ARM® Guide to Unity: Enhancing Your Mobile Games

Version 1.0

ARM 100140_0100_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

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

Chapter 3 Profiling Unity Applications
This chapter describes profiling your Unity application.

Chapter 4 Optimizations for Applications in Unity
This chapter lists a number of optimizations for your Unity application.

Chapter 5 Advanced Techniques in Unity
This chapter lists a number of advanced techniques you can use in your Unity application.

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 *Infocenter* 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:

Mali Developer Center.

Other publications

Relevant documents published by third parties:

OpenGL ES 1.1 Specification at *Khronos*.
OpenGL ES 2.0 Specification at *Khronos*.
OpenGL ES 3.0 Specification at *Khronos*.
Unity Scripting Reference at *Unity*.
GPU Gems: Programming Techniques, Tips, and Tricks for Real-time Graphics by Randima Fernando (Series 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.
- The number ARM 100140_0100_00_en.
- 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 3.0, and OpenCL 1.1.

The Mali Midgard family includes the following Mali GPUs:

- Mali-T604.
- Mali-T620.
- Mali-T624.
- Mali-T628.
- Mali-T720.
- Mali-T760.

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 have limited computing power and memory bandwidth to save power.

1.4 About the Student Project

The student project is a demonstration game created by ARM using Unity.

This book describes the graphical techniques that are used in the game, and solutions to problems that were encountered during the development process of the student project.

Chapter 2
About 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:

If you have a game that does not perform as well as you would like, use the Unity profiler to take measurements

Use the Unity profiler to analyze the measurements so you can isolate and identify the source of the performance problem.

You might, for example, find your game is rendering too many vertices and you can identify this with the profiler.

Reduce the number of vertices in your code and 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 Quality settings in Unity

It is useful to know the quality settings in Unity to ensure you pick the right 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.

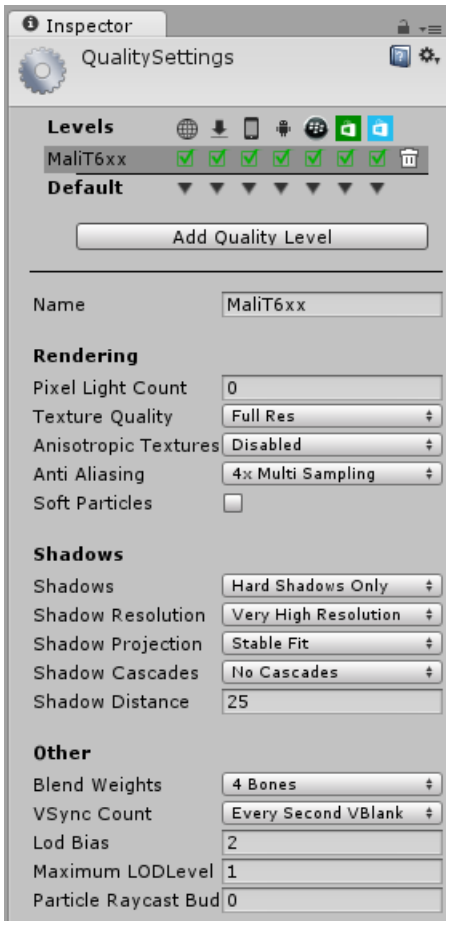

Figure 2-1 Quality settings

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 performing a complex graphical effect. You can do less complex versions of graphical effects such as shadows and lighting for a relatively small impact on the graphical quality. The simpler effects can reduce the load on the GPU quite significantly, providing an higher frame rate.

The default settings in Unity 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 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.

If textures are causing performance problems, try using Mip mapping. Mip mapping reduces compute and bandwidth requirements without impacting image quality.

——— **Note** ———

For example, in the student project **Texture quality** is set to full resolution.

———————————

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 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 it 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 if soft particles are important enough to your game to use them.

Anisotropic Textures

Anisotropic Textures is a technique that removes distortion from triangle textures drawn at a high gradient. This improves the image quality but it is an expensive technique. It might be worth it depending on the style of your game.

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.

Chapter 3
Profiling Unity Applications

This chapter describes profiling your Unity application.
It contains the following sections:

3.1 About profiling Unity

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.
- Mali Graphics Debugger.
- DS-5 Streamline.

3.2 Profiling Unity with the Unity profiler

The Unity profiler is a tool included in Unity Pro. It provides detailed per-frame performance data in a series of profiler 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.

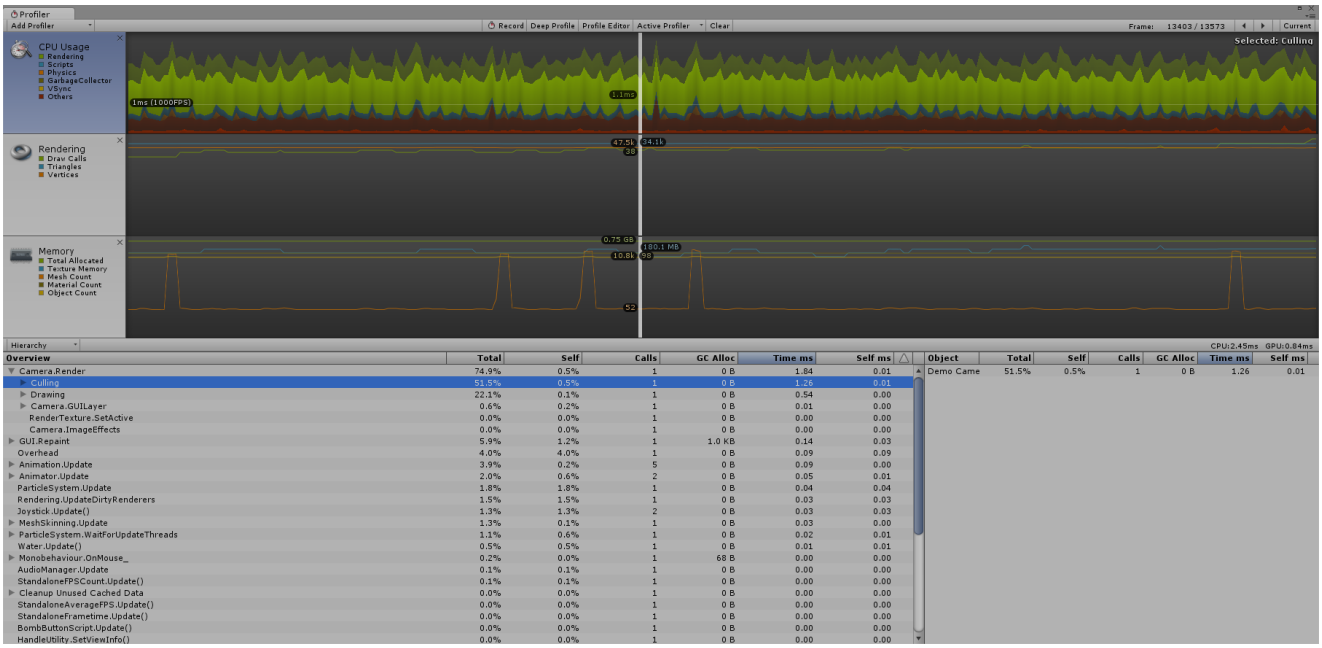

Figure 3-1 Unity profiler

The Unity profiler provides the following functions:

CPU Usage

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** ———

The values are averaged in a multi-processor system.

———————————

Rendering

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

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 makes the panel display 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 and attach a custom label to it and this region appears in the profiler hierarchy as a separate entry. Doing this enables you can get information about a specific piece of code without the overhead of computation and memory of the `Deep Profile` option.

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

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

Chapter 4
Optimizations for Applications in Unity

This chapter lists a number of optimizations for your Unity application. It contains the following sections:

4.1 Application processor optimizations in Unity

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.

```
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 had 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));
```

——— **Note** ———

The student project switched from the `Monobehaviour.Invoke()` method to using coroutines to have 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.

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

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

——— **Note** ———

This technique is used in the student project to spawn enemies on an irregular interval. An infinite loop inside the coroutine spawns an enemy and generates a random number that it passes 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.

```
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

The student project uses a Tags class with public constant strings to assist adding new tags in a consistent and scalable manner.

Reduce 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.

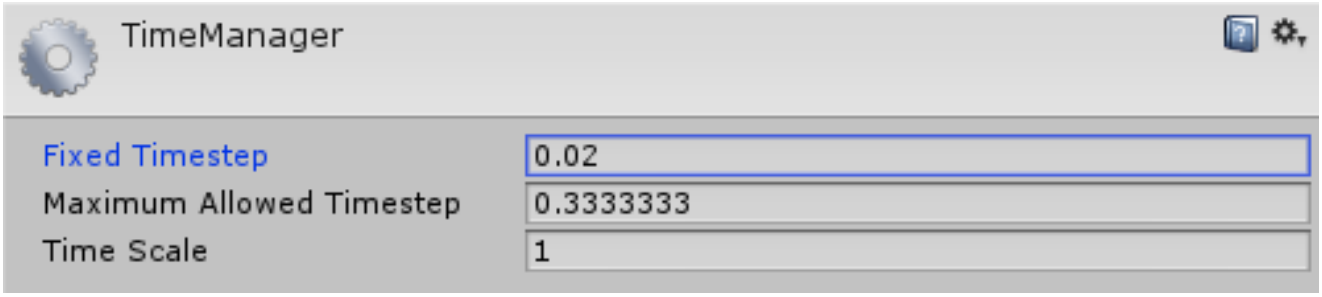

Figure 4-1 Fixed timestep settings

Remove empty callbacks

If your code includes empty definitions for functions like `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.

```
// 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.

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

A better technique is to call `GameObject.Find()` on start up 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

The student project uses a dedicated class called `LocatorManager` that performs all the object retrievals at once when the scene finishes loading. The other classes in The student project 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();
```

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
}
```

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 at once 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 found.

——— Note ———

The student project uses 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.

——— Note ———

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);
}
```

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

——— Note ———

In Unity 5 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 a GUI manager to group OnGUI() calls

One technique to optimize the performance of the Unity GUI system is to centralize all the `MonoBehaviour.OnGUI()` calls. You can have multiple scripts dealing with GUI components but instead of each one having their own `OnGUI()` method, they call a GUI manager that handles all the GUI components.

The manager groups all the GUI interactions into a single GUI callback. Using a single class to call the `OnGUI()` method also frees the remaining GUI components classes from having to inherit from the `MonoBehaviour` class.

Consider using an observer pattern to implement this.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	84.1%	84.1%	1	0 B	12.41	12.41
▼ GUI.Repaint	13.6%	2.1%	1	4.4 KB	2.01	0.31
▶ GUIUtility.BeginGUI()	4.5%	0.4%	20	2.8 KB	0.66	0.06
▶ GUIUtility.EndGUI()	2.4%	0.5%	20	480 B	0.36	0.08
▶ Gui00Test.OnGUI()	0.4%	0.0%	2	116 B	0.07	0.00
▶ Gui04Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Gui09Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Gui05Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Gui01Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Gui03Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Gui07Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Gui08Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Gui02Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Gui06Test.OnGUI()	0.4%	0.0%	2	116 B	0.06	0.00
▶ Event.Internal_MakeMasterEventCurrent()	0.0%	0.0%	1	0 B	0.00	0.00
Undo.ClearSnapshotTarget()	0.0%	0.0%	20	0 B	0.00	0.00

Figure 4-4 Multiple-on GUI

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	81.7%	81.7%	1	0 B	12.10	12.10
ProcessRemoteInput	11.4%	11.4%	1	0 B	1.69	1.69
▼ GUI.Repaint	5.3%	0.3%	1	1.3 KB	0.79	0.05
▶ GuiSingleTest.OnGUI()	3.6%	0.3%	2	1.0 KB	0.54	0.05
▶ GUIUtility.BeginGUI()	0.9%	0.0%	2	288 B	0.14	0.00
▶ GUIUtility.EndGUI()	0.3%	0.0%	2	48 B	0.04	0.01
▶ Event.Internal_MakeMasterEventCurrent()	0.0%	0.0%	1	0 B	0.00	0.00
Undo.ClearSnapshotTarget()	0.0%	0.0%	2	0 B	0.00	0.00

Figure 4-5 Single-on GUI

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 like 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 a list of empty GameObjects to act as the collider planes.

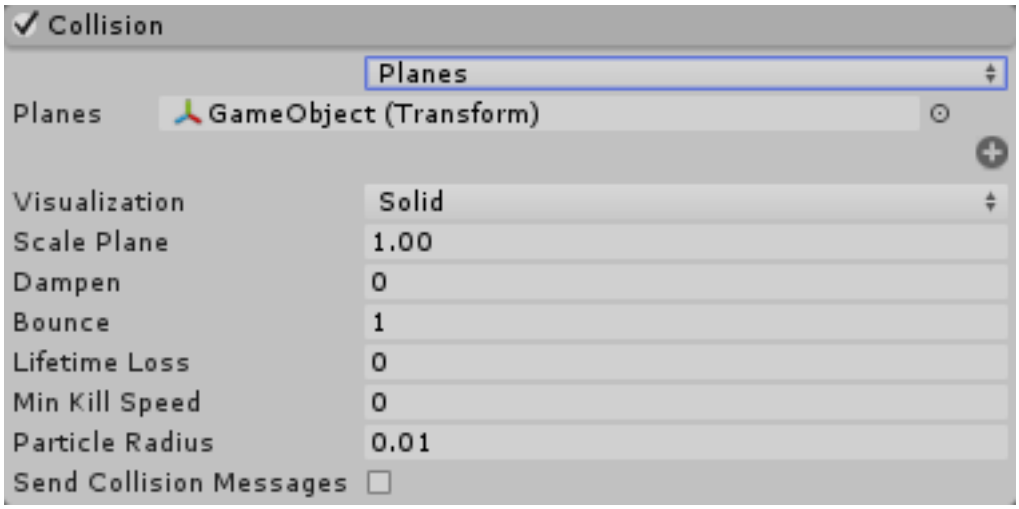

Figure 4-6 Collision settings

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 like 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 in Unity

This section contains the following subsections:

4.2.1 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.

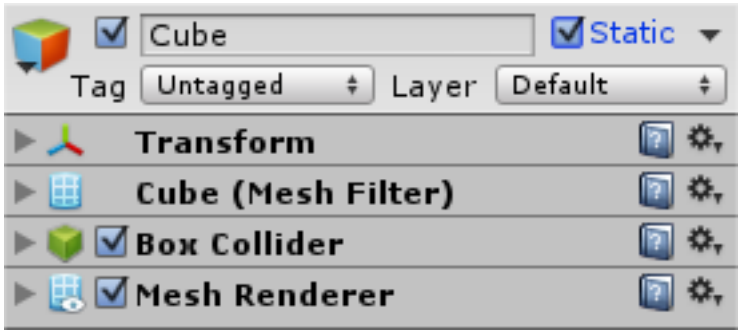

Figure 4-7 Static batching settings

Note

Static batching is only available in Unity Pro.

Use 4x MSAA

ARM Mali GPUs can do 4x multisample anti-aliasing with very low computational overhead. You can enable 4x MSAA in the Unity `Quality Settings`.

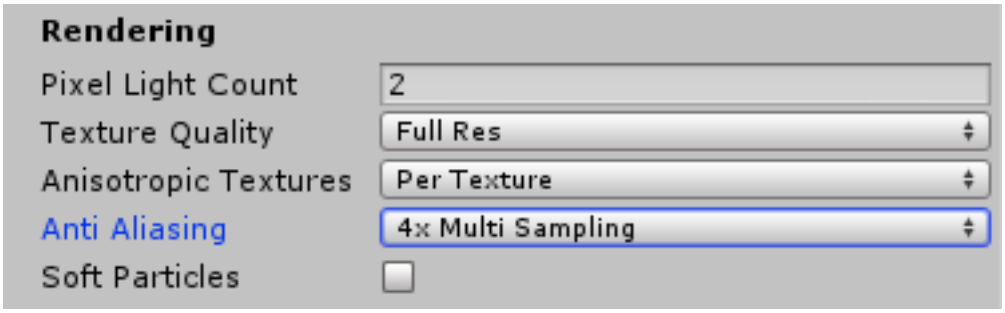

Figure 4-8 MSAA settings

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.

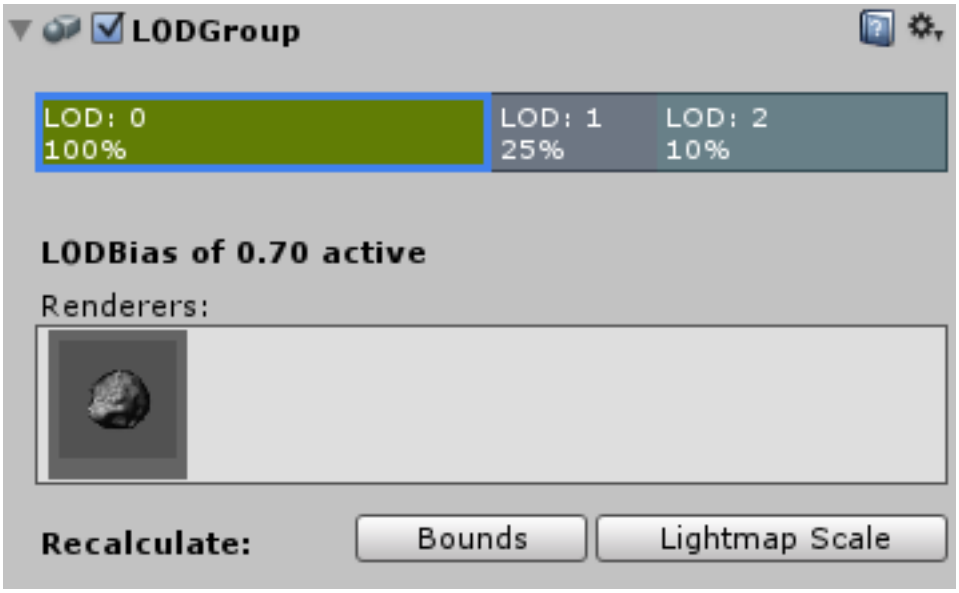

Figure 4-9 LOD group settings

Note

LOD is only available on Unity Pro.

Avoid mathematical functions in custom shaders

When you are writing custom shaders try to avoid the use of expensive built-in mathematical functions such as:

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

4.2.2 Lightmaps and light probes in Unity

Lighting calculations are computationally expensive if they are performed at runtime. One of the main techniques to reduce the computation of lighting in your scene is called lightmapping. The calculations required to light up objects are made before hand and are baked into a texture called a lightmap. You lose the flexibility of a fully dynamically lit environment, but pre-computed lighting can provide very high quality images without impacting performance.

Note

The light from the torches in the student project do not change, so it is pre-baked into a texture. This texture is combined with the textures of the surrounding models to create the impression of a lit object.

Figure 4-10 Lightmapped torch

Figure 4-11 Lightmapped torches

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**.
- A light within range of the model.

Open up the **lightmapping** tab inside the Unity editor. There are 3 buttons:

- **Object**.
- **Bake**.
- **Maps**.

Figure 4-12 Lightmap options

Object

Clicking the **Object** button enables you to change settings related to light mapping on the object you have selected in the hierarchy. This enables you to tweak the objects settings that impact the lightmapping process. Selecting a light enables you to change a number of options:

- **Baked Only** enables the light at Baking time and disables it at runtime.

  Setting the majority of lights to **Baked Only** ensures the number of calculations at runtime is relatively low.
- **Realtime Only** disables the light at Baking time and enables it at runtime.
- **Auto** enables the light at baking time and for objects close to the camera at runtime.

Bake

The **Bake** option contains most options that affect the visual quality of your lightmap. The settings you use depend on your scene.

The following describes the settings:

- **Mode** is the lightapping mode, there are a number of settings:
  - **Directional Lightmaps** preserve specular and normal data. These are important for creating a realistic scene.
  - Dual lightmaps.
  - Single lightmaps.
- **Quality** has two options that both have preset values. It is more likely you shall want the setting for your game to be more specific.
- **Bounce Boost** and **Intensity** control the bouncing of light around the scene. Experiment with these settings to get the best results. **Final Gather Rays** are the totally number of rays that can bounce to one pixel or texel in the world. The higher the number the more realistic the pixel but this increases the baking time. **Contrast Threshold** and **Interpolation** both create a smoother light map but this can smooth out detail. For **Ambient occlusion** the higher the number the more obvious the occlusion. For more information about this option see the Unity website.
- **Lock Atlas** prevents new indices being generated for your models within the lightmap. You can use this if you are happy with your current map indices or if you want to create your own Atlas.
- **Resolution** is a multiplier to all models UVs labeled as resolution in texels per world unit. Increasing the resolution value increases the number of texels dedicated for all objects sent to the unity lightmapper. Some large objects might not require a high resolution lightmap so experiment to see if you can use a lower resolution one.
- **Padding** is the space in texels in between each object within the lightmap. You can increase this if you see artifacts such as bleeding. After this is done bake your scene, after the lightmapper has finished doing all the calculations click the Maps button, and the new lightmaps are displayed.
- **Lightmaps** switches lightmaps on or off.
- **Shadow Distance** is the distance from the camera where shadows are rendered, in world units.
- **Show Resolution** is the resolution of model UVs within the lightmaps.
- **Show Probes** toggles the visibility of light probes.
- **Show Cells** toggles the visibility of light probe cells.

Selecting a renderer provides you with a number of settings and enables you to set whether its lightmap is static or not.

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.

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 needs to 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.

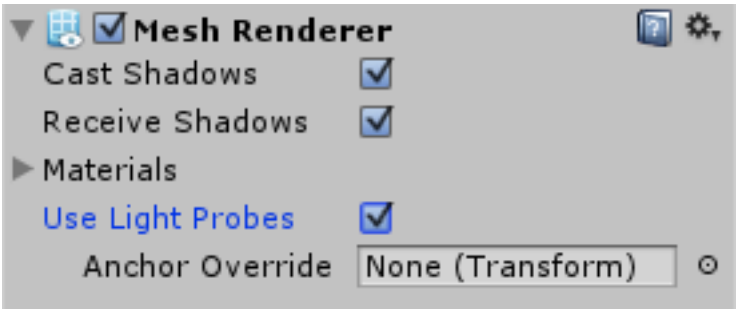

Figure 4-13 Light probes setting

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.

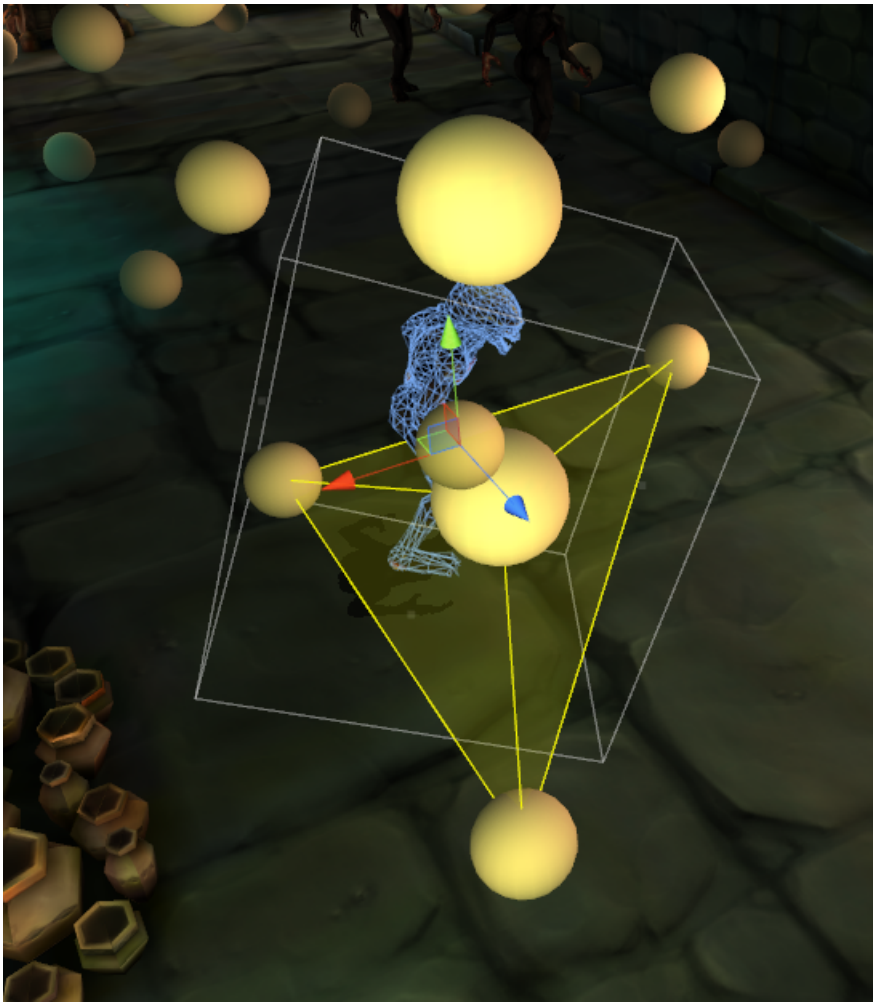

Figure 4-14 Light probes

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.

——— **Note** ———

Light probes are a Unity Pro option.

———————————

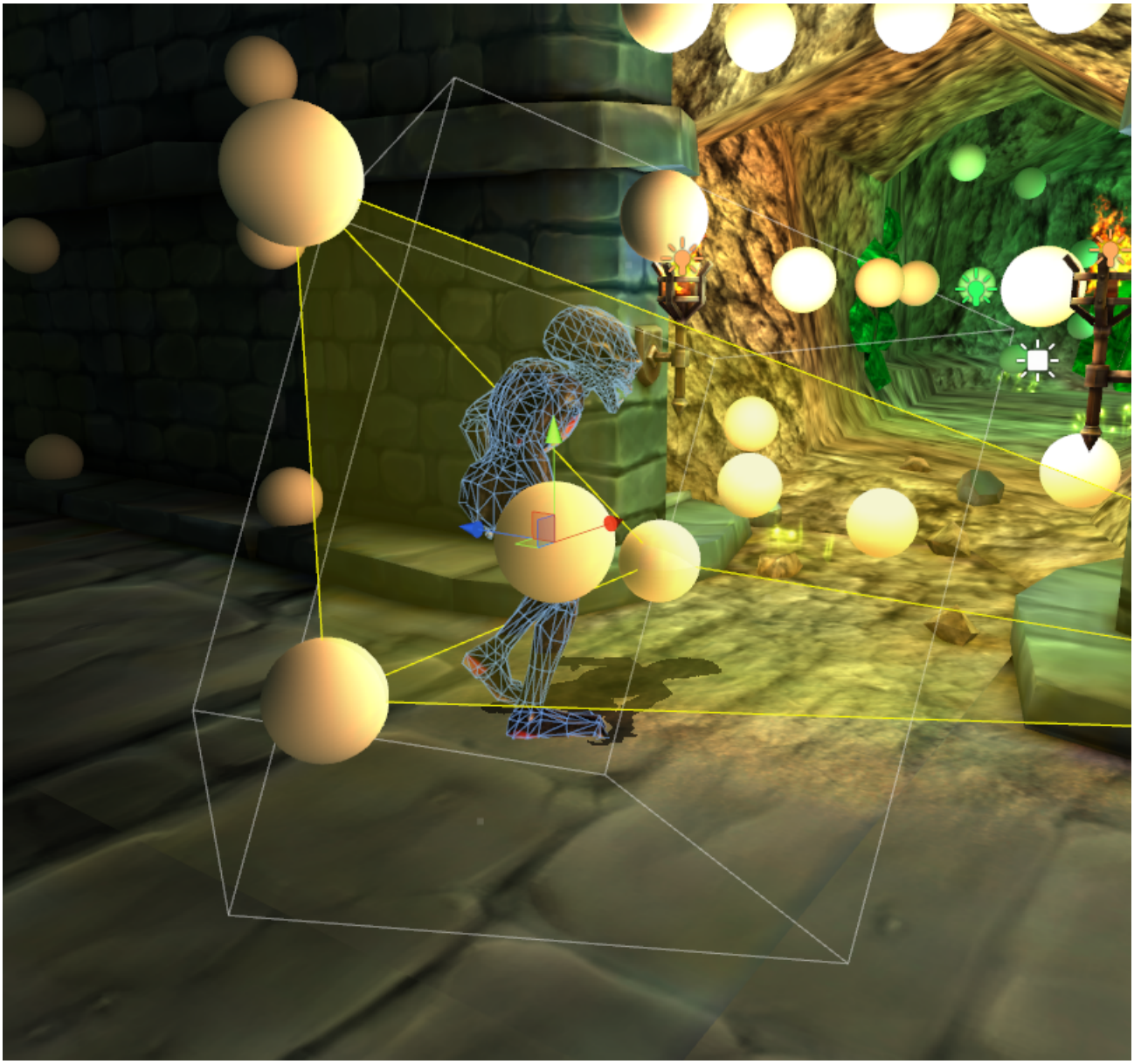

Figure 4-15 Multiple light probes

4.2.3 ASTC texture compression in Unity

ASTC is a technology developed by ARM that has been adopted as an official extension to both the OpenGL and OpenGL ES graphics APIs.

ASTC enables you to reduce the memory required by your application and reduce the memory bandwidth required by the GPU. ASTC offers texture compression with higher quality, lower bitrate and with more control options than other compression formats.

ASTC includes the following features:

- Bit rates range from 8 *bits per pixel* (bpp) to less than 1 bpp. This enables you to fine-tune the tradeoff 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.

ASTC texture compression is available in Unity version 4.3 and above.

Figure 4-16 ASTC settings

If ASTC is supported by your device, use it to compress the textures in your 3D content. If your device does not support ASTC, try using ETC2.

Figure 4-17 ETC2 settings

Note

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.

Selecting the correct format for ASTC textures in Unity

When compressing an ASTC texture within Unity 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.

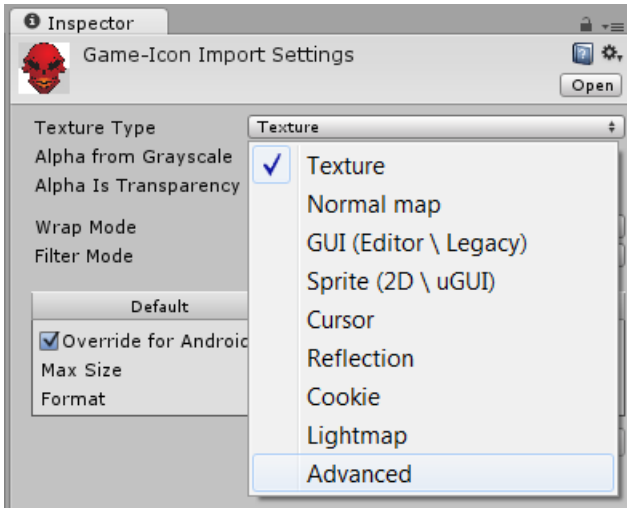

Figure 4-18 Texture settings

Do not compress all 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.

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 a RGBA 8 bit per channel texture of 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 Mip mapping

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

Mip maps 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 than 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 Mip maps do the following:

1. Select a texture in the **Project window**.
2. Enable the **Generate Mip Maps** option in the **Inspector**.
3. change the type to **Advanced**.

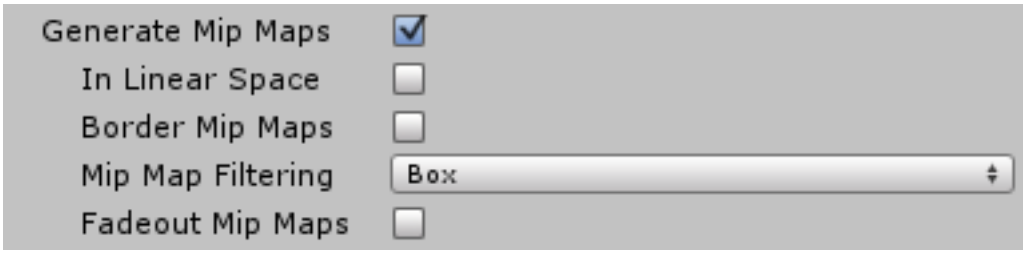

Figure 4-19 Mip map settings

Normally, 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, even with a filter to interpolate the pixel colors, some accuracy is lost in this process.

If a texture has Mip map 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 Mip mapping is it requires 33% more memory to store the texture data.

Mip maps and GUI textures

You do not usually require Mip mapping 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 Mip map chain.

To change this setting select a texture in the **Project** window then go in the **Inspector** and look at **Texture Type**. Either set the type to **GUI** or set it 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.

The method used to draw the skybox in the student project is to render 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 `RenderFX Skybox Cubed` shader. This shader inversely maps the cubemap to the normal of your skybox, and your skybox cubemap. 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 student project uses the techniques built into Unity for shadows.

_____

Unity has a number of options for shadows that can impact the performance of your game:

Hard/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.

The student project uses 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 real time shadows, the more static shadows you can bake into the scene fewer less real-time calculations your GPU must do.

Figure 4-20 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.

4.2.7 Occlusion Culling in Unity

Occlusion Culling is a process that disables the rendering of objects when they are obscured from the view of the cameras view. 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 4 includes an occlusion culling system called Umbra. For more information on Umbra see occlusion culling in the Unity Manual.

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.

Note

This technique is only available in Unity Pro.

4.2.8 OnBecameVisible()and OnBecomeInvisible() callbacks in Unity

With callbacks such as `MonoBehaviour.OnBecameVisible()` and `MonoBehaviour.OnBecameInvisible()` your scripts can be notified if their associated game object moves from being outside of a camera frustum to being inside a camera frustum or the inverse. When the visibility state changes your application can act accordingly.

In the student project uses `OnBecameVisible()` and `OnBecameInvisible()` when rendering the reflection of the water in the dungeon room. The reflection of the pool is rendered using 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:

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.

4.3 Asset optimizations in Unity

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 large solid seamless ones assets.

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, 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 parent of all the meshes that you want to combine and then attach to it a script.

For more information on the mesh combine script, see the Unity documentation.

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, when you place 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 atlases 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 new unique material and this 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.

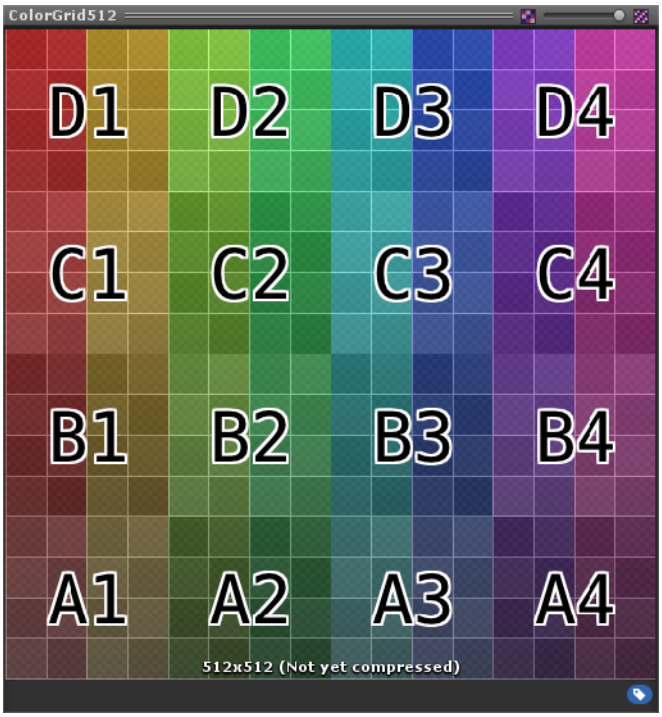

Figure 4-21 Texture atlas

Chapter 5
Advanced Techniques in Unity

This chapter lists a number of advanced techniques you can use in your Unity application. It contains the following sections:

5.1 Implementing reflections with a local cubemap in Unity

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 spherical surface is mapped into 2D:

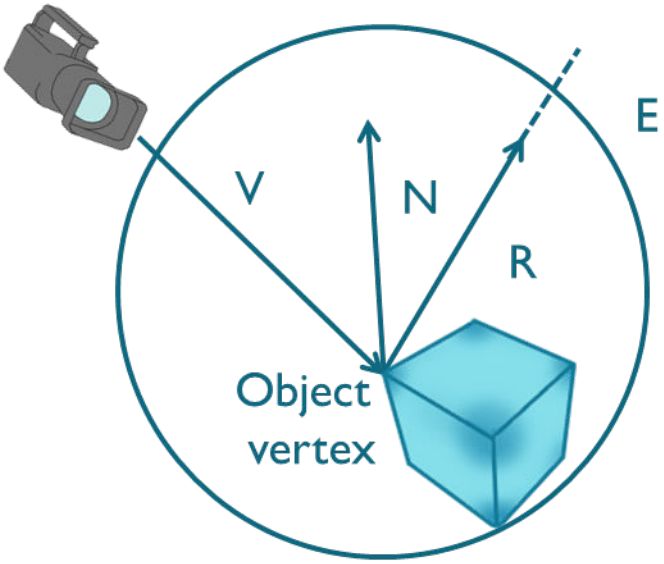

Figure 5-1 Environment map on a sphere

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 5-2 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.

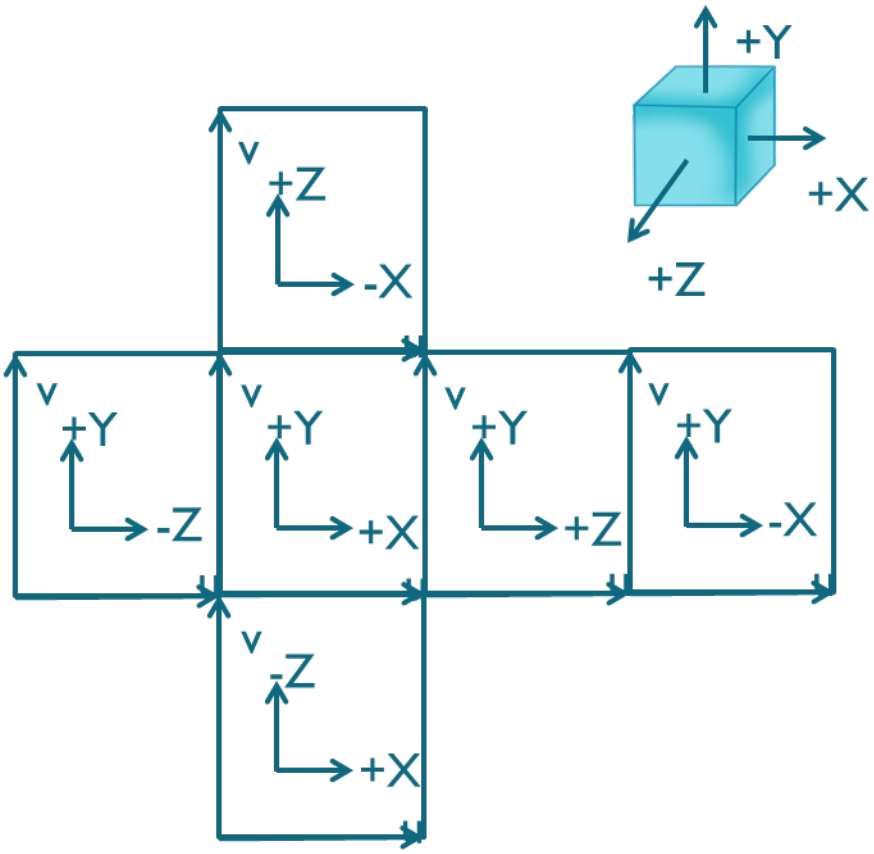

Figure 5-3 Unfolded Cube

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.

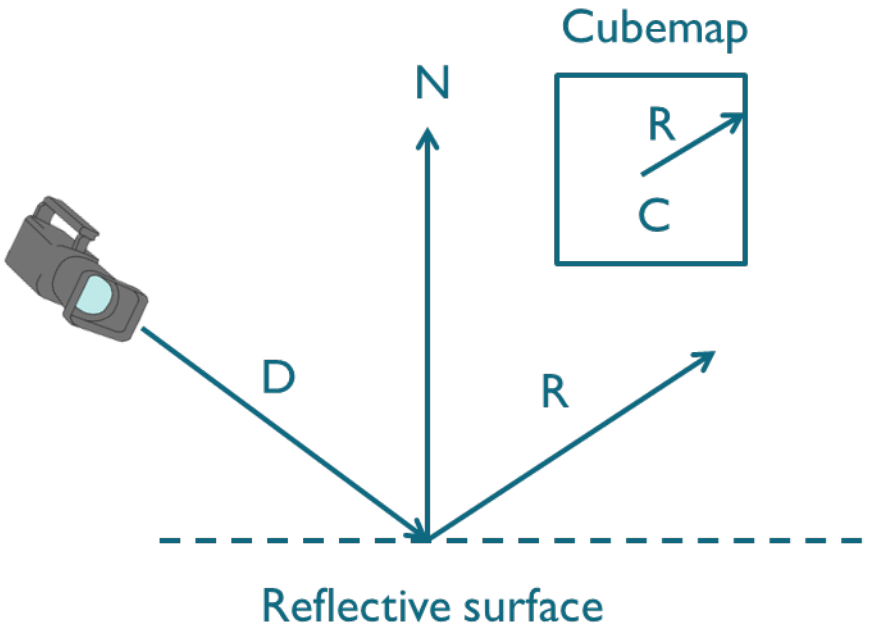

Figure 5-4 Infinite reflections

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.

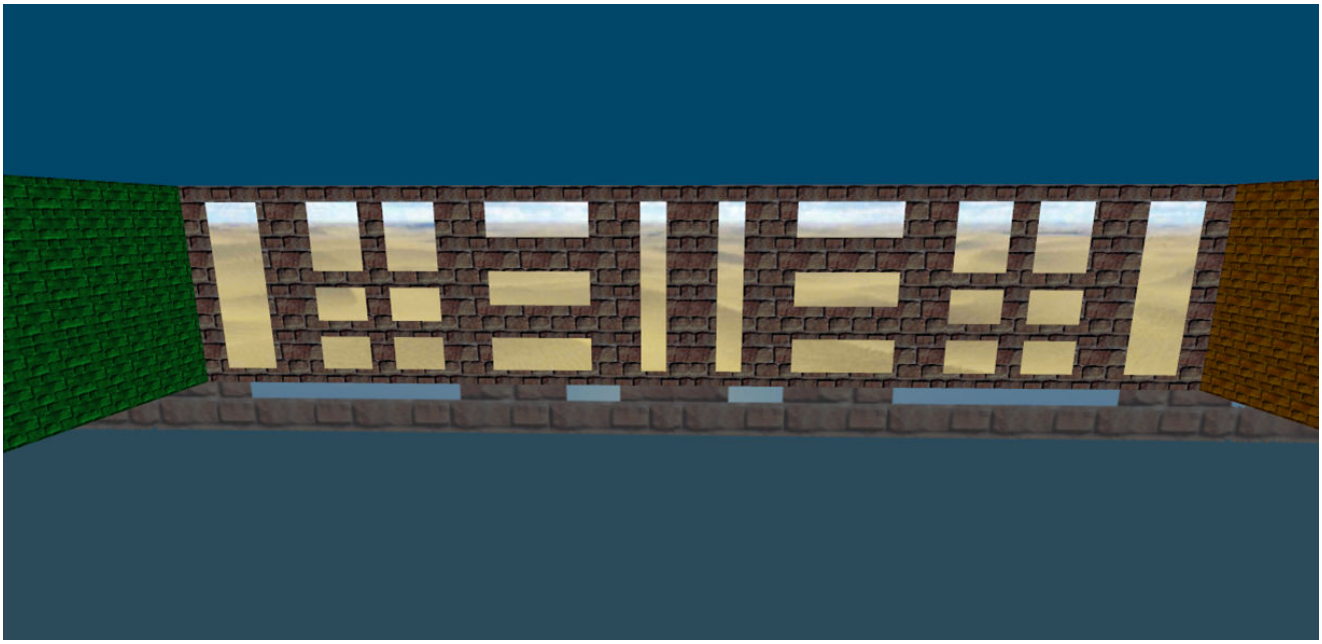

Figure 5-5 Incorrect reflections

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.

Kevin Bjorke proposed a solution to this problem in 2004. This involves binding to the local geometry in the procedure to calculate the reflection. See *GPU Gems 2004*, chapter 19.

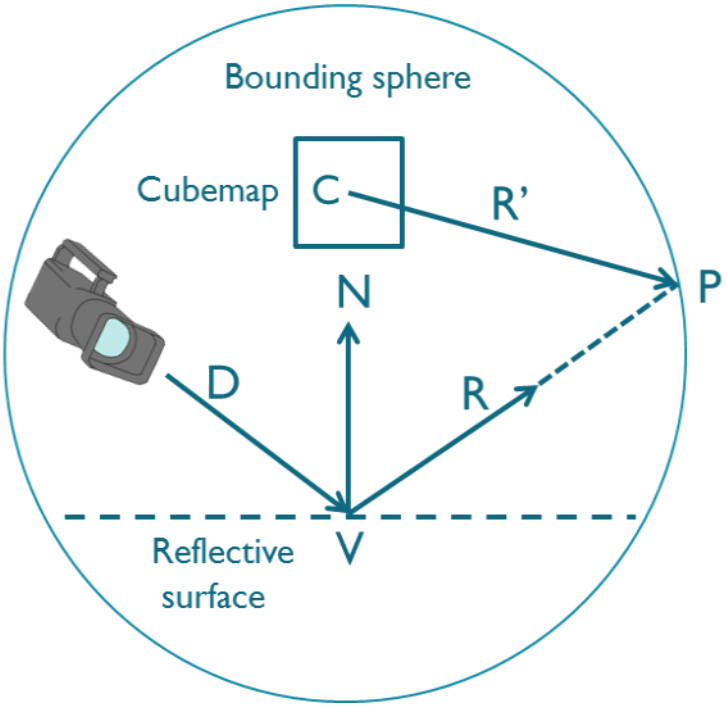

Figure 5-6 Local correction using a bounding sphere

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*

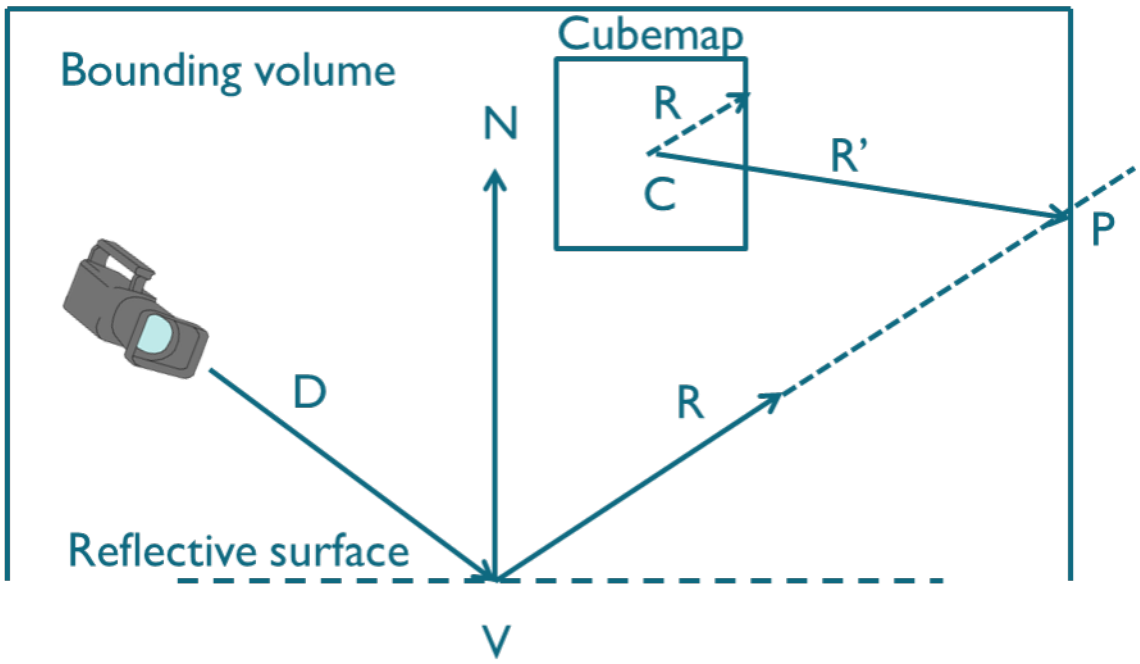

Figure 5-7 Local correction using a bounding box

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 5-1 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 image shows the scene with correct reflections using local cubemaps.

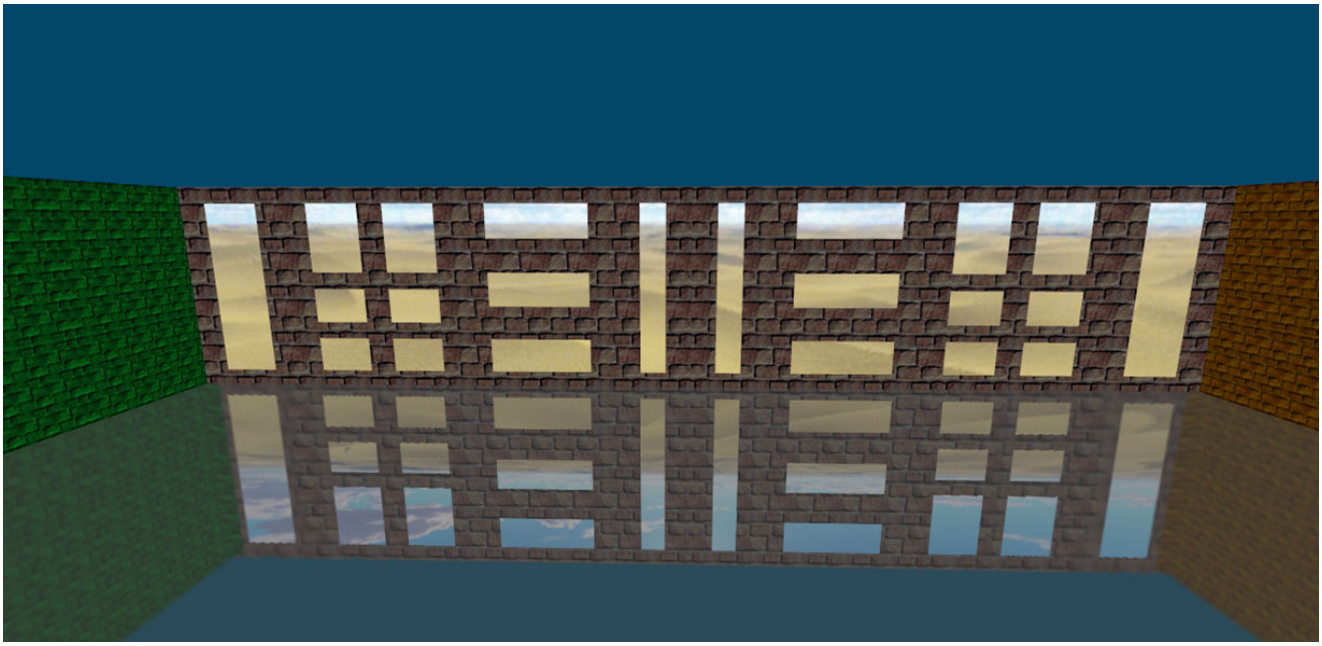

Figure 5-8 Correct reflections

Shader Implementation

The following code shows the shader implementation in Unity of 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.

```
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 *5.2 Ray-box intersection algorithm in Unity* on page 5-61.

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

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 for a tool that saves the images, see *5.3 Source code for editor script to generate cubemaps for Unity* on page 5-64. 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.

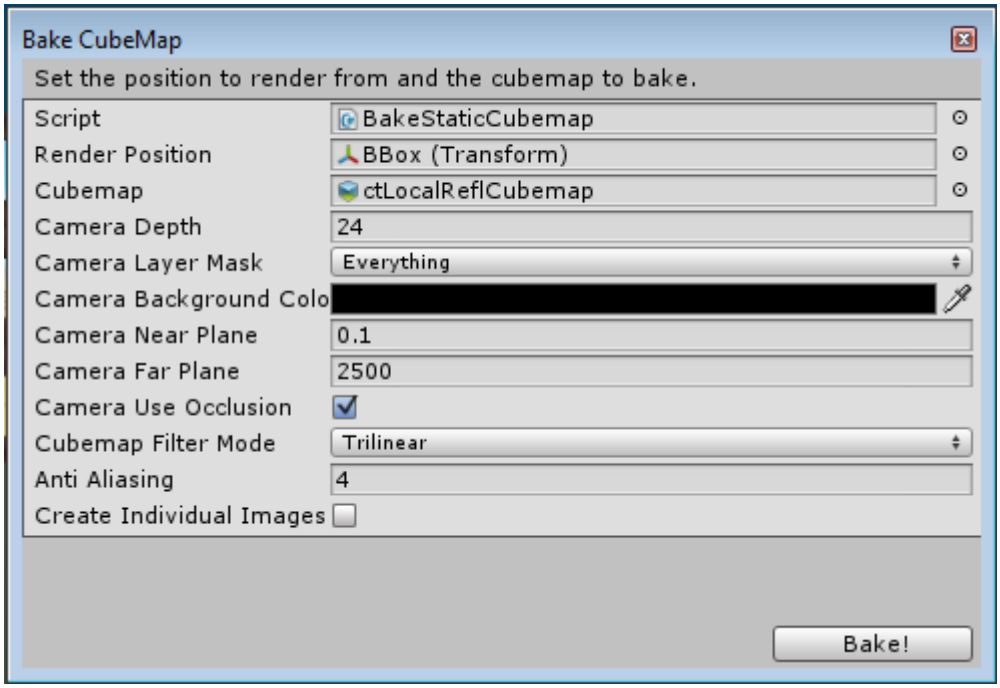

Figure 5-9 Cubemap Bake tool interface

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 CubeMapGen tool and load cubemap images following the relations shown in the following table.
3. Save 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 5-2 Equivalence of cubemap face index between CubeMapGen and Unity

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

Table 5-2 Equivalence of cubemap face index between CubeMapGen and Unity (continued)

AMD CubeMapGen	Unity
Z+	+Z
Z-	-Z

The following images show CubeMapGen after loading the six cubemap images and after applying a Gaussian filtering to achieve a *frosty* effect.

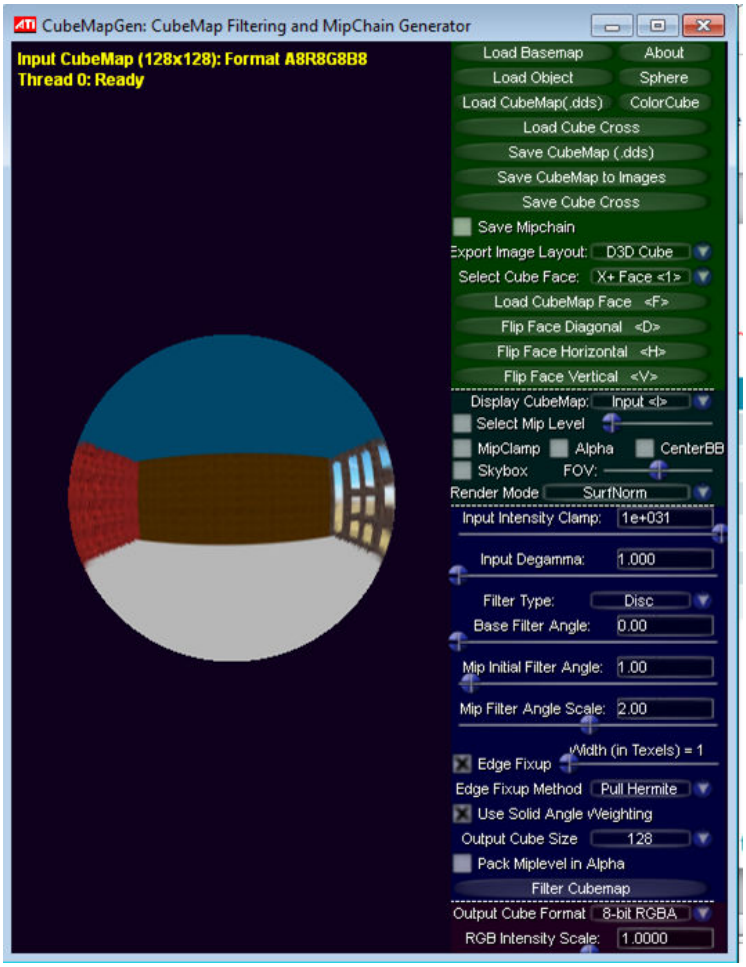

Figure 5-10 CubeMapGen

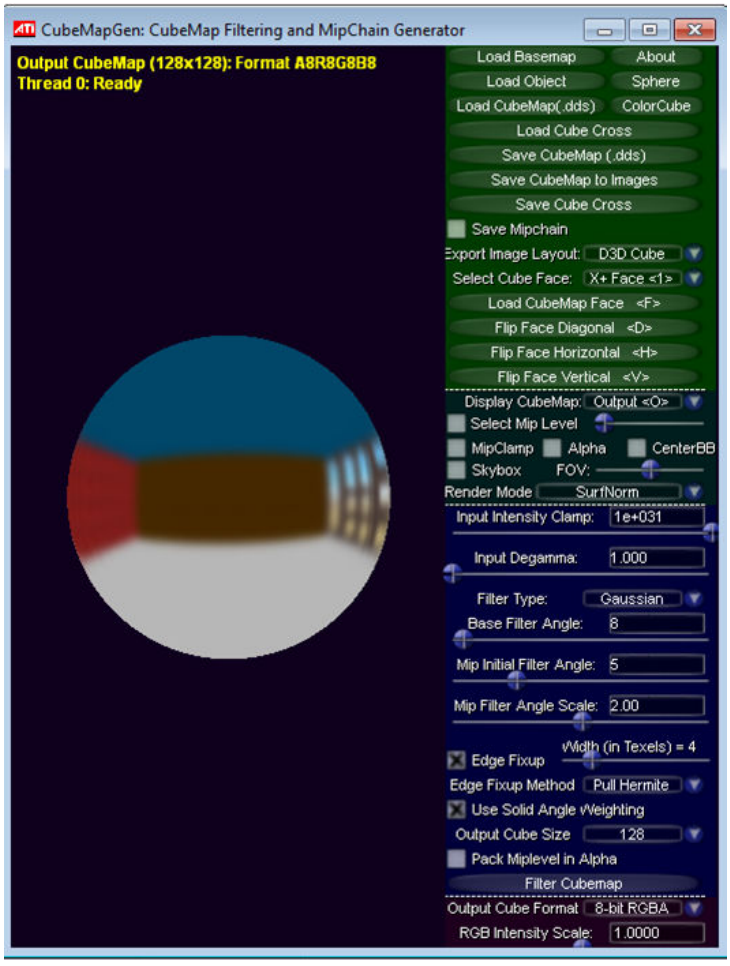

Figure 5-11 CubeMapGen showing frosty effect

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

Table 5-3 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

Figure 5-12 Reflection with frosty effect

The following images summarize the work flow to apply filtering to Unity cubemaps with the CubeMapGen tool.

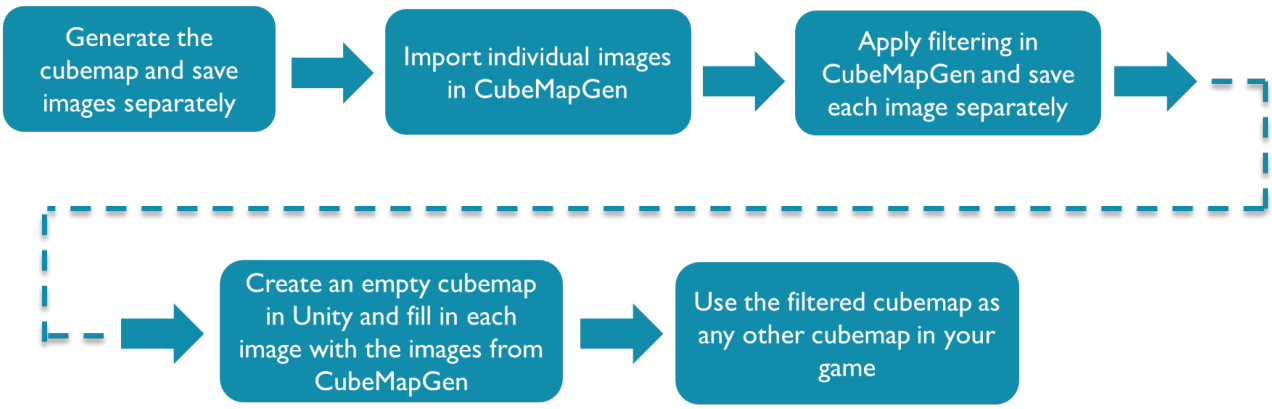

Figure 5-13 Cubemap filtering workflow

5.2 Ray-box intersection algorithm in Unity

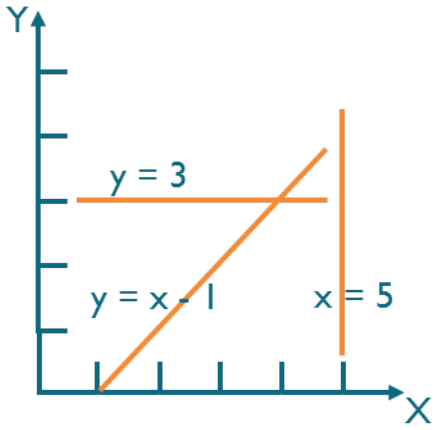

Figure 5-14 Graph with line equations

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

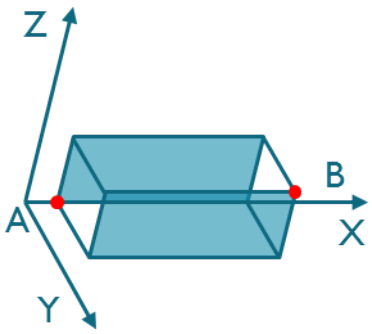

Figure 5-15 Axis aligned bounding box

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.

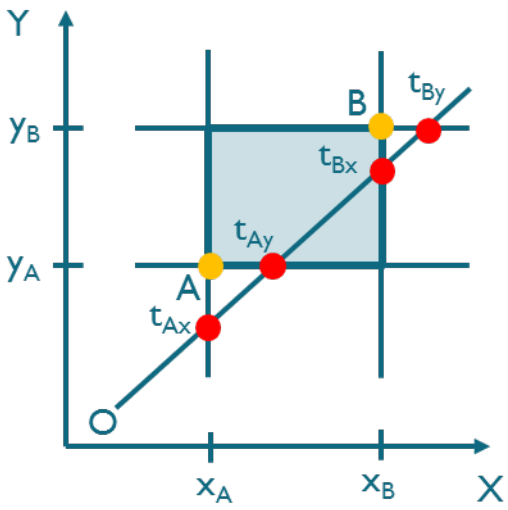

Figure 5-16 Ray-box intersection 2D representation

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.

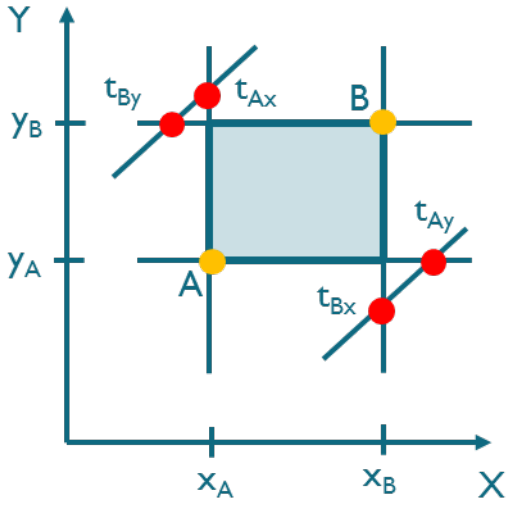

Figure 5-17 Ray-box with no intersection

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.

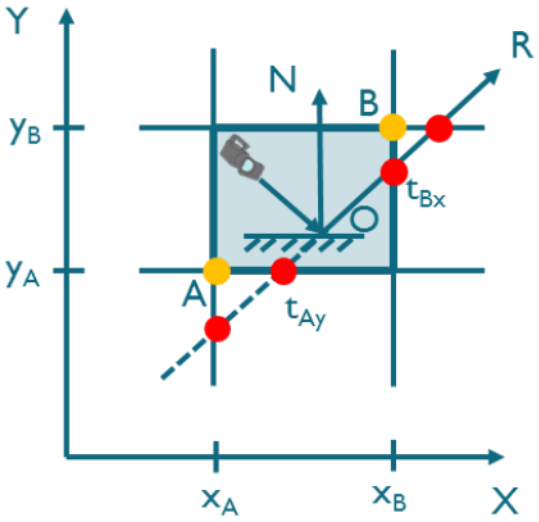

Figure 5-18 Ray-box intersection in BBox

5.3 Source code for editor script to generate cubemaps for Unity

```
/*
* 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!");
    }
}
```

5.4 Custom shaders in Unity

This section contains the following subsections:

5.4.1 About custom shaders in Unity

Unity has a number of built in shaders that are very useful for beginners. If you create your own material it is assigned a default Diffuse shader. If you click on the shader drop down menu in the **Inspector** you can see all the available built-in shaders divided into families.

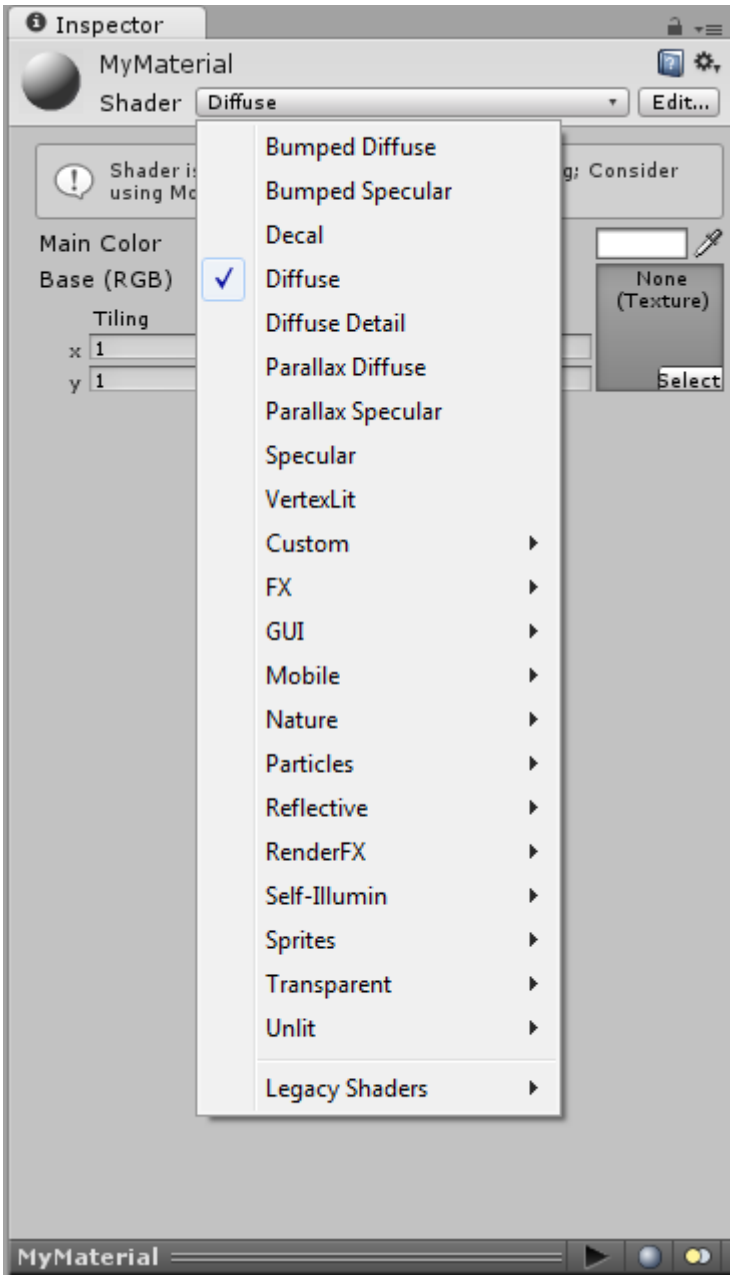

Figure 5-19 Unity built-in shaders

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.

Given the number of existing shaders available you might think you shall not require your own, however 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 *5.1 Implementing reflections with a local cubemap in Unity* on page 5-48.

In Unity there are mainly two ways of writing shaders:

- Surface shaders
- Vertex and fragment shaders.

Surface shaders 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 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.

5.4.2 Shader structure in Unity

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 *C for Graphics* (Cg). Cg is based on C language with some modifications to make it more suitable for GPU programming. 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 runs on the current 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 replace 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`.

5.4.3 Compilation Directives in Unity

You pass compilation directives as `#pragma` statements. The compilation directives indicate the shader functions to be compiled. Each snipped 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 (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 `d3d9` renderers used by Unity Editor:

```
#pragma only_renderers gles gles3 [opengl, d3d9]
```

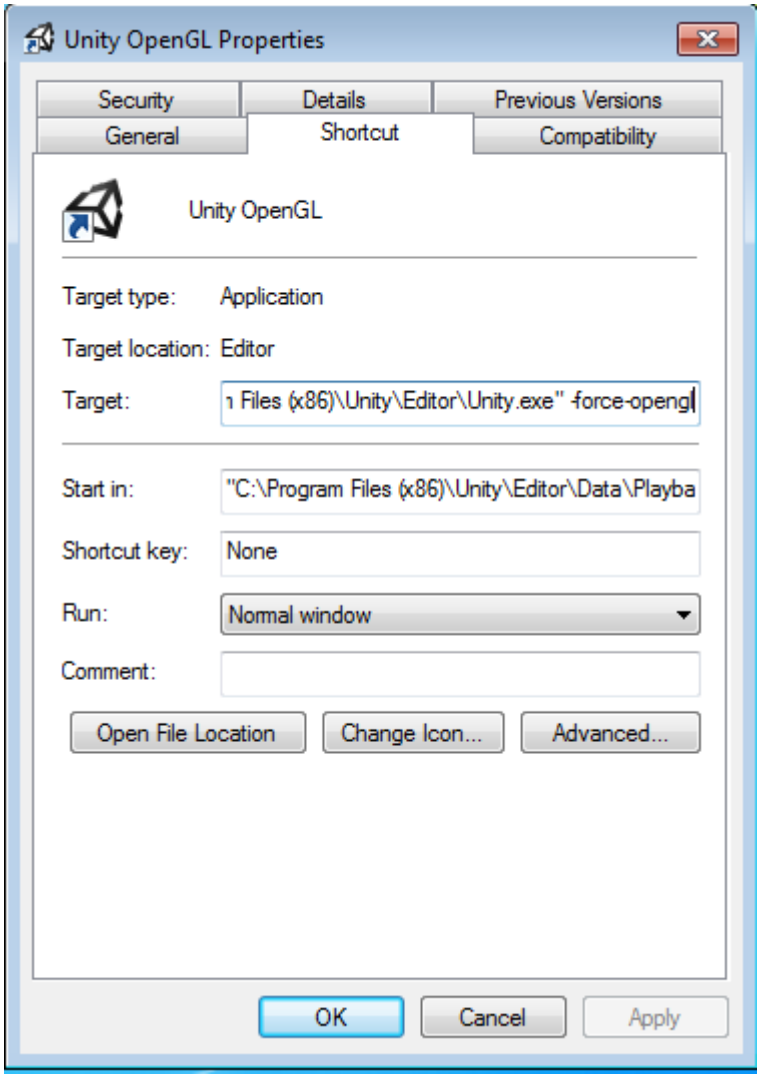

Figure 5-20 Forcing Unity to use the OpenGL renderer in the Editor

If you are working on a desktop workstation targeting OpenGL ES 2.0 or 3.0 devices, set OpenGL as the Editor renderer instead of the default Direct3D renderer to avoid any unnecessary warnings or errors during Direct3D shader compilation. To do this you must edit the target field of the Unity shortcut and pass the parameter `-force-opengl` to the Unity executable:

```
"C:\Program Files (x86)\Unity\Editor\Unity.exe" -force-opengl
```

5.4.4 Includes in Unity

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 (x86)\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 our 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 5-4 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 a components of a vector
_Time	Current time and fractions in a vector (t/20, t, t*2, t*3)

5.4.5 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 below picture 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.

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 which 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.

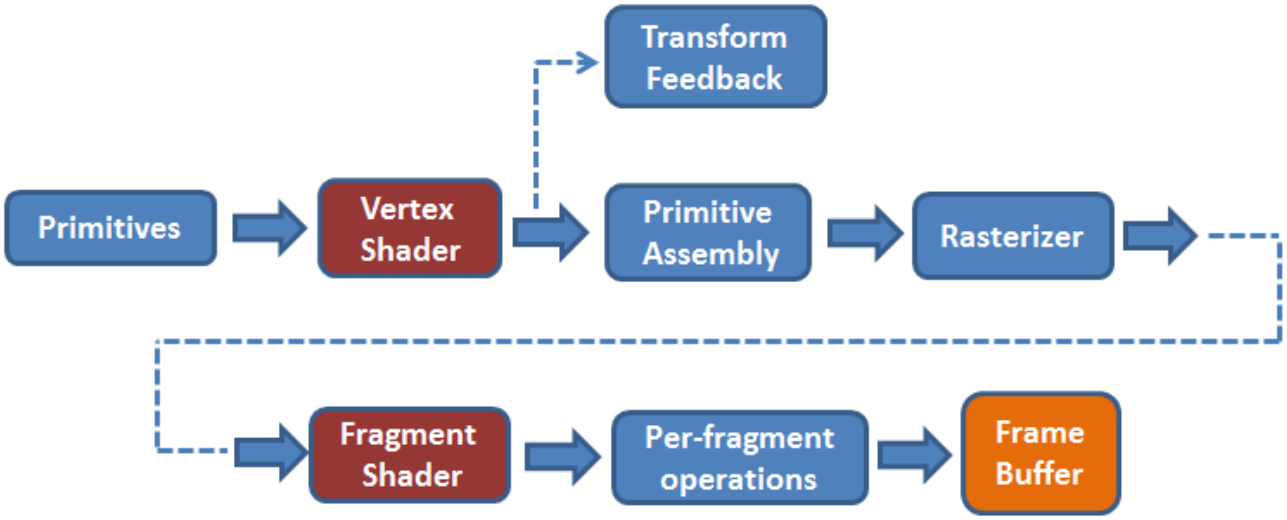

Figure 5-21 OpenGL ES 3.0 Programmable Pipeline

5.4.6 Vertex Shaders in Unity

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 *5.4.2 Shader structure in Unity* on page 5-68.

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 in the 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 also 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.

5.4.7 Vertex shader input in Unity

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.

5.4.8 Vertex shader output and varyings in Unity

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 is 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 through `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.

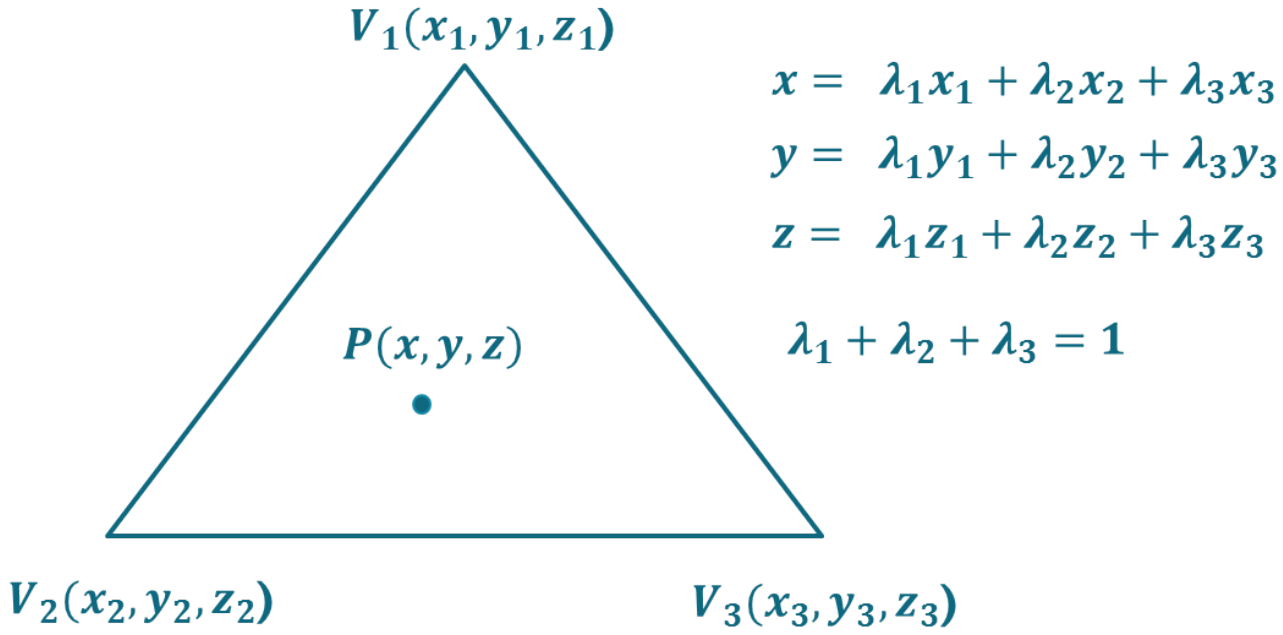

Figure 5-22 Linear interpolation using barycentric coordinates

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

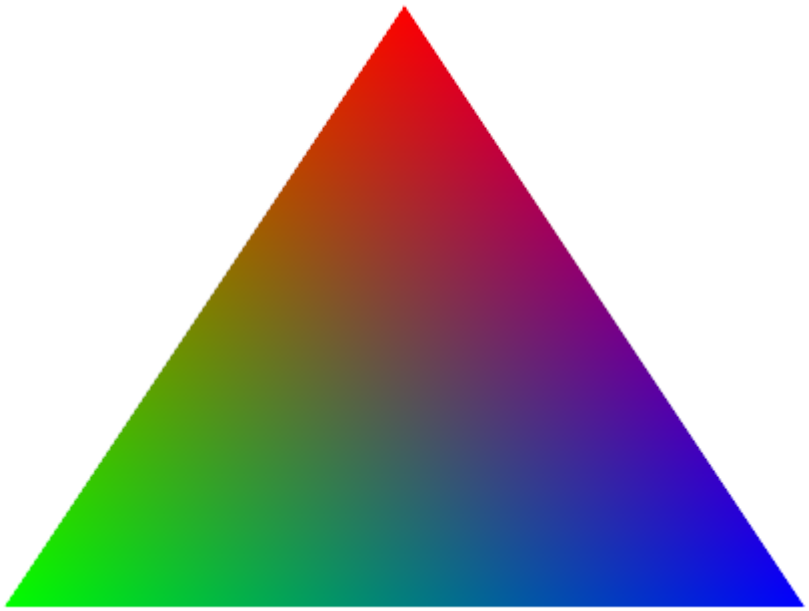

Figure 5-23 Color Interpolation

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 linearly interpolable value calculated 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.

5.4.9 Fragment Shaders in Unity

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 amount 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 *5.4.2 Shader structure in Unity* on page 5-68, 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.

5.4.10 Providing data to shaders in Unity

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 as shown in the following image:

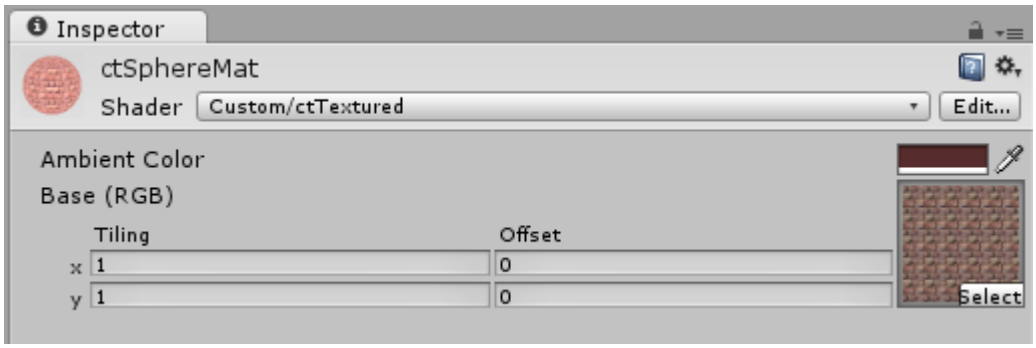

Figure 5-24 Properties in Material Inspector

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 5-5 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;
}
```

5.4.11 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 picture shows the output of the shader `ctReflLocalCubemap.shader` applied to the reflective surface of the floor from *5.1 Implementing reflections with a local cubemap in Unity on page 5-48*. The output color of the fragment shader 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 blue color zone in the floor indicates that the reflected vector has a strong Z component, that is, they are mostly oriented to toward the Z Axis. The blue part must show the reflection coming from that direction, that is, from the windowed wall.

The green zone indicates a predominance of reflected vectors oriented to the Y axis or up vector, that is, the reflection of the ceiling.

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.

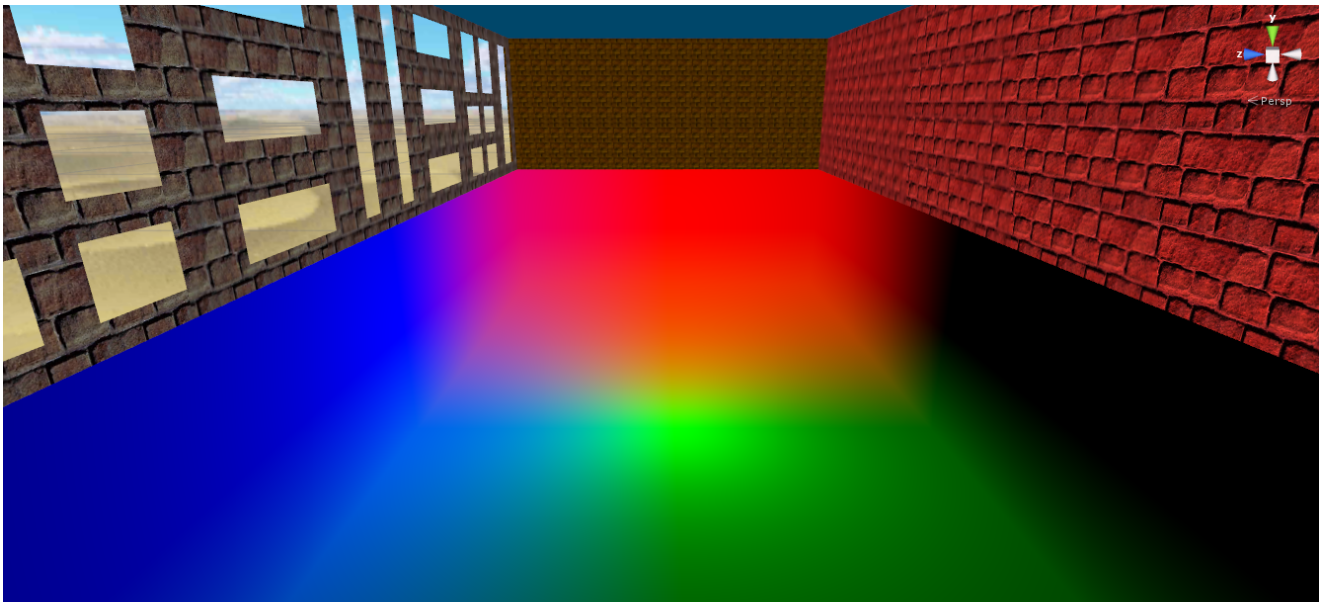

Figure 5-25 Shader debugging with multiple colors

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 X component of the normalized local corrected reflected vector:

```
float3 normLocalCorrReflDirWS = normalize(localCorrReflDirWS);

return float4(normLocalCorrReflDirWS.x, 0, 0, 1);
```

In this case the output is only the reflections coming mainly from the wall that is in front of the camera. That is, the wall oriented to the X axis and partially from the extremes of the walls at both sides of the camera. If you move closer to the camera the X component of the reflected vector becomes gradually zero (black color) because the reflections come mainly from other walls that are oriented to Y and Z directions.

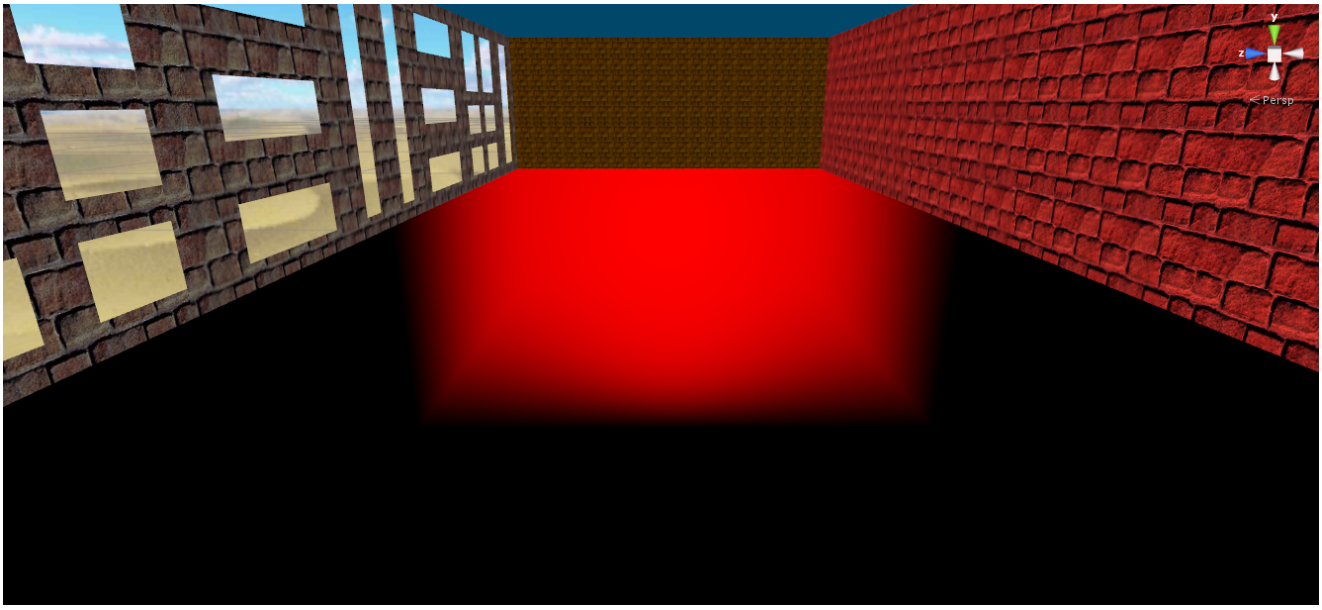

Figure 5-26 Shader debugging with a single color

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.