channel. The alpha channel or transparency, represents the amount of light entering the local environment.

Work out the position from where to render the six faces of the cubemap. In most cases this is in the center of the bounding box of the local environment. You require this position for the generation of the cubemap. You must also pass this position to the shaders to calculate a local correction vector to fetch the right texel from the cubemap.

When you have decided where to position the center of the cubemap, you can render all the faces to the cubemap texture and record the transparency or alpha of the local environment. The more transparent an area is, the more light that comes into the environment. If there is no geometry it is fully transparent. If required, you can use the RGB channels to store the color of the environment for colored shadows such as stained glass, reflections, or refractions.

6.3.3 Rendering shadows

Build a vector P_iL in world space from a vertex or fragment, to the light or lights, and fetch the cubemap shadow using this vector.

Before fetching each texel, you must apply local correction to the P_iL vector. ARM recommends you make the local correction in the fragment shader to obtain more accurate shadows.

To compute the local correction, you must calculate an intersection point of the fragment-to-light vector with the bounding box of the environment. Use this intersection point to build another vector from the cubemap origin position `C` to the intersection point `P`. This gives you the final vector `CP` that you use to fetch the texel.

You require the following input parameters to calculate the local correction:

- `_EnviCubeMapPos` The cubemap origin position.
- `_BBoxMax` The maximum point of the bounding box of the environment.
- `_BBoxMin` The minimum point of the bounding box of the environment.
- `Pi` The fragment position in world space.
- `PiL` The normalized fragment-to-light vector in world space.

Compute the output value `CP`. This is the corrected fragment-to-light vector that you use to fetch a texel from the shadow cubemap.

The following figure shows the local correction of the fragment-to-light vector.

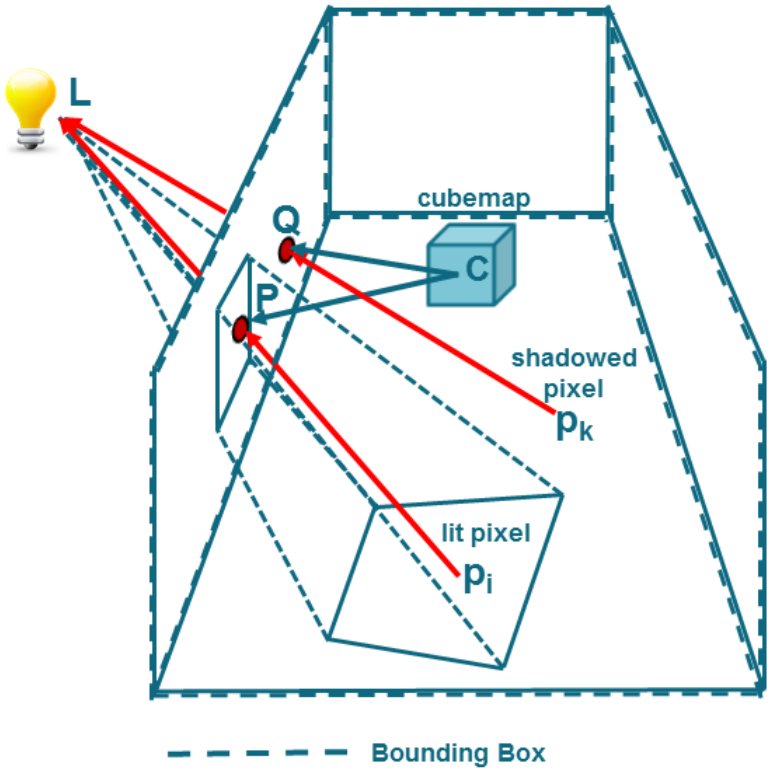

Figure 6-27 Local correction of the fragment-to-light vector

The following example code shows how to calculate the correct `CP` vector:

```
// Working in World Coordinate System.
vec3 intersectMaxPointPlanes = (_BBoxMax - Pi) / PiL;
vec3 intersectMinPointPlanes = (_BBoxMin - Pi) / PiL;

// Looking only for intersections in the forward direction of the ray.
vec3 largestRayParams = max(intersectMaxPointPlanes, intersectMinPointPlanes);

// Smallest value of the ray parameters gives us the intersection.
```

```
float dist = min(min(largestRayParams.x, largestRayParams.y), largestRayParams.z);

// Find the position of the intersection point.
vec3 intersectPositionWS = Pi + PiL * dist;

// Get the local corrected vector.
CP = intersectPositionWS - _EnviCubeMapPos;
```

Use the `CP` vector to fetch a texel from the cubemap. The alpha channel of the texel provides information about how much light or shadow you must apply to a fragment:

```
float shadow = texCUBE(cubemap, CP).a;
```

The following figure shows a chess room with hard shadows:

Figure 6-28 Chess room with hard shadows

This technique generates working shadows in your scene, but you can improve the quality of the shadows with two more steps:

- Back faces in shadow
- Smoothness

Back faces in shadow

The cubemap shadow technique does not use depth information to apply shadows. This means that some faces are incorrectly lit when they are meant to be in shadow.

The problem only occurs when a surface is facing in the opposite direction to the light. To fix this problem, check the angle between the normal vector and the fragment-to-light vector, P_iL. If the angle, in degrees, is outside of the range -90 to 90, the surface is in shadow.

The following code snippet does this check:

```
if (dot(PiL,N) < 0)
shadow = 0.0;
```

The previous code causes each triangle into a hard switch from light to shade. For a smooth transition use the following formula:

```
shadow *= max(dot(PiL, N), 0.0);
```

Where:

- `shadow` is the alpha value fetched from the shadow cubemap.
- P_iL is the normalized fragment-to-light vector in world space.
- `N` is the normal vector of the surface in world space.

The following figure shows a chess room with back faces in shadow:

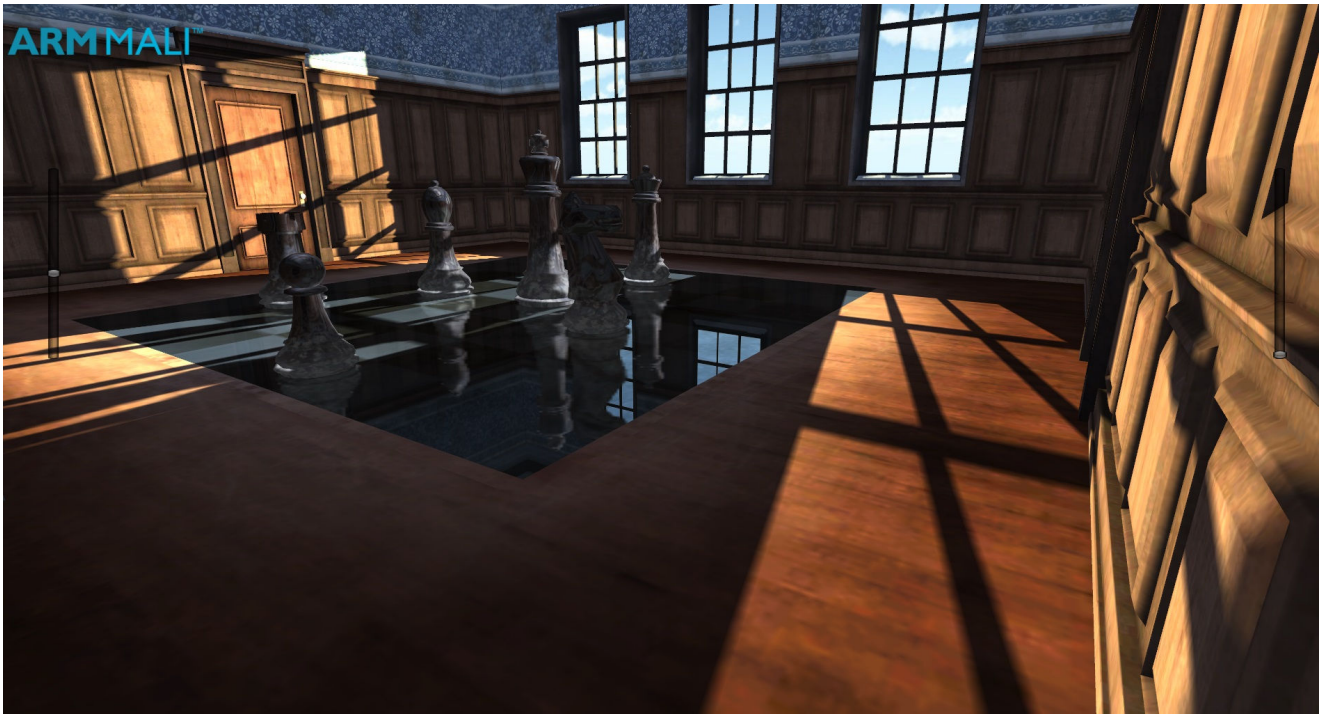

Figure 6-29 Chess room with back faces in shadow

Smoothness

This shadow technique can provide realistic soft shadows in your scene.

1. Generate mipmaps and set trilinear filtering for the cubemap texture.
2. Measure the length of a fragment-to-intersection-point vector.
3. Multiply the length by a coefficient.

The coefficient is a normalizer of a maximum distance in your environment to the number of mipmap levels. You can calculate this automatically against bounding volume and mipmap levels. You must customize the coefficient to your scene. This enables you to tweak the settings to suit the environment, improving visual quality. For example, the coefficient used in the Ice Cave project is `0.08`.

You can reuse the results of calculations that you did for the local correction. Reuse `dist` from the code snippet for local correction as the length of the segment from the fragment position to the intersection point of the fragment-to-light vector with the bounding box:

```
float texLod = dist;
```

Multiply `texLod` by the distance coefficient:

```
texLod *= distanceCoefficient;
```

To implement softness, fetch the correct mipmap level of the texture using the Cg function `texCUBElod()` or the GLSL function `textureLod()`.

Construct a `vec4` where `XYZ` represents a direction vector and the `W` component represents the LOD.

```
CP.w = texLod;
shadow = texCUBElod(cubemap, CP).a;
```

This technique provides high-quality, smooth shadows for your scene.

The following figure shows a chess room with smooth shadows:

Figure 6-30 Smooth shadows

6.3.4 Combining cubemap shadows with a shadow map

To get shadows complete with dynamic content, you must combine cubemap shadows with a traditional shadow map technique. This is more work but it is still worth it because you are only required to render dynamic objects to the shadow map.

The following figure shows the chess room with smooth shadows only:

Figure 6-31 Smooth shadows

The following figure shows the chess room with smooth shadows combined with dynamic shadows:

Figure 6-32 Smooth shadows combined with dynamic shadows

6.3.5 Results of the cubemap shadow technique

In traditional techniques, rendering shadows can be quite expensive because it involves rendering the whole scene from the perspective of each shadow-casting light source. The cubemap shadow technique described here delivers improved performance because it is mostly pre-baked.

This technique is also independent of output-resolution. It produces the same visual quality at 1080p, 720p and other resolutions.

The softness filtration is calculated in hardware so the smoothness comes at almost no computational cost. The smoother the shadow, the more efficient the technique is. This is because the smaller mipmap levels result in less data than traditional shadow map techniques. Traditional techniques require a large kernel to make shadows smooth enough to be more visually appealing. This requires high memory bandwidth and this reduces performance.

The quality you get with the cubemap shadow technique is higher than you might expect. It provides realistic softness and stable shadows with no shimmering at the edges. Shimmering edges can be observed when using traditional shadow map techniques because of rasterization and aliasing effects. However, none of the anti-aliasing algorithms can fix this problem entirely.

The cubemap shadow technique does not have a shimmering problem. The edges are stable even if you use a much lower resolution than that used in the render target. You can use four times lower resolution than the output and there are neither artifacts nor unwanted shimmering. Using four times lower resolution also saves memory bandwidth and this improves performance.

This technique can be used on any device on the market that supports shaders such as those with OpenGL ES 2.0 or higher. If you already know where and when to use the reflections based on local cubemaps technique, then you can easily apply the shadow technique to your implementation.

Note

The technique cannot be used for everything in your scene. Dynamic objects, for instance, receive shadows from the cubemap but they cannot be pre-baked to the cubemap texture. For dynamic objects, use shadow maps for generating shadows, blended with the cubemap shadow technique.

The following figure shows the Ice Cave with shadows:

Figure 6-33 Ice Cave with shadows

The following figure shows the Ice Cave with smooth shadows:

Figure 6-34 Ice Cave with smooth shadows

The following figure shows the Ice Cave with smooth shadows:

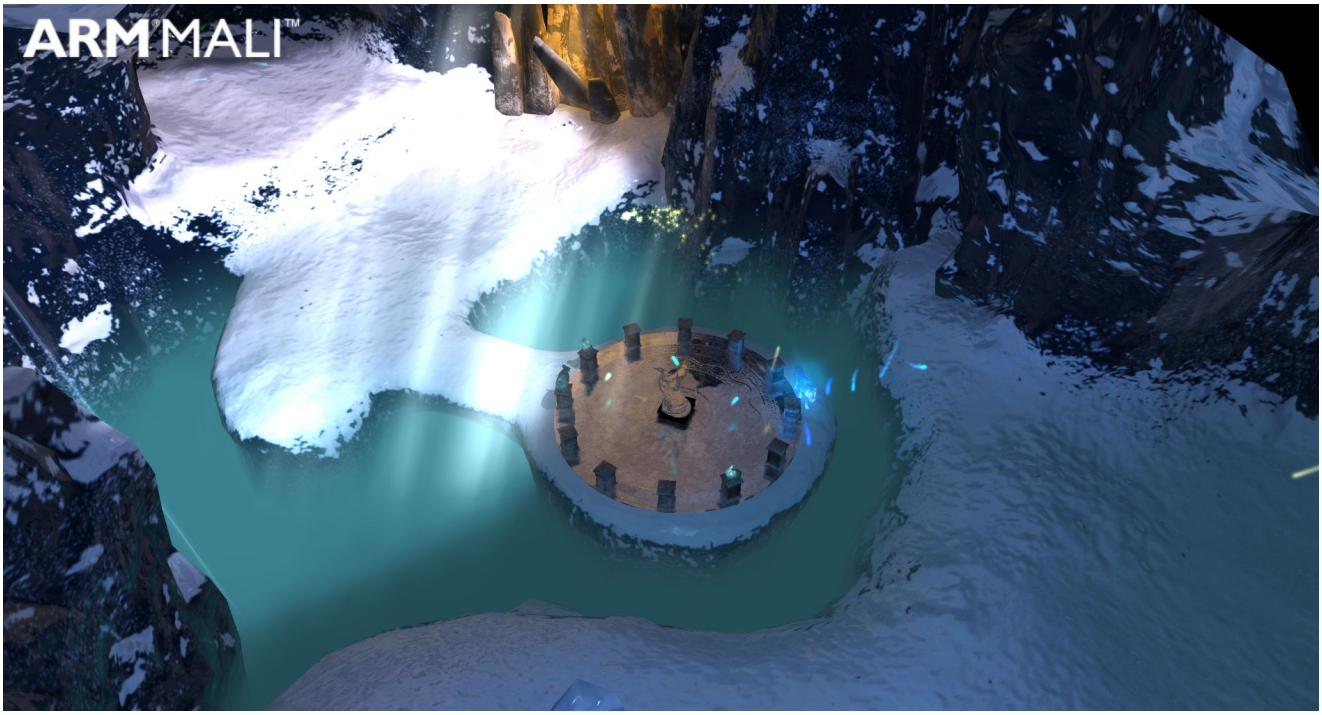

Figure 6-35 Ice Cave with smooth shadows

6.4 Refraction based on local cubemaps

You can use local cubemaps to implement high quality refractions. You can combine these with reflections at runtime.

This section contains the following subsections:

6.4.1 About refractions

Game developers are regularly looking for efficient methods to implementing visually impressive effects in their games. This is especially important when targeting mobile platforms because you must carefully balance resources to achieve maximum performance.

Refraction is the change in direction of a light wave because of a change in the medium it is passing through. If you want extra realism with semi-transparent geometry, refraction is an important effect to consider.

The refractive index determines how much light is bent, or refracted, when entering a material. Refraction is defined as the bending of light as it passes from one medium, with refrative index n_1 to another medium with recractive index n_2.

You use Snell's Law to calculate the relationship between the refractive indices and the sine of the incident and refracted angles.

The following figure shows Snell's Law and refraction:

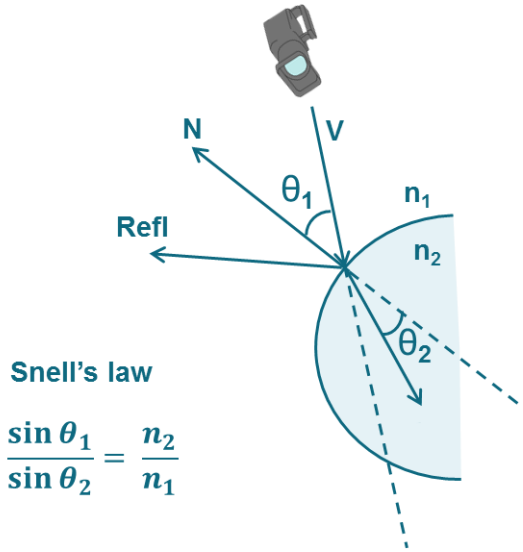

Figure 6-36 Refraction of light as it passes through one medium to another

6.4.2 Refraction implementations

Developers have tried to render refraction since they started to render reflections because these processes take place together in any semi-transparent surface. There are several techniques for rendering reflections but not many for refraction.

Existing methods for implementing refraction at runtime differ depending on the specific type of refraction. Most of the techniques render the scene behind the refractive object to a texture at runtime, and then apply a texture distortion in a second pass to achieve the refracted appearance. Depending on

the texture distortion, you can use this approach to render refraction effects such as water, heat haze, glass, and other effects.

Some of these techniques can achieve good results, but the texture distortion is not physically based so the results are not always correct. For example, if you render a texture from the point of view of the refraction camera, there might be areas that are not directly visible to the camera but are visible in a physically based refraction.

The main limitation of using render-to-texture methods is quality. When the camera is moving, pixel shimmering or pixel instability is often visible.

6.4.3 About refractions based on local cubemaps

Local cubemaps are an excellent technique for rendering reflections and developers have used static cubemaps to implement both reflections and refractions since they became available.

However, if you use static cubemaps to implement reflections or refractions in a local environment, the results are incorrect if you do not apply a local correction.

In the technique described here a local correction is applied to ensure correct results. This technique is highly optimized. It is especially useful for mobile devices where runtime resources are limited so must be carefully balanced.

6.4.4 Preparing the cubemap

You must prepare the cubemap to be used in the refraction implementation:

To prepare the cubemap, do the following:

1. Place a camera in the center of the refractive geometry.
2. Hide the refractive object and render the surrounding static environment to a cubemap in the six directions. You can use this cubemap for implementing both refraction and reflection.
3. Bake the environment surrounding the refractive object into a static cubemap.
4. Determine the direction of the refraction vector, and find where it intersects with the bounding box of the local environment.
5. Apply the local correction in the same way as *6.3 Dynamic soft shadows based on local cubemaps* on page 6-100.
6. Build a new vector from the position where the cubemap was generated, to the intersection point. Use this final vector to fetch the texel from the cubemap, to render what is behind the refractive object.

Instead of fetching the texel from the cubemap using the refracted vector R_{rf}, you find the point P where the refracted vector intersects the bounding box and build a new vector R'_{rf} from the center of the cubemap C to the intersection point P. Use this new vector to fetch the texture color from the cubemap.

```
float eta=n2/n1;
```

```
float3R_rf = refract(D,N,eta);
```

Find intersection point P

Find vector `R'_rf = CP;`

```
Float4 col = texCube(Cubemap, R'_rf);
```

The following figure shows a scene with a cubemap and the refraction vectors:

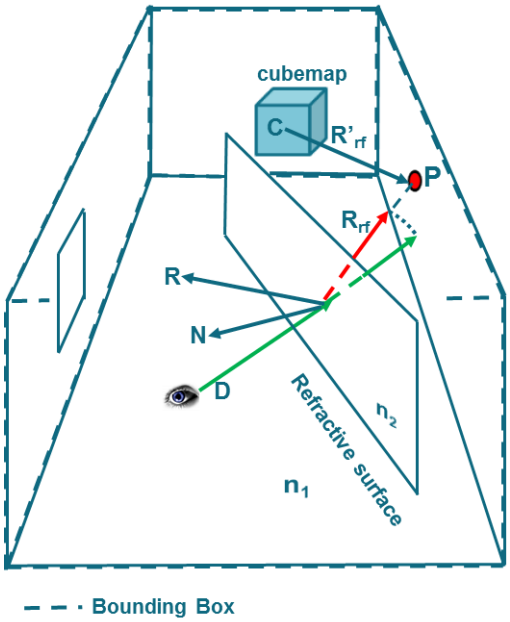

Figure 6-37 The local correction to refraction vector

The refraction produced by this technique is accurately physically based, because the direction of the refraction vector is calculated using Snell's Law.

There is also a built-in function that you can use in your shader to find the refraction vector `R` strictly according to the Snell's law:

```
R = refract( I, N, eta);
```

Where:

- `I` is the normalized view or incident vector.
- `N` is the normalized normal vector.
- `eta` is the ratio of indices of refractions n_1/n_2.

The following figure shows the flow of shaders that implement refraction based on a local cubemap:

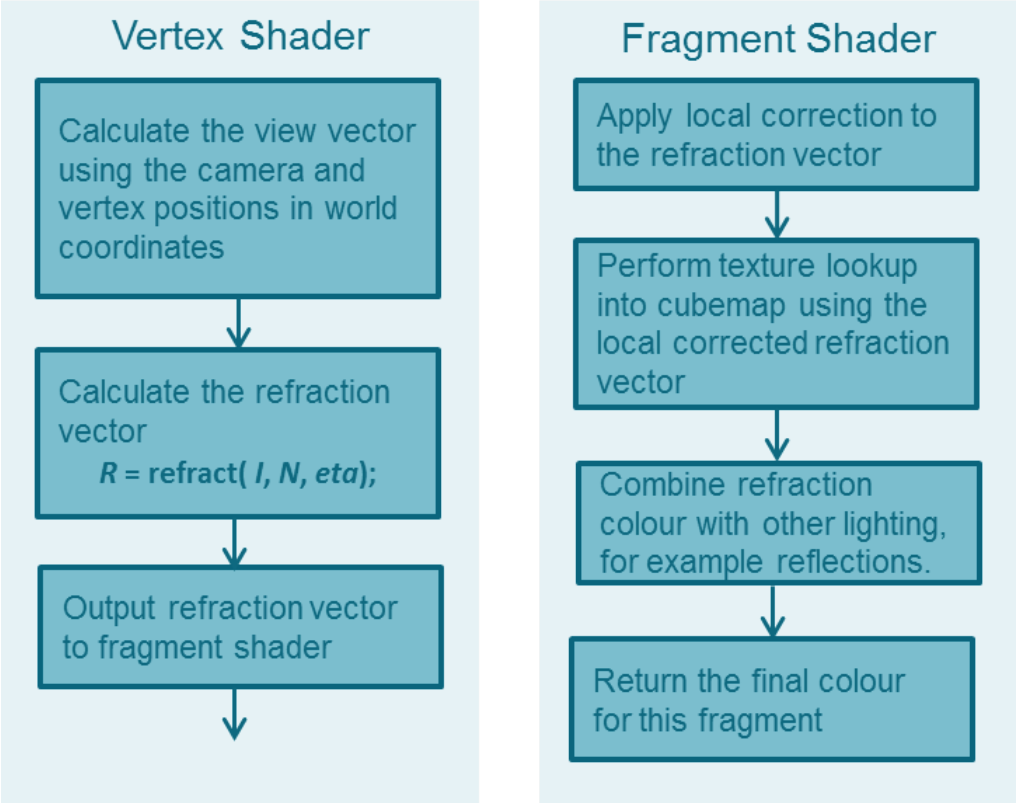

Figure 6-38 Shader implementations of refraction based on local cubemap

6.4.5 Shader implementation

When you fetch the texel corresponding to the locally-corrected refraction direction, you might want to combine the refraction color with other lighting. For example, reflections that take place simultaneously with refraction.

To combine the refraction color with other lighting, you must pass an additional view vector to the fragment shader and apply the local correction to it. Use the result to fetch the reflection color from the same cubemap.

The following code snippet shows how to combine reflection and refraction to produce the final output color:

```
// ----------------- Environment reflections ----------------
float3 newReflDirWS = LocalCorrect(input.reflDirWS, _BBoxMin, _BBoxMax, input.posWorld,
_EnviCubeMapPos);
float4 staticReflColor = texCUBE(_EnviCubeMap, newReflDirWS);
// ----------------- Environment refractions ----------------
float3 newRefractDirWS = LocalCorrect(RefractDirWS, _BBoxMin, _BBoxMax, input.posWorld,
_EnviCubeMapPos);
float4 staticRefractColor = texCUBE(_EnviCubeMap, newRefractDirWS);
// ----------------- Combined reflections and refractions ----------------
float4 combinedReflRefract = lerp(staticReflColor, staticRefractColor, _ReflAmount);

float4 finalColor = _AmbientColor + combinedReflRefract;
```

The coefficient `_ReflAmount` is passed as a uniform to the fragment shader. Use this coefficient to adjust the balance between reflection and refraction contributions. You can manually adjust `_ReflAmount` to achieve the visual effect you require.

You can find the implementation of the `LocalCorrect` function in the reflections blog at: *http://community.arm.com/groups/arm-mali-graphics/blog/2014/08/07/reflections-based-on-local-cubemaps*.

When the refractive geometry is a hollow object, refractions and reflections take place in both the front and back surfaces.

The following figure shows refraction on a glass bishop based on a local cubemap:

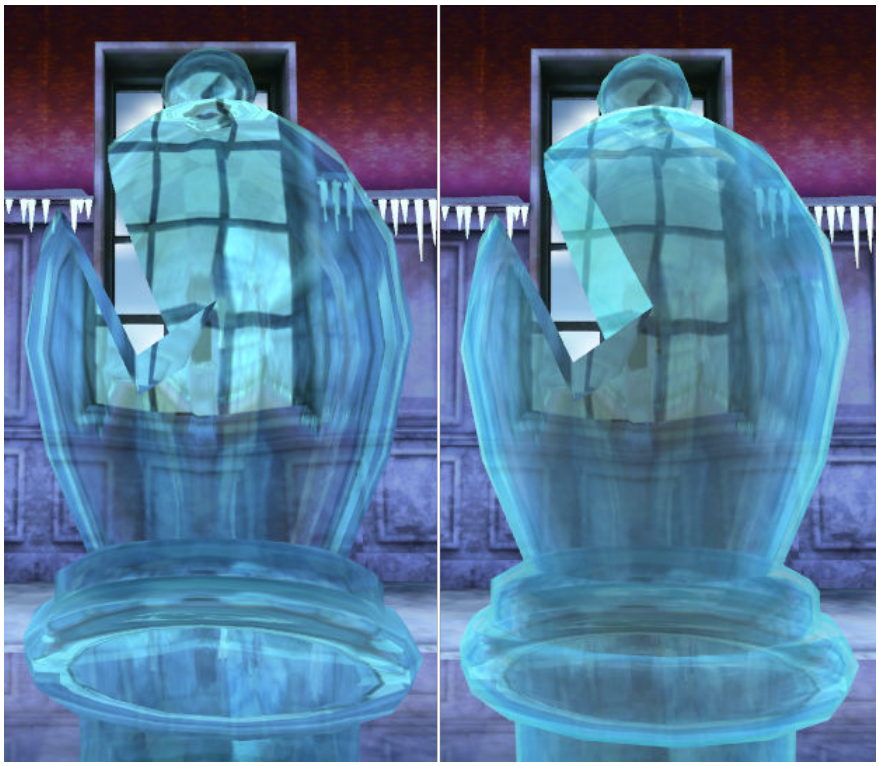

Figure 6-39 Refraction on a glass bishop based on a local cubemap

The image on the left shows the first pass that renders only back faces with local refraction and reflections.

The image on the right shows the second pass renders only front faces with local refraction and reflections and alpha blending with the first pass.

- In the first pass, render the semi-transparent object in the same manner that you render opaque geometry. Render the object last with front-face culling on, that is, only render the back faces. You do not want to occlude other objects so do not write to the depth buffer.

  The color of the back face is obtained by mixing the colors calculated from the reflection, refraction, and the diffuse color of the object itself.
- In the second pass, render the front faces with back face culling, do this last in the rendering queue. Ensure depth writing is off. Obtain the front-face color by mixing the refraction and reflection textures with the diffuse color. The refraction in the second pass adds more realism to the final rendering. You can skip this step if the refraction on the back faces is enough to highlight the effect.
- In the final pass you alpha-blend the resulting color with the first pass.

The following figure shows the result of implementing refractions based on a local cubemap, on a semi-transparent phoenix in the Ice Cave demo.

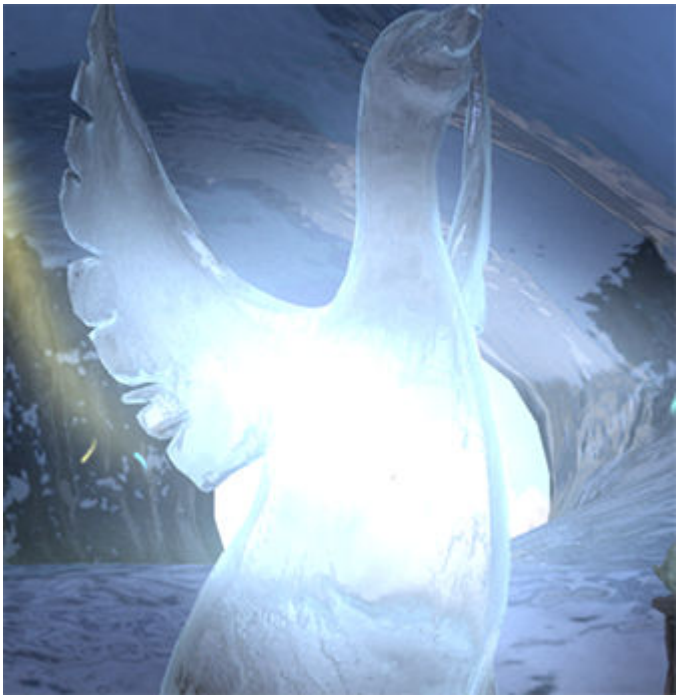

Figure 6-40 Semi-transparent phoenix refractions

The following figure shows a semi-transparent phoenix wing:

Figure 6-41 Semi-transparent phoenix wing

6.5 Specular effects in the Ice Cave demo

Specular effects in the Ice Cave demo are implemented using the Blinn technique. This is a very efficient technique that produces good results.

The following code shows how to implement specular effects with the Blinn technique:

```
// Returns intensity of a specular effect without taking into account shadows
float SpecularBlinn(float3 vert2Light, float3 viewDir, float3 normalVec, float4 power)
{
    float3 floatDir = normalize(vert2Light - viewDir);
    float specAngle = max(dot(floatDir, normalVec), 0.0);
    return pow(specAngle, power);
}
```

A downside of the Blinn technique is that it can produce incorrect results in certain circumstances. For example, specular effects can appear in regions that are in shadow.

The following figure shows an example of a shadowed region with no specular effects:

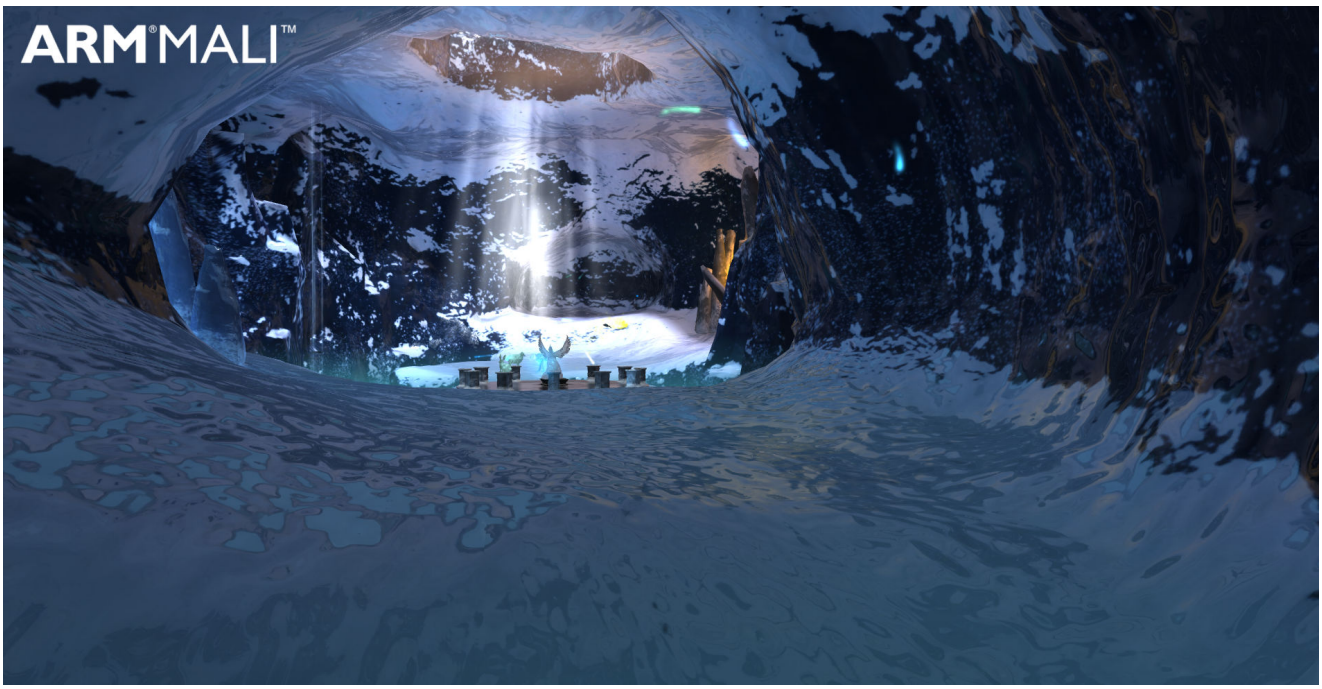

Figure 6-42 Shadowed region

The following figure shows an example of specular effects in a shadowed region. These are incorrect:

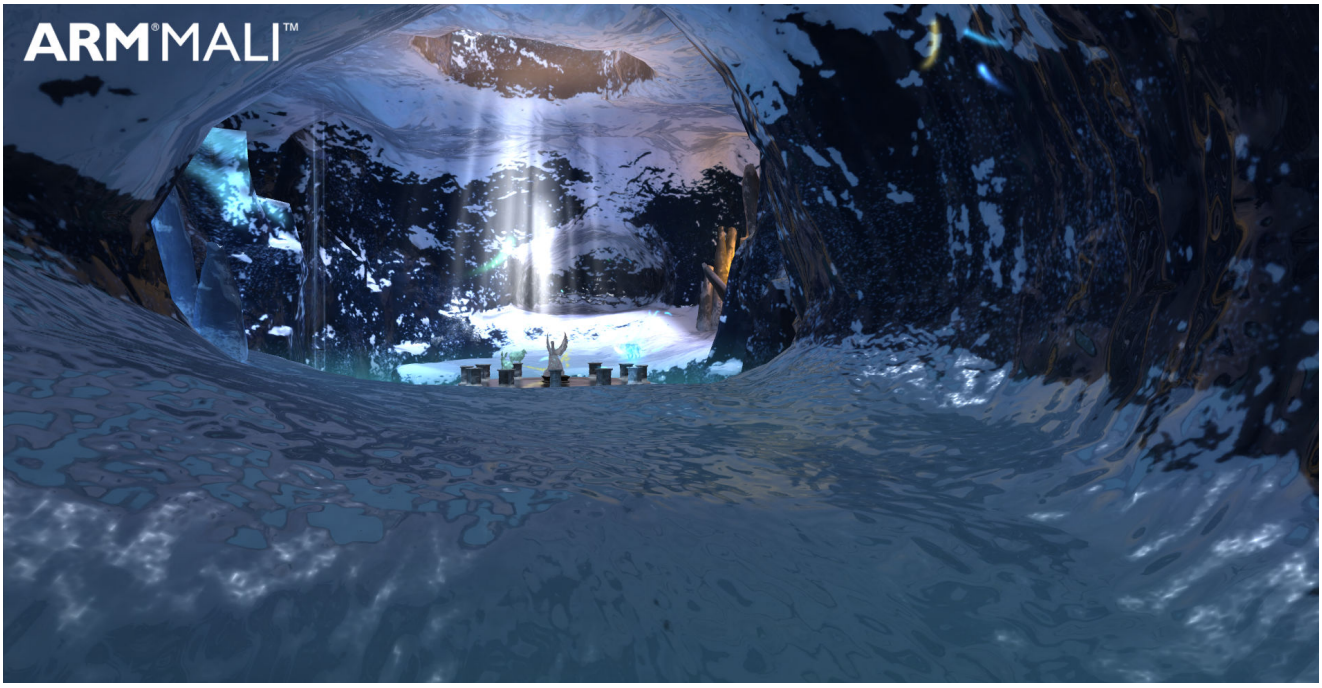

Figure 6-43 Shadowed region with incorrect specular effects

The following figure shows an example of specular effects in a lit region. This is correct:

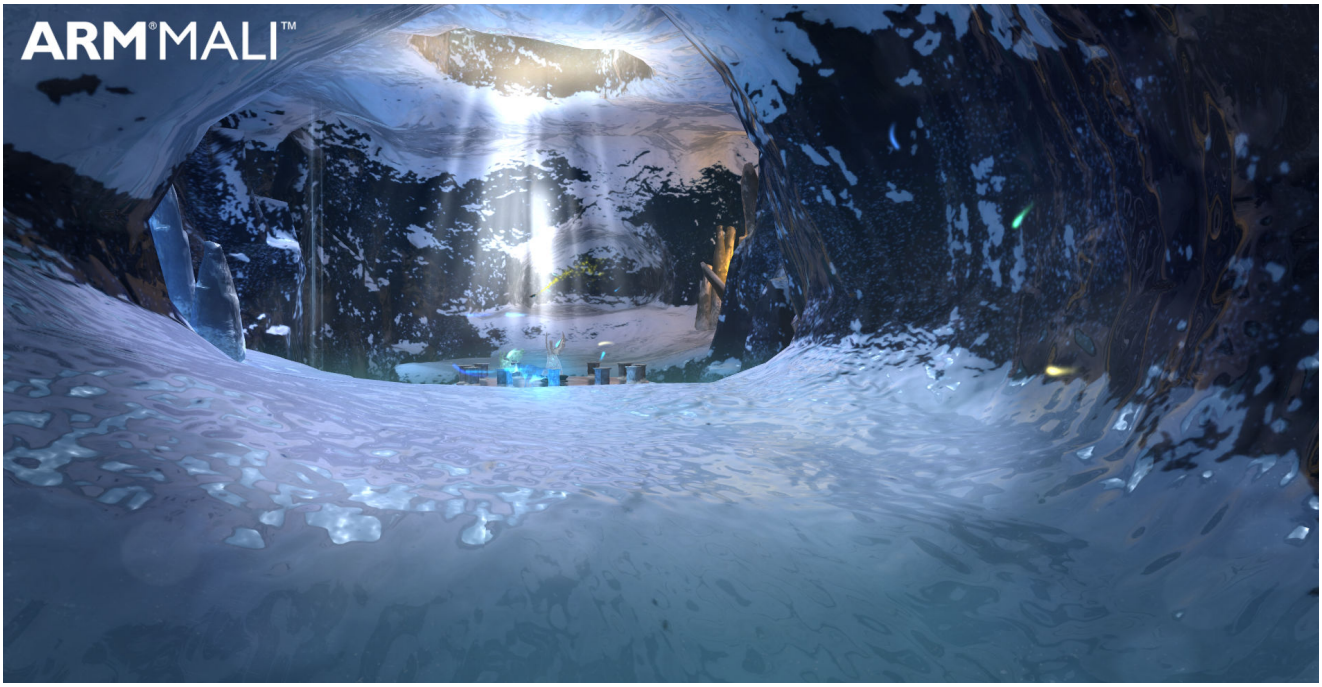

Figure 6-44 Lit region with correct specular effects

The shadows should make the specular effects intensity either stronger or weaker, depending on the light reaching the specular surface. All the information required to correct the intensity of the specular effects is already in the Ice Cave demo, so fixing it is relatively simple. The environment cubemap texture that is used for reflection and shadow effects contains two types of information. The RGB channels contain environment colors that are used for reflections. The alpha channel contains opacity and this is used for shadows. You can use the alpha channel to determine the specular intensity because the alpha channel

represents holes in the cave that let light in to the environment. The alpha channel is therefore used to ensure the specular effect is applied only to surfaces that are lit.

To do this, calculate a corrected reflection vector in a fragment shader to make the reflection effect. For more information about creating the corrected reflection vector see *6.2 Implementing reflections with a local cubemap* on page 6-84. Use this vector to fetch the RGBA texel from a cubemap texture:

```
// Locally corrected static reflections
 const half4 reflColor = SampleCubemapWithLocalCorrection(
        ReflDirectionWS,
        _ReflBBoxMinWorld,
        _ReflBBoxMaxWorld,
        input.vertexInWorld,
        _ReflCubePosWorld,
        _ReflCube);
```

For more information about what the `SampleCubemapWithLocalCorrection()` function, see *6.2 Implementing reflections with a local cubemap* on page 6-84.

`reflColor` is in RGBA format where the RGB components contain color data for reflections and the alpha channel contains the intensity of the specular effect. In the Ice Cave demo, the alpha channel is multiplied by the specular color, that is calculated with the Blinn technique:

```
half3 specular = _SpecularColor.rgb *
        SpecularBlinn(
            input.vertexToLight01InWorld,
            viewDirInWorld,
            normalInWorld,
            _SpecularPower) *
        reflColor.a;
```

The `specular` value represents the final specular color. You can add this to your lighting model.