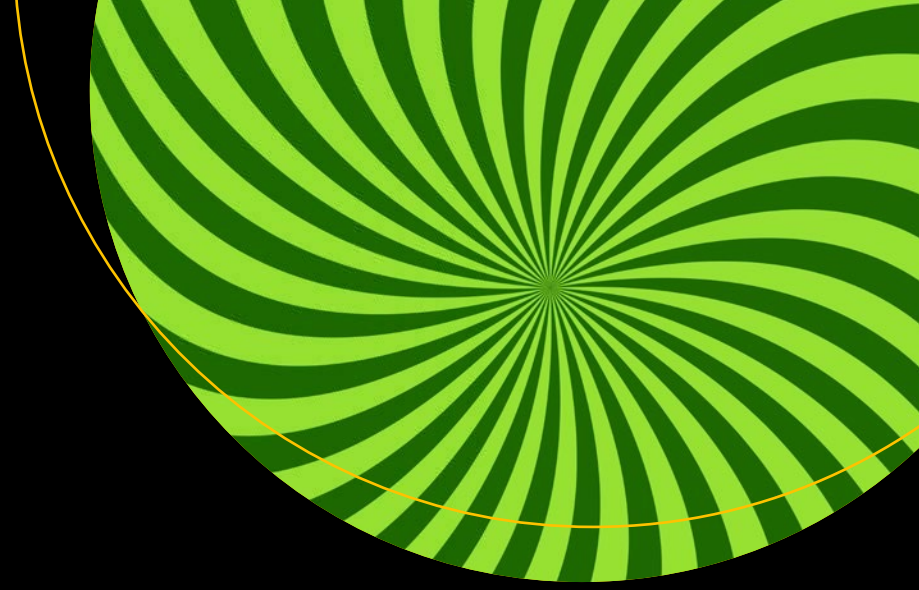

Physically Based Shader Development for Unity 2017

Develop Custom Lighting Systems

Claudia Doppioslash

Apress®

Physically Based Shader Development for Unity 2017

Develop Custom Lighting Systems

Claudia Doppioslash

Apress®

Physically Based Shader Development for Unity 2017

Claudia Doppioslash
Liverpool, Merseyside, United Kingdom

ISBN-13 (pbk): 978-1-4842-3308-5 ISBN-13 (electronic): 978-1-4842-3309-2
https://doi.org/10.1007/978-1-4842-3309-2

Library of Congress Control Number: 2017962301

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Pramila Balan
Development Editor: Matthew Moodie
Technical Reviewer: Druhin Mukherjee
Coordinating Editor: Prachi Mehta
Copy Editor: Kezia Endsley

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-3308-5. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Pe mê moæ e mê nònno

To my mother and my grandfather

Contents

About the Author

Claudia Doppioslash is a game developer and a functional programmer.

Game development and functional programming are fighting in her head for dominance, and so far, neither one has taken over completely.

She writes on `ShaderCat.com` about shader development, graphics programming, and DCC scripting (in Blender and Houdini).

She speaks at programming conferences and is a Pluralsight author (check out her *Developing Custom Shaders in Unity* course over there).

She spends her free time mulling over game AI research problems, attempting to become a decent artist, learning Japanese, and praising her cat.

She speaks three languages fluently (Zenéize, Italian, and English), and one badly (Japanese).

If you want to say hi, tweet `@doppioslash` or `@shadercat`.

Her personal web page is at `www.doppioslash.com`.

Acknowledgments

Let it never be said that one person alone could singlehandedly complete a book. As is normally the case, many people helped me along the way, some by way of it being their job, some by having been semi-reluctantly dragged in. I'm going to attempt to list everyone who should be thanked:

Julian Fisher (`julian-fisher.com`), for making all the figures included in this book and lending his brain to the (nontrivial) task of figuring out what they should look like in the first place.

Prachi Mehta, coordinating editor, for her heroic sangfroid in the face my ever-slipping deadlines; Druhin Mukherjee, technical reviewer, for being precise, yet encouraging, in his feedback; Pramila Balan and Celestin Suresh John, acquisition editors, for having gotten me into this.

Nico Orru, for lending his sublimely nitpicky eye for detail, and his knowledge of the graphics pipeline, to the cause of ridding this book of mistakes, imprecisions, and bad writing.

Anna Limodoro, for much needed aid, encouragement, and general counsel.

Jason Chown and Clemens Wangering, from Starship, who created the chance for me to spend the better part of two years learning physically based shader development.

Any remaining mistakes, bad writing, and narrowly averted disasters, are only my fault and nobody else's, etc.—you know the drill.

Introduction

I assume you currently have this book in your hands (or on your e-reader) because you're interested in Unity, or you're interested in physically based shading, or both.

Shaders are cool, as the Shadertoy web site can attest, and Unity is a good tool through which to learn how to write them. Physically Based Shading is also cool, as the last few years' worth of AAA games can attest, and Unity is again a convenient way to dip in, without having to write your own renderer.

This book will teach you shader development and physically based shading, using Unity as a convenient medium that keeps the effort needed to learn both within sanity thresholds.

Who This Book Is For

I'm going to assume that you, the reader, have some Unity experience, but little or no shader or coding experience. Some programming experience will help, as we're not going through the absolute basics of dealing with code.

You might be a technical artist, wanting to learn the principles of physically based shading, or maybe learn how to use code for shader programming, rather than node-based interfaces.

You might be a game programmer, wanting to learn how to get started with shader programming, or wanting to implement some technique from physically based shading.

In both cases, this book should work for you. Beware that the book includes some math, but understanding it is not strictly necessary. You'll get to the end fine, even if you skip the math, but you'll learn less.

What You'll Learn

The following topics are covered in this book.

- Shader development in Unity
- Graphics pipeline
- Writing Unlit shaders
- Writing Surface shaders
- Physically Based Shading theory
- How to research custom lighting models
- How to implement physically based custom lighting models in Unity
- The state of the art in custom lighting models
- How to hook your custom lighting model implementation into the Unity Standard shader functionality

- How to implement advanced techniques, such as translucency
- Debugging shaders
- Advice on writing shaders that are easy for artists to use
- How to keep up with the ever-progressing rendering advancements

What Do You Need to Use This Book

You need a PC running any version of Unity between 5.6 and 2017.2. Outside of that optimal version range, higher version numbers might break the code, and going too far back (before Unity 5) will definitely break the code.

How This Book Is Organized

This book is divided in three parts and 18 chapters. It's meant to be read in sequence, but if you know some parts already, you can skip ahead.

If you're a seasoned shader developer, looking for knowledge specific to Unity, you might want to skim Chapter 1 and then skip to Chapter 5.

If you are only interested in physically based shading and are already handy with graphics programming and Unity shader development, you should skim Chapter 1 and then skip to Chapter 8.

If you're a programmer but haven't done shader development, I recommend reading the whole book in sequence.

If you're a technical artist and have never written a line of shader code (you may be used to node editors), read the book in order and pay particular attention to Chapter 2, as it'll get you started in writing shader code in Unity.

Part I: Introduction to Shaders in Unity

This first part will get you from shader newbie to having working knowledge of the important graphics programming concepts and being handy at writing non-physically based shaders.

Chapter 1: How Shader Development Works

This chapter covers many foundational concepts about rendering and graphics programming. You'll get an overview of what shaders are, how the graphics pipeline works, the process of rendering, the behavior of light, and shader and render types.

Chapter 2: Your First Unity Shader

In this chapter, you'll get started writing shaders in Unity. It covers project setup, making a scene, the syntax, and the parts that compose a Unity shader. We're going to write a simple Unlit, monochrome shader.

Chapter 3: The Graphics Pipeline

This chapters explains how the graphics pipeline works, and how different parts of a shader hook into it, what data is sent in it, how it's processed, and some things you can do with it.

Chapter 4: Transforming Coordinate Spaces

Coordinate spaces are a necessary, but often confusing, part of the graphics pipeline. This chapter presents and explains each of the commonly used coordinate spaces, where they are used in the graphics pipeline, and what tools Unity gives to transform between them.

Chapter 5: Your First Unity Lighting Shader

This chapter introduces some common lighting concepts (such as the role of the angle of incidence) and approximations (such as diffuse and specular), and how to implement them within an Unlit shader. By the end of the chapter, you'll have written your first custom lighting shader.

Chapter 6: Specular Implementation

Continuing from the previous chapter, we're going to implement a specular approximation to complete the diffuse one from the previous chapter. This chapter also explains how to support more than one light within an Unlit shader, a practical use of ShaderLab passes.

Chapter 7: Surface Shaders

So far we're only used Unlit shaders, because they're more straightforward and they don't hide where the graphics pipeline comes in contact with shaders. But they can be quite verbose, so this chapter introduces Surface shaders and explains how they can save you quite a bit of time and typing. We're going to reimplement the Unlit shader from the two previous chapters, in a surface shader with a custom lighting function.

Part II: Physically Based Shading

This part is entirely dedicated to physically based shading, from theory to implementation. Now that you have experience implementing shaders in Unity and you have the solid basics in how rendering works, you can focus your attention completely on writing shaders according to physics principles.

Chapter 8: What Is Physically Based Shading?

This chapter presents *microfacet* theory and corrects some simplifications we had made in explaining how light works in Chapter 1. We go through Fresnel reflectance, index of refraction, as well as how we can measure light, and represent the behavior of light hitting a material with a function. The chapter explains what makes a lighting model physically based and explains every part of the rendering equation.

Chapter 9: Making a Shader Physically Based

To put into practice the concepts of physically based shading immediately, we take our custom lighting surface shader from Chapter 7 and we make it conform to physically based requirements.

Chapter 10: Post-Processing Effects

Post-processing is a necessary part of HDR rendering, which is itself necessary for physically based shading. While the post-process effects stack that Unity offers through the Asset Store is powerful, you'll still need to understand and occasionally be able to implement post effects, and that's what this chapter covers. The chapter also includes an overview of the Unity post-processing stack version 1 and version 2.

Chapter 11: BRDFs Who's Who

It's time to get to know common lighting functions by name and based on what they look like. This chapter presents BRDF Explorer, which is an excellent program developed by Disney Research to develop and analyze custom lighting functions.

Chapter 12: Implementing a BRDF

Now that you know a few physically based lighting functions, it's time to implement one or two. This chapter covers how to gather information and implement a physically based BRDF. We'll implement the `CookTorrance` specular and the `DisneyBRDF` diffuse.

Chapter 13: Hooking into the Standard Shader

This chapter takes our implemented lighting function from the last chapter and explains how to hack it into the Unity standard shader infrastructure, which is going to give you reflections, global illumination, and more, for almost free.

Chapter 14: Implementing Advanced Techniques

Having gone through everything about implementing a BRDF, this chapter explains how to add light phenomena that cannot be described by BRDFs, such as translucency. It also explains the complex mechanisms that cause reflection probes to work.

Part III: Shader Development Advice

Now that you know all the essentials of implementing custom lighting systems in Unity, this part expounds a bit about the art of it—debugging, writing good code, and making sure the artists won't snub your shaders.

Chapter 15: Making Shaders Artists Will Use

There are many ways that programmers make shaders that are too complex for artists to use, without even realizing it. This chapter lists the top five shader usability mistakes that people make and includes some solutions.

Chapter 16: Complexity and Ubershaders

After Chapter 13, you've had quite a bit of exposure about what complexity looks like in a shader system. This chapter explains why it gets to that, and why it's the still the best solution we have, with the current shading languages.

Chapter 17: When Shading Goes Wrong

There is no such thing as writing a shader right on the first try. Hence, you need to know how to track bugs and find problems. This chapter is about profiling and debugging shaders, and the many tools you can use to do that.

Chapter 18: Keeping Up with the Industry

The game and movie industries never sleep; they keep progressing year after year. After having read the entire book, you'll be in a good position to deepen your knowledge by going straight into SIGGRAPH papers. This chapter lists many ways you can keep abreast of the latest developments in rendering and shader development.

PART I

Introduction to Shaders In Unity

This part of the book includes everything you need to know to be a competent shader developer in Unity.

Starting from how to write your first shader, going into the role of each step in the graphics pipeline, and how your shaders hook into it.

We explain how to develop Unlit and Surface shaders, common lighting approximations, the fundamental concepts of light behavior, and rendering.

CHAPTER 1

How Shader Development Works

Shader development is a black art that's essential for game development.

You might have heard about shaders before. The first encounter with shaders sometimes happens due to thorny hard-to-fix rendering issues, which only appear on some rare GPU you can't get your hands on. Or maybe you heard the word "shader" being whispered softly, wrapped in nursery rhymes about developers who poke their noses where they shouldn't, and the dire ends they meet.

Fear not, this book is going to demystify shaders and give you a grounding that will allow you to develop great looking lighting shaders and effects from scratch. We're also going to touch upon the mathematical reasoning necessary to understand and implement correct lighting shaders, so that you'll be able to choose your tradeoffs, to achieve good performance without sacrificing fidelity too much.

This chapter introduces the fundamental knowledge necessary to understand shaders. To make it easier to get started, I'll simplify liberally. We'll go through most of these concepts in more depth in later chapters. I don't assume any knowledge about shaders, but I do assume that you are familiar with Unity and game development.

What Is a Shader?

Going straight to the heart of the matter, a shader is both:

- A simulation made in code of what happens at the surface microscopic level, which makes the final image look realistic to our eyes
- A piece of code that runs on GPUs

Shaders as Light Simulations

To explain the first definition, look at the three images in Figure 1-1. They show you three surfaces made of different materials. Your brain can instantly understand what material an object is made of, just by looking at it. That happens because every material's pattern of interaction with light is very characteristic and recognizable to the human brain. Lighting shaders simulate that interaction with light, either by taking advantage of what we know about the physics of light, or through a lot of trial and error and effort from the artists.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_1

Figure 1-1. *Skin, metal, wood*

In the physical world, surfaces are made of atoms, and light is both a wave and a particle. The interaction between light, the surface, and our eyes determines what a surface will look like. When light coming from a certain direction hits a surface, it can be *absorbed, reflected* (in another direction), *refracted* (in a slightly different direction), or *scattered* (in many different directions). The behavior of light rays, when they come in contact with a surface, is what creates the specific look of a material. See Figure 1-2.

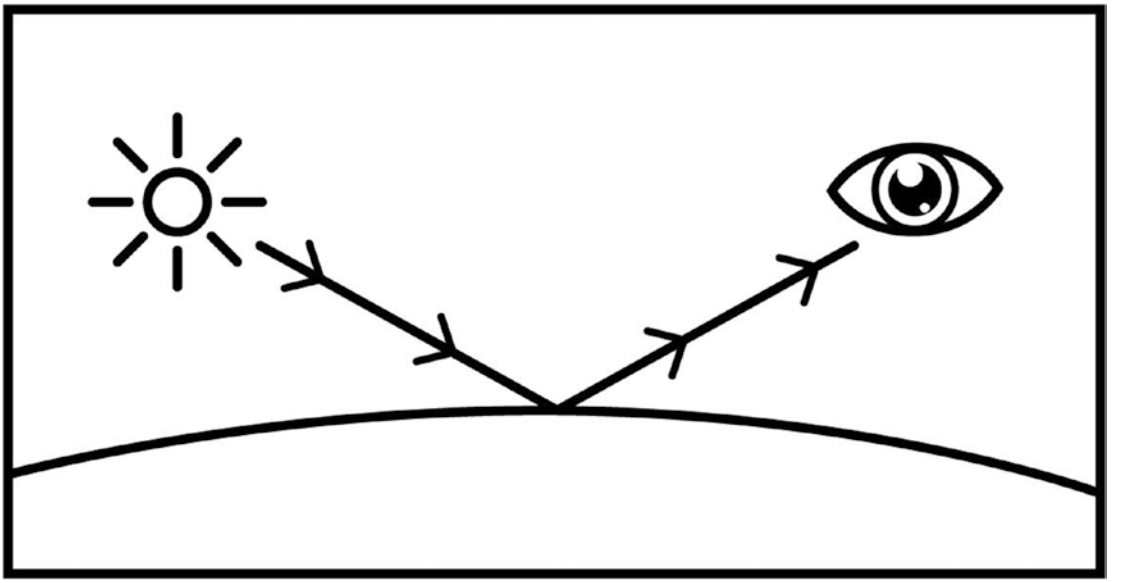

Figure 1-2. *A ray of light, hitting a surface and bouncing off in the direction of our eyes*

Even if a surface looks smooth at the macroscopic level, like skin does, at the microscopic level, it can have micro-facets that scatter light in different directions.

Inside computers we don't have the computational power needed to simulate reality to that level of detail. If we had to simulate the whole thing, atoms and all, it would take years to render anything. In most renderers, surfaces are represented as 3D models, which are basically points in 3D space (vertices) at a certain position, that are then grouped in triangles, which are then again grouped to form a 3D shape. Even a simple model can have thousands of vertices.

Our 3D scene, composed of models, textures, and shaders, is rendered to a 2D image, composed of pixels. This is done by projecting those vertex positions to the correct 2D positions in the final image, while applying any textures to the respective surfaces and executing the shaders for each vertex of the 3D models and each potential pixel of the final image. See Figure 1-3.

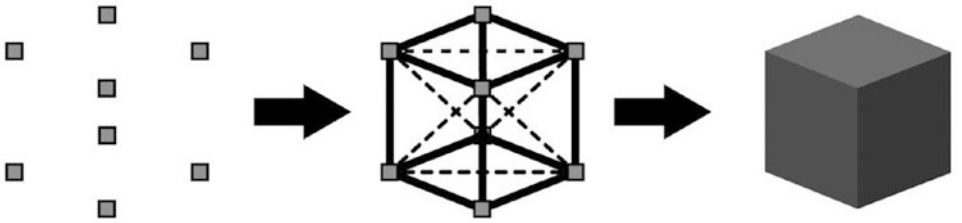

Figure 1-3. *From points, to triangles, to the renderer shaded model*

Regardless of how far you are willing to go in detail when modeling, it's impossible to match the level of detail in the real world. We can use our mathematical knowledge of how lighting works, within a shader, to make our 3D models look as realistic as possible, compensating for not being able to simulate surfaces at a higher level of detail. We can also use it to render our scenes fast enough, so that our game can draw a respectable number of frames per second. Frames per second appropriate for games range from 30 fps to 60 fps, with more than 60 fps being needed for virtual reality games.

This is what physically based rendering is all about. It's basically a catalog of various types of lighting behaviors in surfaces and the mathematical models we use to approximate them.

Rendering as Perspective Drawing

Rendering is conceptually (and mathematically) very similar to the painter's process of drawing from life, into a canvas, using perspective.

The techniques of perspective drawing originated in the Italian Renaissance, more than 500 years ago, even if the mathematical foundations for it were laid much earlier, back in Euclid's times. In our case, the canvas is our final image, the scene and 3D models are reality, and the painter is our renderer.

In computer graphics, there are many ways to render the scene, some more computationally expensive and some less. The fast type (rasterizer-based) is what real-time rendering, games included, has been using. The slow type (raytracing, etc.) is what 3D animated movies generally use, because rendering times can reach even hours per frame.

The rendering process for the fast type of renderers can be simplified like so: first the shapes of the models in the scene are projected into the final 2D image; let's call it the "sketching the outline" phase, from our metaphorical painter's point of view. Then the pixels contained within each outline are filled, using the lighting calculations implemented in the shaders; let's call that the "painting" phase.

You could render an image without using shaders, and we used to do so. Before the programmable graphics pipeline, rendering was carried out with API calls (APIs such as OpenGL and DirectX3D). To achieve better speed, the APIs would give you pre-made functions, to which you would pass arguments. They were implemented in hardware, so there was no way to modify them. They were called *fixed-function* rendering pipelines.

To make renderers more flexible, the programmable graphics pipeline was introduced. With it, you could write small programs, called *shaders*, that would execute on the GPU, in place of much of the fixed-function functionality.

Rendering Process

As mentioned, this type of rendering could be conceptually broken down in two phases:

- The *outline* phase
- The *painting* phase

The outline phase determines which pixels in the final image are going to belong to a certain triangle, by projecting the vertices of the models into the final image, and checking for whether another model is in front, from the camera's point of view. The painting phase calculates the color of each pixel, according to the scene data (lights, textures, and lighting calculations).

The first phase manipulates vertices, the second phase manipulates the information it gets from the first phase and outputs the pixel colors.

Shaders as Code Running on GPUs

As mentioned, there can be many thousands of vertices in a model, and a rendered image can have millions of pixels. Game scenes vary in complexity, according to the platform they're going to run on. On PlayStation 4 Pro, the final image resolution reaches 3840×2160 pixels (commonly called 4k resolution), and a scene can have more than hundreds of thousands of vertices. Typically a shader will run on every vertex in the scene, and on every pixel in the final image. To achieve that real-time rendering speed, we need a special processor that's capable of running very short programs millions of times in just milliseconds. Such a processor is a commonly known as a *Graphics Processing Unit,* or GPU.

Shading is a dataflow process in one direction, which means that vertices, textures, and shaders enter, and then, at the other end, colors exit, and are put into a render target, meaning basically a 2D image. We don't need to know anything about the vertices near the one we're processing, or the pixels near the one we're calculating (at least most of the time), hence all those shaders can be executed independently, at the same time, on a large number of vertices/pixels.

Shader Execution

Figure 1-4 shows how a simple scene is rendered.

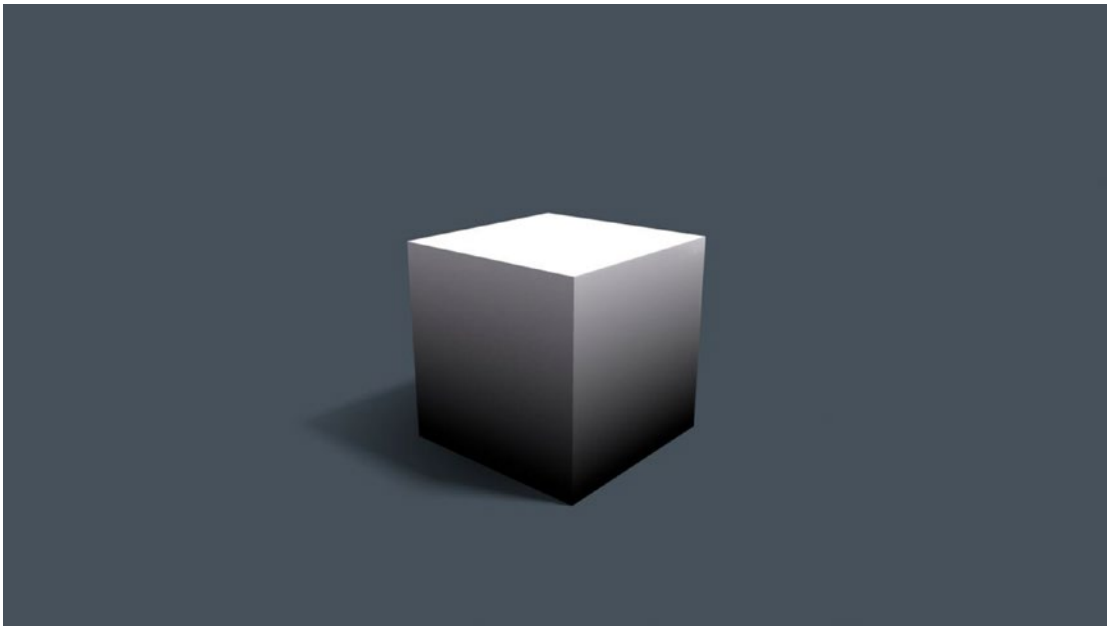

Figure 1-4. *A rendered scene containing only a cube with colored vertices*

This scene has eight vertices, and it has been rendered to a 1920x1080 image (full HD resolution). What is happening exactly in the rendering process?

1. The scene's vertices and their respective data are passed to the vertex shader.
2. A vertex shader is executed on each of them.
3. The vertex shader produces an output data structure from each vertex, containing information such as color and position of the vertex on the final image.
4. Sequences of vertices are assembled into primitives, such as triangles, lines, points, and others. For the purposes of this book, we'll assume triangles.
5. The rasterizer takes a primitive and transforms it into a list of pixels. For each potential pixel within that triangle, that structure's values are interpolated and passed to the pixel shader. (For example, if one vertex is green, and an adjacent vertex is red, the pixels between them will form a green to red gradient.) The rasterizer is part of the GPU; we can't customize it.
6. The fragment shader is run for any potential pixel. This is the phase that will be more interesting for us, as most lighting calculations happen in the fragment shader.
7. If the renderer is a forward render, for every light after the first, the fragment shader will be run again, with that light's data.

8. Each potential pixel (aka, *fragment*) is checked for whether there is another potential pixel nearer to the camera, therefore in front of the current pixel. If there is, the fragment will be rejected.
9. All the fragment shader light passes are blended together.
10. All pixel colors are written to a render target (could be the screen, or a texture, or a file, etc.)

As you can see in Figure 1-5, this cube has colored vertices. The gradient from black to gray in the shaded cube is due to the interpolation happening in Step 4.

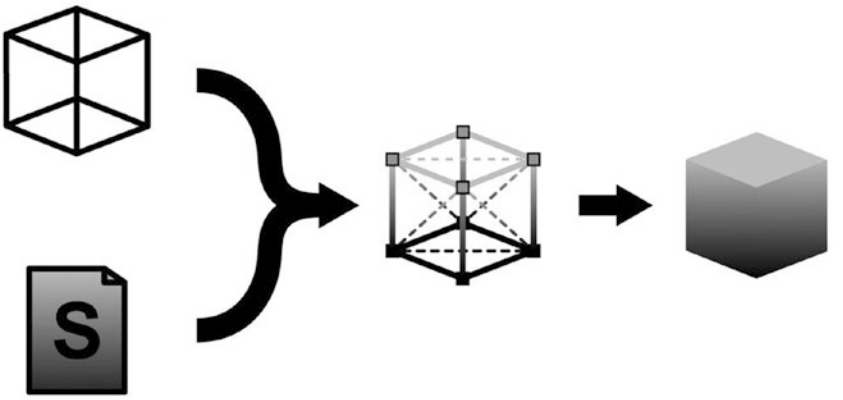

Figure 1-5. *Scene data being sent to the renderer, vertex processing phase, to fragment processing*

This was an overview of how shaders are executed when rendering a scene. Now we'll more accurately define some technical terms that I've mentioned in passing up to now.

Different Types of Shaders

We have already mentioned a couple types of shaders. Here they are and a few more:

- *Vertex shader*: Executed on every vertex.
- *Fragment shader*: Executed for every possible final pixel (known as a fragment).
- *Unlit shader*: Unity-only, a shader that combines a vertex and pixel shader in one file.
- *Surface shader*: Unity-only, contains both vertex and fragment shader functionality, but takes advantage of the ShaderLab extensions to the Cg shading language to automate some of the code that's commonly used in lighting shaders.
- *Image Effect shader*: Unity-only, used to apply effects like Blur, Bloom, Depth of Field, Color Grading, etc. It is generally the last shader run on a render, because it's applied to a render of the geometry of the scene.
- *Compute shader*: Computes arbitrary calculations, not necessarily rendering, e.g., physics simulation, image processing, raytracing, and in general, any task that can be easily broken down into many independent tasks. In this book, we spend a fair amount of time on Unity surface shaders, but we won't cover compute shaders. There are even more types of shaders, but since they are not used as often, we won't mention them.

Coordinate Systems

Every calculation in a shader lives in a particular coordinate system. Think of the Cartesian coordinate system—almost everyone has dealt with it at one point or another. That is a 2D coordinate system, while many of the ones used for rendering calculations are 3D rendering systems. Here's a list:

- *Local (or Object) Space*: The 3D coordinate system relative to the model being rendered
- *World Space*: The 3D coordinate system relative to the entire scene being rendered
- *View (or Eye) Space*: The 3D coordinate system relative to the viewer's point of view (the camera you're rendering from)
- *Clip Space*: A 3D coordinate system that has a range of -1.0 to 1.0
- *Screen Space*: The 2D coordinate system relative to the render target (the screen, etc.)
- *Tangent Space*: Used in Normal Mapping

We're occasionally going to mention coordinate spaces. It's important to be aware of them, even though most of the time, you won't need to deal with them directly. It's very easy to get confused about which space is used in which calculation, so it's worthwhile to learn to recognize each space and when each is useful.

Various phases of the rendering pipeline translate between two spaces, in order to execute the calculation in the most appropriate space. Choosing the right coordinate system can make calculations simpler and computationally cheaper.

Types of Light

In nature, every light is emitted from a 3D surface. There is no such thing as a real-life pixel. In rendering, we use approximations to reduce the computing power needed, but those approximations can limit the fidelity of our rendering. In Unity, we use three different approximations of a real-life light (see Figure 1-6):

- Point light
- Directional light
- Area light (only for baking lightmaps)

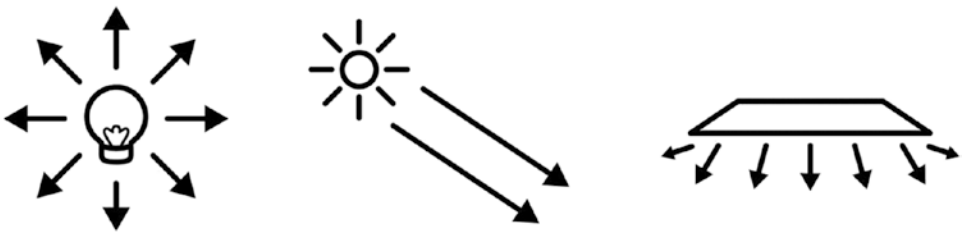

Figure 1-6. *Point light, directional light, and area light*

Point Light

Think of a night lamp, which is a small light that sends rays all around, but those rays don't reach very far. Point lights have a falloff, meaning at a certain distance, the light fades off completely.

Directional Light

Think of the sun; it's so far away from us, that even if it is a point light, all the rays that reach us are parallel. If you were to zoom in on the light of a point light very very near, you would get to a point where the visible rays are all parallel. As a consequence, we're not going to reach the falloff, thus a directional light goes on infinitely.

Area Light

Area light is the best approximation of the three for physical reality, but more expensive computationally. Any real object that emits light will most likely be tridimensional, and therefore have an area. But that complicates the rendering calculations making them more expensive. Unity doesn't have an area light usable for real-time lighting; it can only be used when baking lightmaps.

The Rendering Equation

The calculations that we want to implement in a lighting shader can be represented by the *rendering equation*:

$$L_0(x,\omega_0) = L_e(x,\omega_0) + \int_{\Omega} f(x,\omega_i \rightarrow \omega_0) \cdot L_i(x,\omega_i) \cdot (\omega_i \cdot n) dw$$

Don't panic! It's just an innocuous equation. You don't need to know anything about it at this point. But it will help you to get used to seeing it without shuddering. It's actually a pretty useful way of putting everything you need to calculate lighting together, in one line.

A later chapter covers this equation in more detail. Here, we're going to give an overview of what it represents, meaning the behavior of light on a surface.

The Behavior of Light

Everything that we see, we see because some light has hit that object and it has bounced off of it, in the direction of our eyes. Exactly how and why that bouncing happens is very important to rendering. How much light will bounce off a surface, and in which directions, depends on many factors:

- The angle the light ray is coming from (aka, *reflection*). The more parallel it is, the less it will bounce. See Figure 1-7.

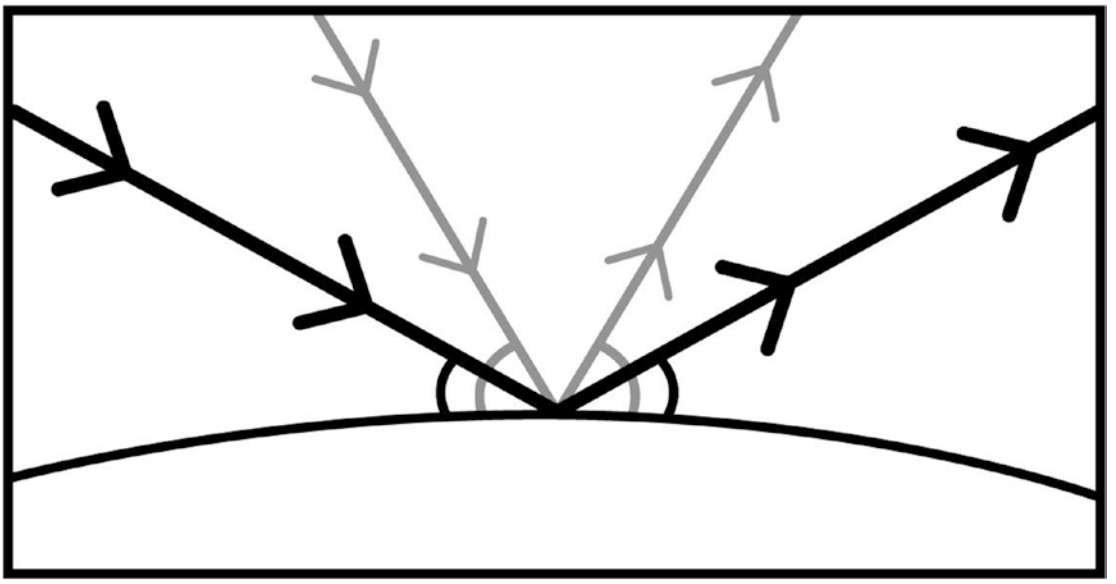

Figure 1-7. *Two different rays come from different directions and bounce away at the same angle they came from*

- The color of the surface (aka, *absorption*). The light spectrum includes all visible colors. A red surface will absorb all other colors in the spectrum, and only reflect the red portion. See Figure 1-8.

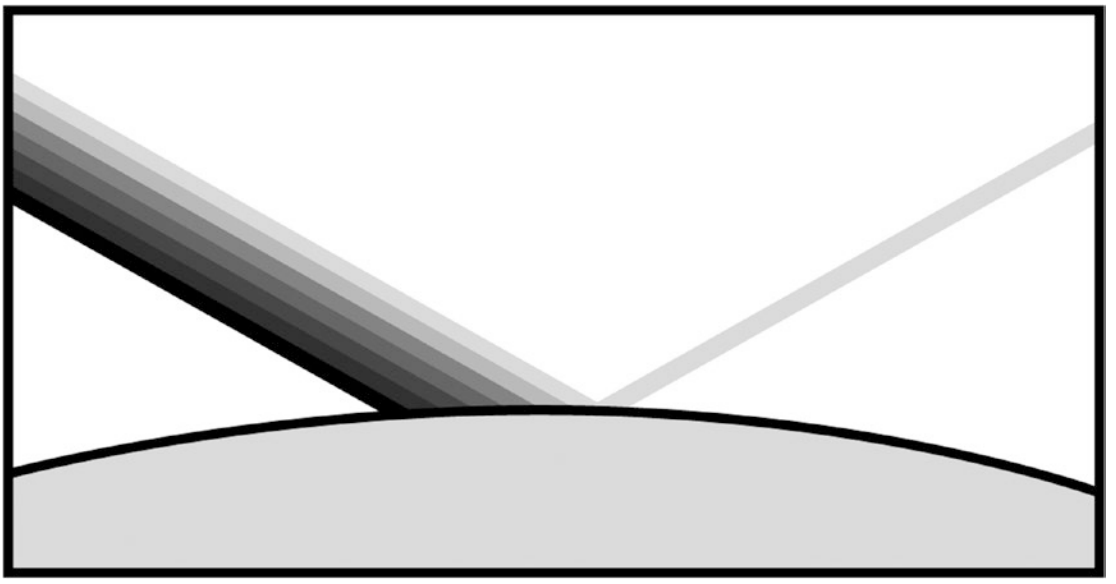

Figure 1-8. *The light spectrum partially absorbed by the surface*

- The smoothness or roughness of the surface. At the microscopic level, surfaces can be rougher than they look, and microfacets can bounce light in different directions. See Figure 1-9.

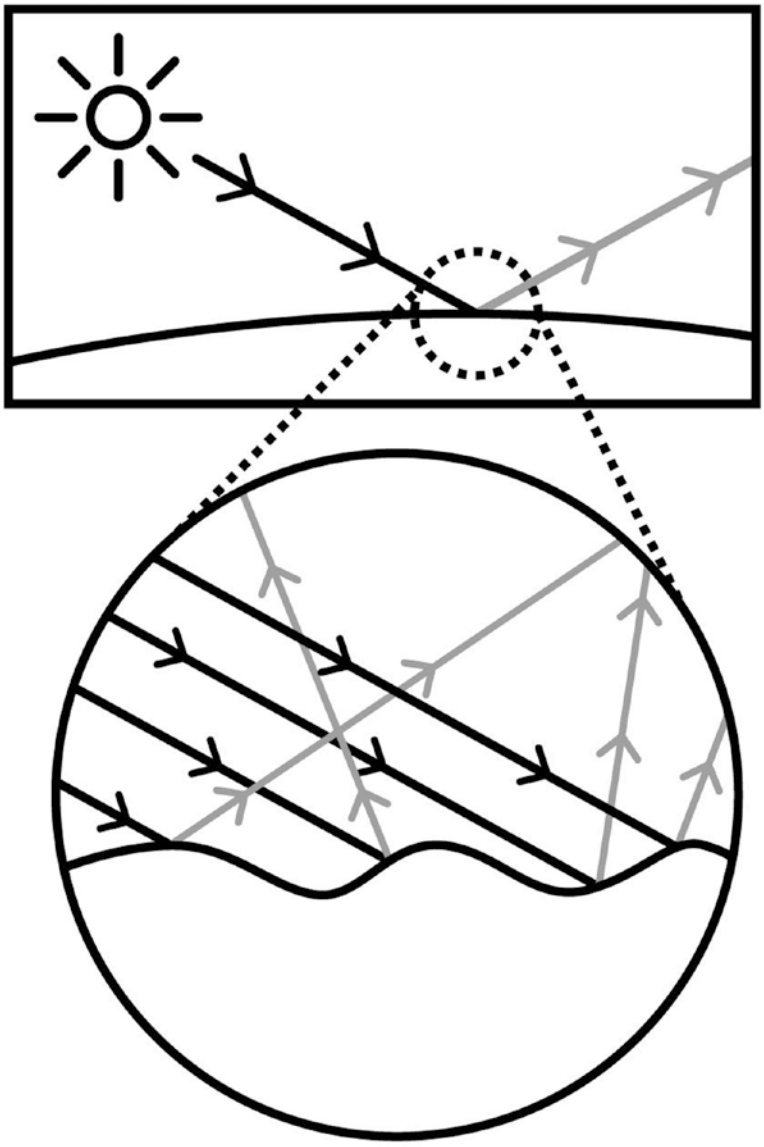

Figure 1-9. *Rays bouncing off microfacets at different angles*

- Semitransparent layers. Think of a puddle. The ground under it looks darker than dry ground, because the water surface is reflecting some light off, thus allowing less light to reach the ground. See Figure 1-10.

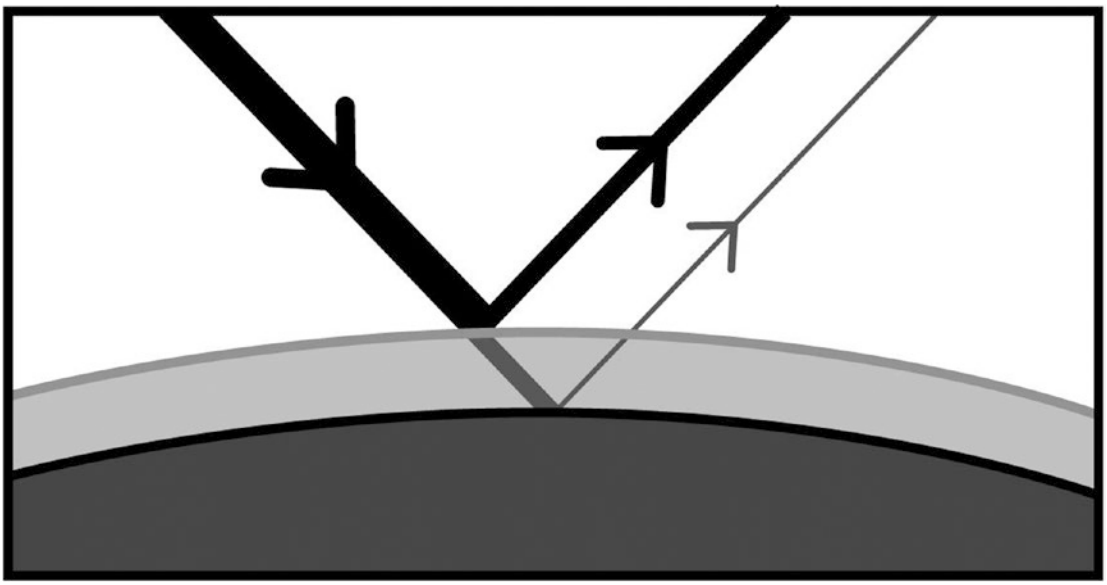

Figure 1-10. *Light rays hitting a puddle and the ground beneath it*

Bounced Light

As you might imagine, the light that is reflected off of one surface often ends up hitting another surface, which again is going to reflect some part of it. Bounced light will keep bouncing until the energy is completely used up. That is what *global illumination* simulates.

The light that hits a surface directly is called *direct light;* the light that hits the object after bouncing off from another surface is called *indirect light*. To make it clearer, Figures 1-11 and 1-12 show the same scene twice. In Figure 1-11, there is only direct illumination, while Figure 1-12 is rendered with indirect light as well. As you can see, the difference is stark.

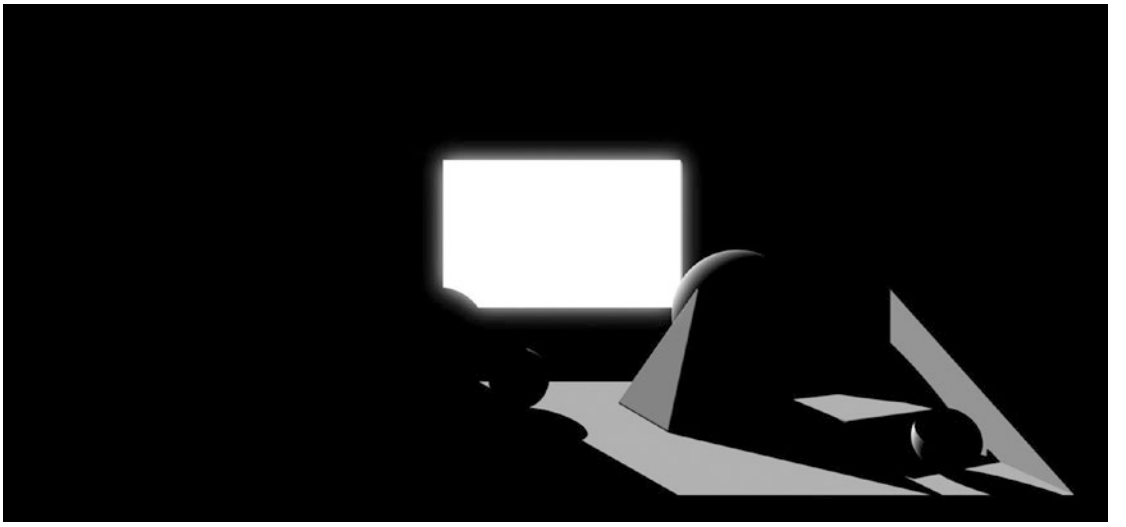

Figure 1-11. *Direct light only*

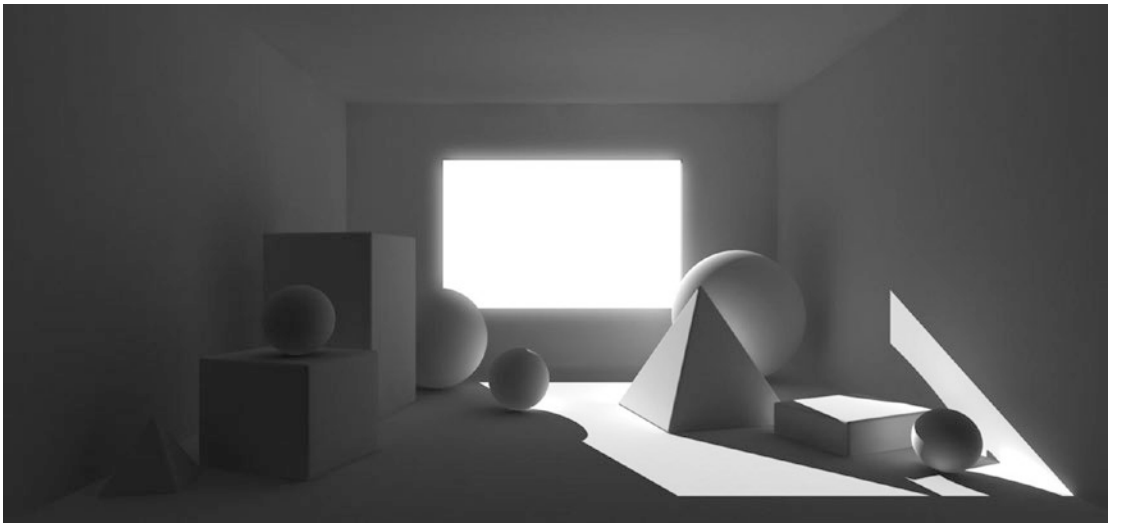

Figure 1-12. *Direct and indirect light*

Before 2010, games generally used a very crude approximation of global illumination, such as *ambient light,* which consisted of only one value for the entire scene. *Spherical harmonics* are also used to approximate **GI**. They are a more faithful approximation than ambient, but more expensive and more mathematically complex as well.

Renderer Types

There are a few types of renderers, and many hybrids between them. Depending on the art direction of a game, certain parts of the scene will take more time to render compared to others. From that, comes the need to change the renderer, in order to optimize the more time-consuming parts.

Forward

This is the first type of real-time renderer. It used to be implemented within the Graphics API (OpenGL or Directx3D). It is the type we've talked about until now. The scene information is fed into it, every triangle is rasterized, and for each light, there is a shading pass.

Deferred

Without global illumination, the only way to render more natural scenes was using a large number of lights. This started to be common practice in the PS3/Xbox360 generation. But, as you know, every additional light in a Forward renderer means an extra shader pass on all pixels.

To achieve better performance, this new type of renderer was invented, which would defer the shading of the scene to the last possible moment. That allows it to ignore the lights that are not reaching the model being shaded at the moment, which makes for much better performance.

Deferred renderers have some problematic spots, such as the impossibility of rendering transparent objects properly. They are also less flexible, because the information passed onto the shading phase has to be decided beforehand, while developing the renderer.

Forward+ (Tiled Forward Shading)

In the PS4/Xbox One generation, various approximations of global illumination are possible, which makes Deferred less attractive. A mix of Forward and Deferred, this renderer type breaks the image into tiles, which are shaded with the Forward method, but only considering the lights that are influencing the current tile. This renderer type is not available in Unity at the moment.

Future Renderers

The industry seems to be going toward developing more flexible renderers that can be better customized for the needs of each game. Unity is already working on a scriptable render loop, which will allow you to write the rendering code itself, while at the moment you can only choose between Forward and Deferred. This new functionality is already available in the current Unity betas.

Shader Visual Graphs

Your previous experiences of shader development might have been through node editors. Many game engines and 3D modeling software programs make shader development available through a visual node interface. Figure 1-13 shows Unreal's Shader Editor as an example.

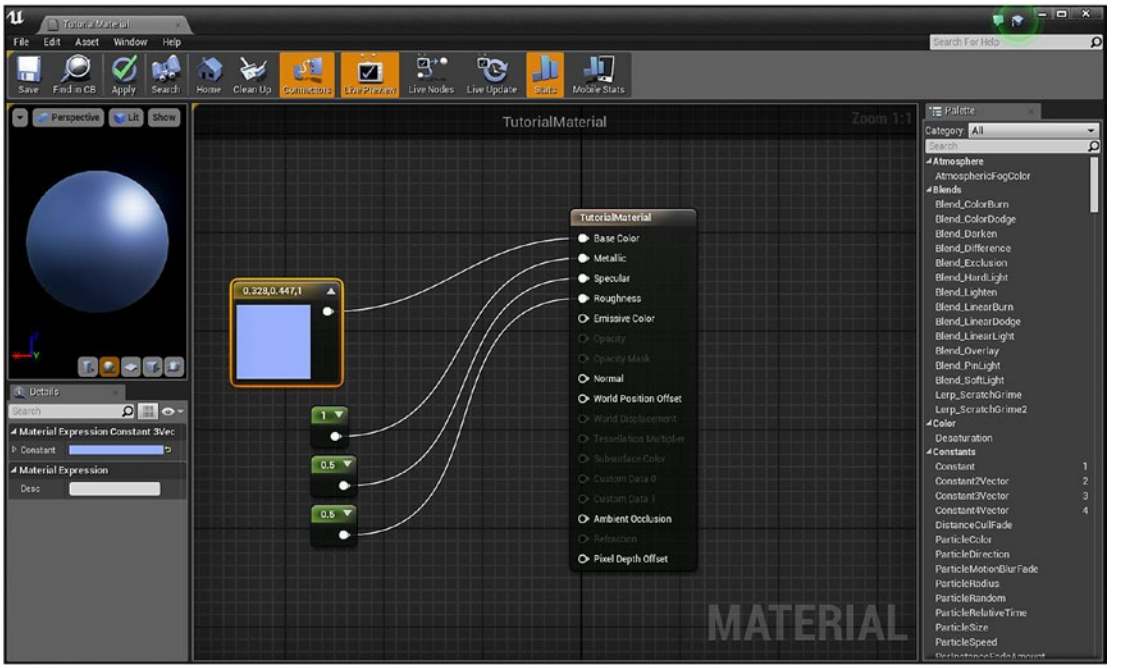

Figure 1-13. *Unreal's Shader Editor*

They are an intuitive way to develop shaders. Unity doesn't include one (although you can download ShaderForge and others from the Asset Store) and, while they're good for many purposes, it would be hard to develop custom lighting shaders with a node editor. Since these tools generate code anyway, and sometimes not good code, it's well worth learning to code shaders by hand. Even if you end up generating your shaders with ShaderForge, by learning to hand code shaders, you'll be able to edit the generated code to improve performance or to fix bugs and compiler errors.

Summary

This chapter introduced shaders to give programmers the ability to further customize the graphics pipeline, which used to be fixed. Graphics programmers would pass the scene data to it and would only be able to give it predetermined parameters. Nowadays, we're going toward more and more customizability, because it gives competent people the possibility of writing faster rendering code.

Shaders are code that runs on the GPU and can simulate lighting for different surfaces. They can also be used for image effects (e.g., Blur), tessellation, and general computation. We're going to stick to lighting and image effects in this book.

Knowing how to write shaders will empower you to control the look of your game much more than you'd be able to otherwise, if you were just using the built-in shaders.

Next

Now that you've been introduced to the fundamentals of shader development, you're going to put them into practice straight away. The next chapter shows you how to create your first Unity shader, while getting familiar with the shader development workflow.

CHAPTER 2

Your First Unity Shader

In the previous chapter, we introduced many of the concepts necessary to develop real-time lighting shaders. In this chapter, I'll get you started with the practical side of shader development in Unity. You'll install Unity (if you haven't done so already) and learn how the shader editing workflow works in Unity. To do that, we'll create a basic project and write a simple shader, so you can put into practice some of the things that you learned in the last chapter.

Introduction to Unity

In order to write shaders, you need a game engine or a renderer. In this book, we're going to use the Unity game engine as the renderer. You can also write shaders for other game engines, graphics APIs, and various 3D modeling software programs like Maya, Blender, etc., but that is out of the scope of this book, though the same principles apply. If you already know how Unity works and have it installed, feel free to skip this section of the chapter.

Set Up

This section lists the steps needed to get Unity up and running on your PC:

1. You need a PC that's running Mac, Windows, or Linux.
2. Download the latest free version of Unity from http://www.unity3d.com/ (Unity 2017.2 at the time of writing; newer versions are not guaranteed to be compatible with this book).
3. Install Unity.
4. You'll need to register an account before being able to use it.
5. Start Unity.
6. Log in to your account.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_2

7. Create a new 3D project (see Figure 2-1).

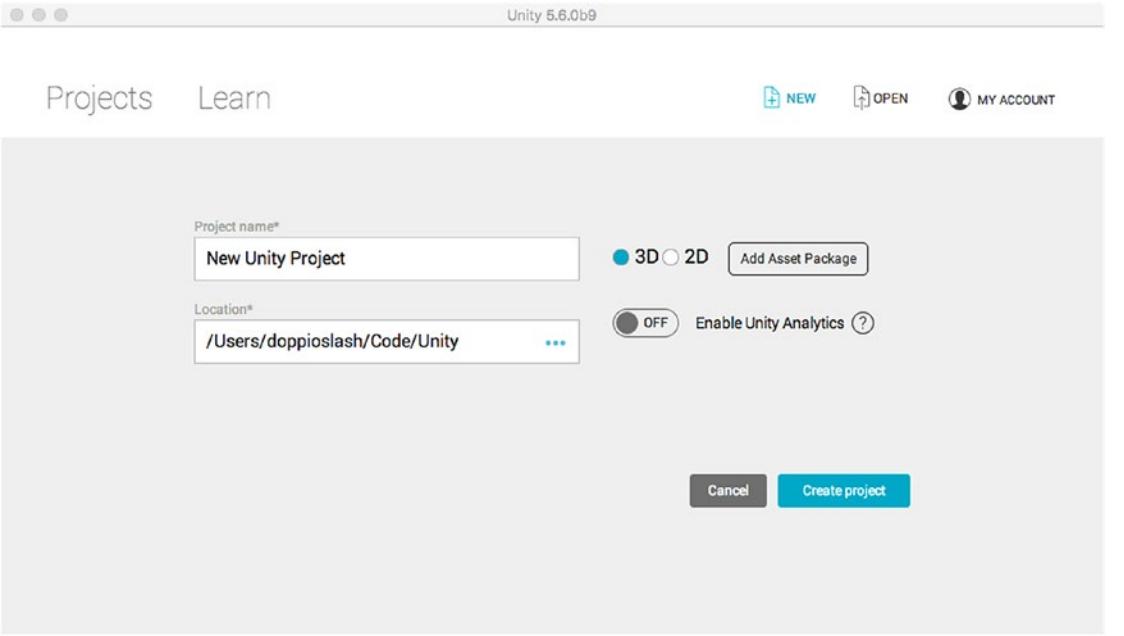

Figure 2-1. *The New Project screen*

That's it; now you have everything you need to get started.

Unity UI

Take a look at the Unity project UI (see Figure 2-2); in it you've got everything you need in one screen:

- *Project panel*: All the files in the project
- *Hierarchy panel*: All the GameObjects in the scene
- *Inspector panel*: Gives you info on files or GameObject components
- *Console panel*: Any compiler or runtime errors will show up in this panel, so make sure it's visible

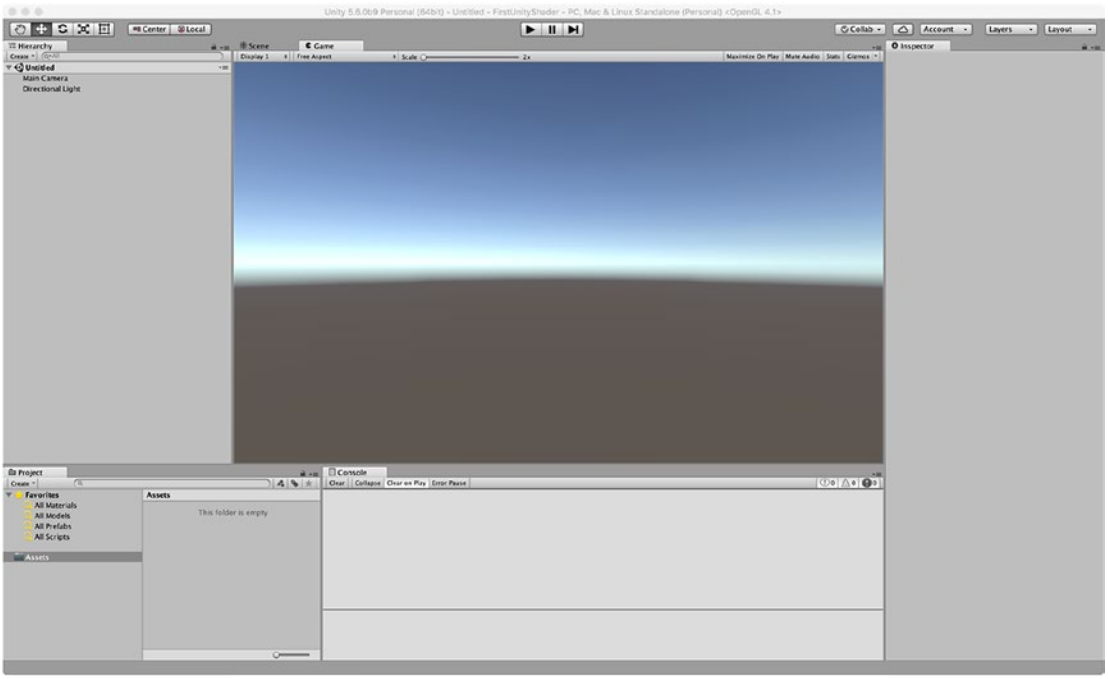

Figure 2-2. *An empty Unity project*

Make Your First Scene

To preview a shader, you first need to set up a scene. You should have an empty scene. Add to it:

- A 3D object (you will apply the shader to it)
- At least one directional light

The simplest way to add a 3D object is to right-click on the Hierarchy panel, then choose 3D Object ➤ Sphere (see Figure 2-3). By doing that, you won't have to worry about 3D models while you're trying to get your head around this first shader.

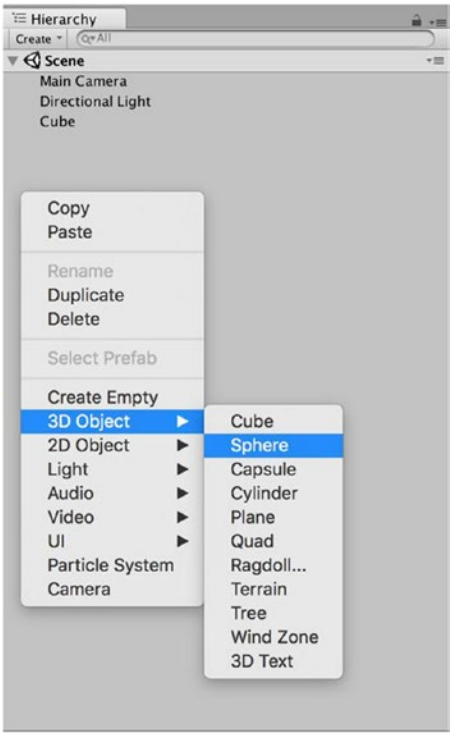

Figure 2-3. *The Create 3D Object menu*

The type of light is actually an important choice from a shading perspective too, not just from an artistic one. In Unity you can only choose between point and directional lights, for real-time, but we'll talk about area lights later and why they are important for physically based rendering. You can add a light by right-clicking within the Hierarchy panel and choosing Light ➤ Directional Light. You can also change the light type by clicking on the light GameObject and changing the light type in the Inspector.

Now we need to create a material. A *material* is a file that contains all the settings relative to a certain kind of surface. That includes which shader will be used to render it, any colors, values, textures to be passed to the shader, and other data. Let's create one.

First create a new directory in the Project panel and call it `Materials`. Then, right-click into the Materials directory and choose Create ➤ Material (see Figure 2-4). Call the new material `RedMaterial`.

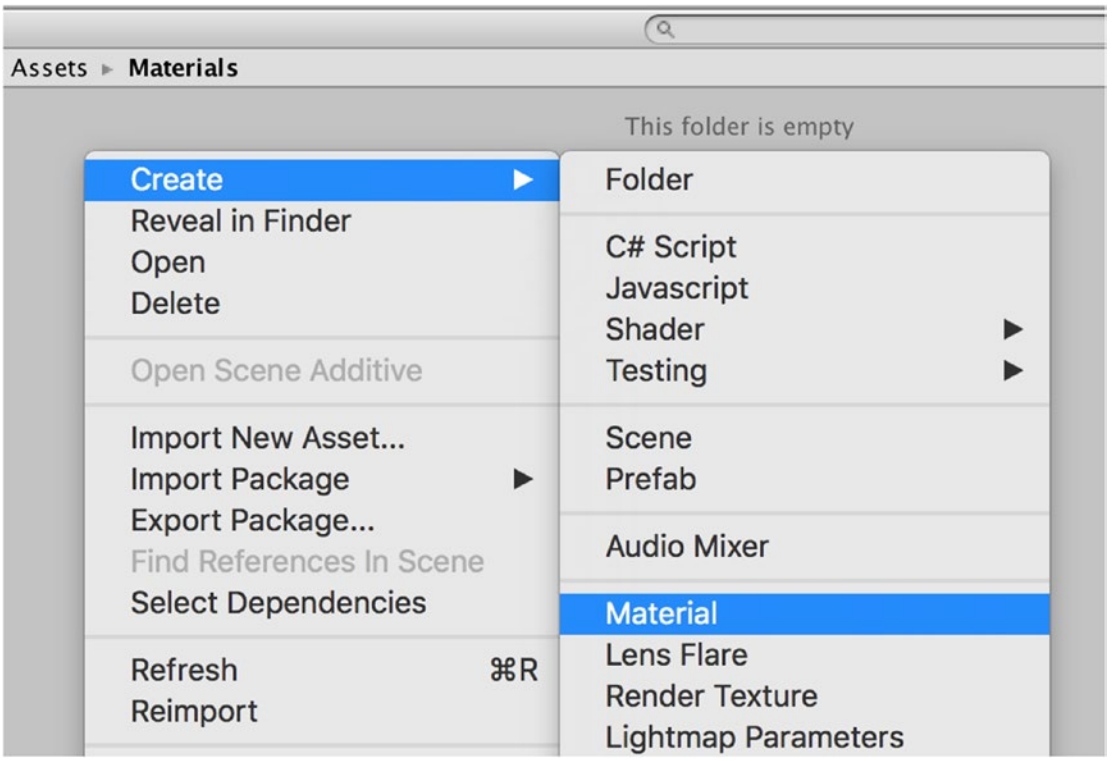

Figure 2-4. *Create a material*

Now you need to create the `Shaders` directory now. Right-click into it and choose Create ➤ Shader ➤ Unlit Shader (see Figure 2-5). Call it `RedShader`. There are a few different types of shaders that you can create, but the Unlit type is the simplest.

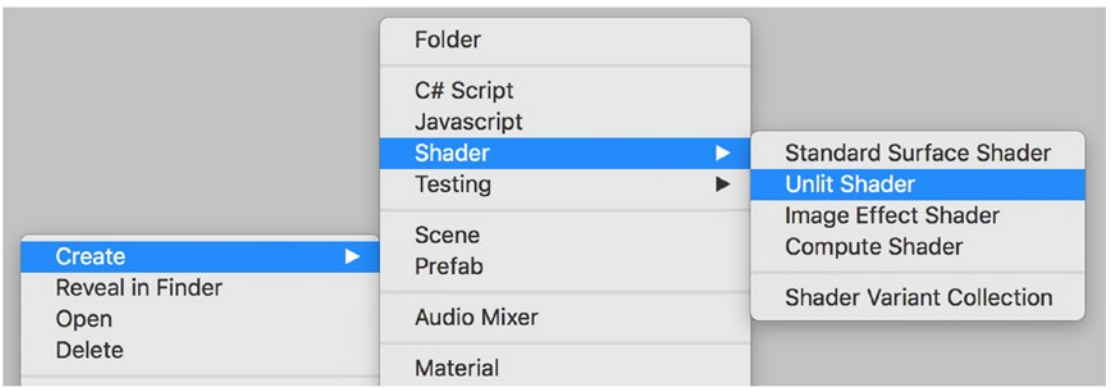

Figure 2-5. *Create an Unlit shader*

Now you need to assign the shader to the material and apply the material to the sphere. The first task is done by clicking on the material file within the Project panel. Now look in the Inspector panel to customize this material. At the very top, you can choose which shader it should use. By default, any new material uses the Unity Standard shader. To find the shader you created, navigate to Unlit ➤ RedShader (see Figure 2-6).

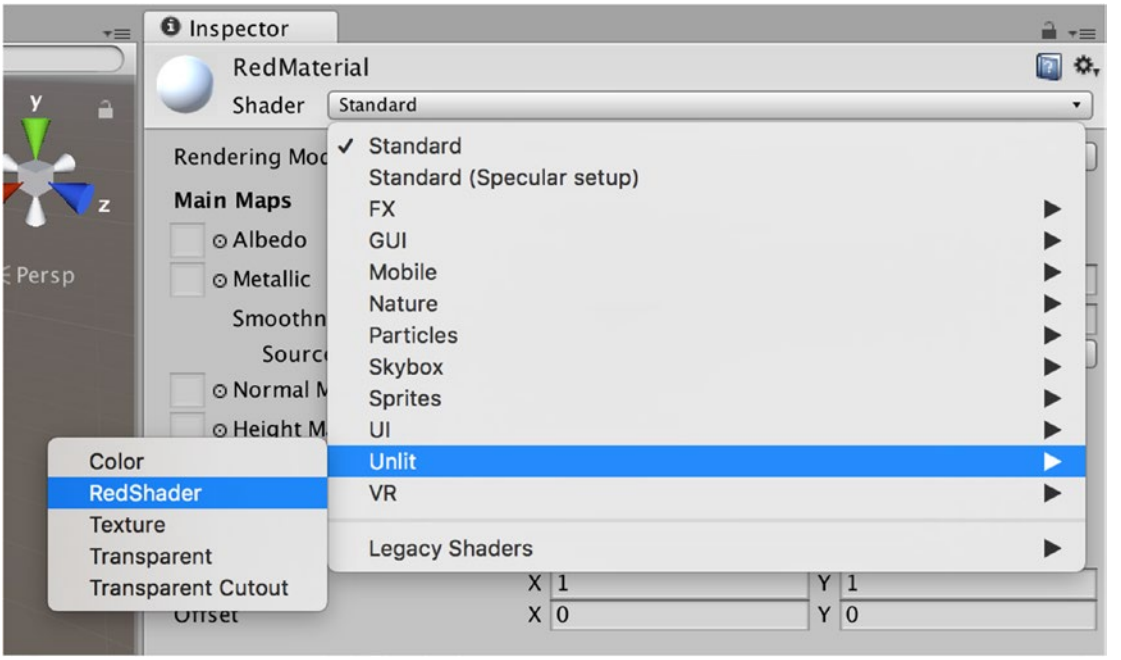

Figure 2-6. *Assign the RedShader to the material you created*

Now drag the material from the Project panel to the Cube GameObject within the Hierarchy panel, or on the cube model within the Scene viewer. You'll see that the cube will change appearance and become a completely white cube without any shading (see Figure 2-7).

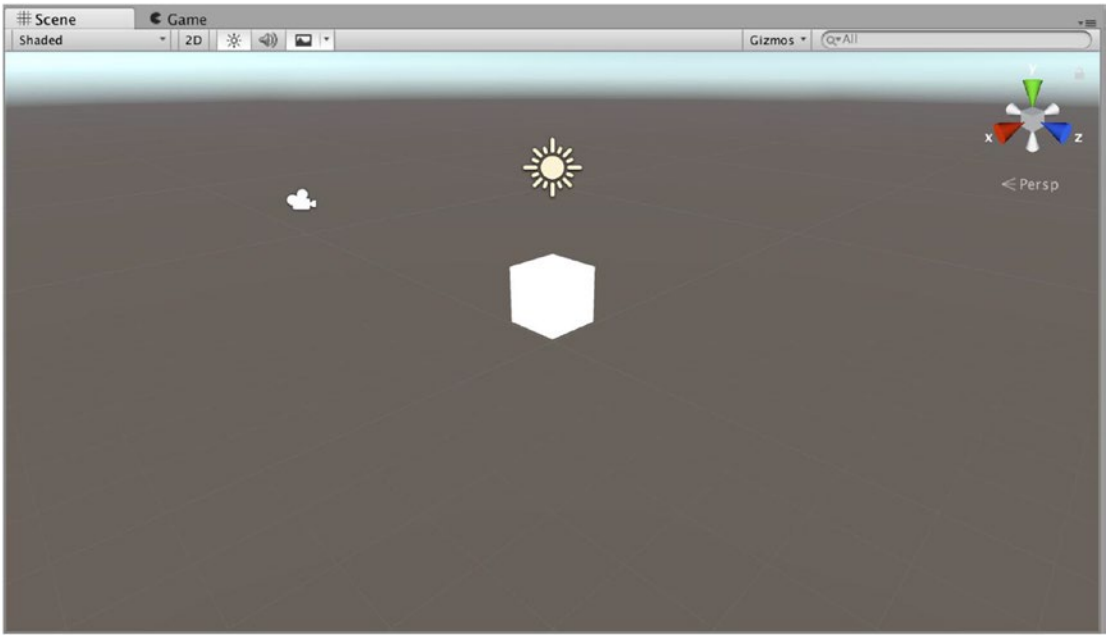

Figure 2-7. *The cube after applying the new shader*

Let's look at the `RedShader` properties. Click on the `Cube` GameObject, and the Inspector will show you the properties for it (see Figure 2-8). Within the Mesh Renderer, notice that there is a Materials list, which has one member, your `RedMaterial`. Below the Mesh Renderer you'll find the Material properties. The assigned shader should be `Unlit/RedShader`; there should be a place to assign a texture, which is empty at the moment. Then you'll see the Tiling, Offset, and RenderQueue options.

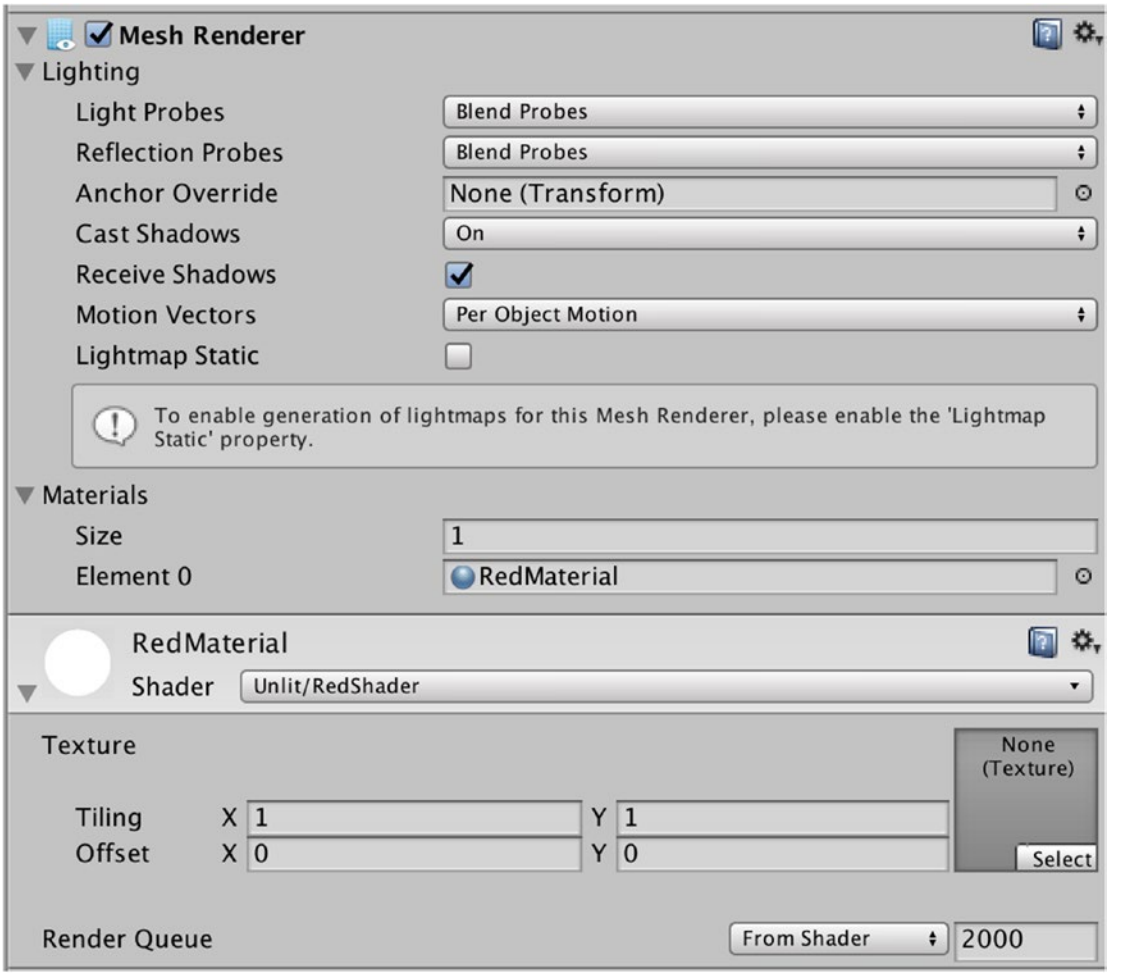

Figure 2-8. *The MeshRenderer and Material inspectors*

Shader Editing

You named it the `RedShader`, but at the moment it's more of a white shader. Let's delve into the code and fix that. Listing 2-1 shows the complete code of the shader, as Unity would have filled it by default. It can be broken down in various parts, which we'll list after the code.

Listing 2-1. Unlit Default Shader

```
Shader "Unlit/RedShader"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
```

```
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                UNITY_FOG_COORDS(1)
                float4 vertex : SV_POSITION;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                UNITY_TRANSFER_FOG(o,o.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                // sample the texture
                fixed4 col = tex2D(_MainTex, i.uv);
                // apply fog
                UNITY_APPLY_FOG(i.fogCoord, col);
                return col;
            }
            ENDCG
        }
    }
}
```

Path and Name of the Shader

```
Shader "Unlit/RedShader"
```

Changing this will change the path you have to go through, within the material, to assign this shader to it. The filename and the path of the shader **can** be different (which can be quite treacherous), so changing the filename won't change the shader path, and vice versa.

Properties

```
Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
```

Each property that shows up in the inspector is declared here, plus some that don't need to be declared. In this case, only one texture is declared, so we can only pass this shader one texture. You can declare as many properties as you want, but if they exceed the capabilities of the target platform, the shader compiler will complain.

Sub-Shaders

```
SubShader
{
```

There can be more than one sub-shader in a shader, and there are a few types of them. When loading the shader, Unity will use the first sub-shader that's supported by the GPU. Each sub-shader contains a list of rendering passes. We'll get back to this in the chapter on image effects.

Tags

```
Tags { "RenderType"="Opaque" }
```

Tags are key/value pairs that can express information, like which rendering queue to use. Transparent and opaque GameObjects are rendered in different rendering queues, which is why the code is specifying "`Opaque`". We'll come back to tags, but we don't need to change them now.

Passes

```
Pass
{
```

Each pass contains information to set up the rendering and the actual shader calculations code. Passes can be executed one by one, separately, from a C# script.

CGPROGRAM (and ENDCG)

```
CGPROGRAM
```

CGPROGRAM and ENDCG mark the beginning and the end of your commands.

Pragma Statements

```
#pragma vertex vert
#pragma fragment frag
// make fog work
#pragma multi_compile_fog
```

These provide a way to set options, like which functions should be used for the vertex and pixel shaders. It's a way to pass information to the shader compiler. Some pragmas can be used to compile different versions of the same shader automatically.

Includes

```
#include "UnityCG.cginc"
```

The "library" files that need to be included to make this shader compile. The shader "library" in Unity is fairly extensive and little documented. Here we're just including the UnityCG.cginc file.

Output and Input Structures

```
struct appdata
{
   float4 vertex : POSITION;
   float2 uv : TEXCOORD0;


v2f
{
    float2 uv : TEXCOORD0;
    UNITY_FOG_COORDS(1)
    float4 vertex : SV_POSITION;
};
```

As covered in Chapter 1, the vertex shader passes information to the fragment shader, through a structure. v2f is that structure in this file. The vertex shader can request specific information through an input structure, which here is appdata.

The words after the semicolons, such as SV_POSITION, are called *semantics*. They tell the compiler what type of information we want to store in that specific member of the structure. The SV_POSITION semantic, when attached to the vertex shader output, means that this member will contain the position of the vertex on the screen.

You'll see other semantic with prefix, SV, which stands for system value. This means they refer to a specific place in the pipeline. This distinction has been added in DirectX version 10; before that, all semantics were predefined.

Variable Declaration

```
sampler2D _MainTex;
float4 _MainTex_ST;
```

Any property defined in the property block needs to be defined again as a variable with the appropriate type in the CGPROGRAM block. Here, the _MainTex property is defined appropriately as a sampler2D and later used in the vertex and fragment functions.

Vertex Function and Fragment Function

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    UNITY_TRANSFER_FOG(o,o.vertex);
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    // sample the texture
    fixed4 col = tex2D(_MainTex, i.uv);
    // apply fog
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

As defined by the pragma statement #pragma vertex name and #pragma fragment name, you can choose any function in the shader to serve as the vertex or fragment shader, but they need to conform to some requirements, which we'll list in later chapters.

Now you're going to become more familiar with editing by making this shader live up to its name of RedShader.

Shader Editing

In order to get used to shader editing, we'll start by making some simple edits and getting rid of code that does not contribute to the final result.

From White to Red

We're going to change the final color of the mesh to be red. Double-click on the shader file, and MonoDevelop (or Visual Studio, depending on your preferences) should open the file for you. It should show the file with syntax coloring, which will make it much easier to read.

Let's think about—what is the bare minimum we can do to make this shader output red—and then we'll clean up the code that will become unused. If you think about the rendering pipeline, which we went through in Chapter 1, you'll realize that if you hardcode the color you want at the end of the fragment function, where it returns `col`, you'll basically overwrite any other calculation done up to then. Listing 2-2 shows you how to do this.

Listing 2-2. On the Left the Default Code, On The Right the Code That Just Returns Red

```
fixed4 frag (v2f i) : SV_Target
{
    // sample the texture
    fixed4 col = tex2D(_MainTex, i.uv);
    // apply fog
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```

```
fixed4 frag (v2f i) : SV_Target
{
return fixed4(1, 0, 0, 1);
}
```

`col` becomes `fixed4(1,0,0,1)`. `fixed4` is a type that contains four decimal numbers with `fixed` precision. `fixed` is less precise than `half`, which is less precise than `float`.

In this case, it doesn't matter which precision we choose, but it will matter if you want to squeeze more performance out of a shader without ruining fidelity. In this position, as the final color outputted by the shader, the first component of the vector is `red`, the second is `green`, the third `blue`, and the fourth the `alpha` value. Keep in mind that the alpha is mostly going to be ignored, unless you're rendering in the `Transparent` queue.

I removed most of the code in the fragment function, because it was no longer relevant. It pays to be tidy and mindful of only leaving code that's actually useful inside a shader. Going still further, we can eliminate anything to do with the fog rendering, since we're hardcoding the final color. The final shader, with all superfluous calculations and options removed, looks like Listing 2-3.

Listing 2-3. The Final RedShader

```
Shader "Unlit/RedShader"
{
    SubShader
    {
        Tags { "RenderType"="Opaque" }

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"
```

```
            struct appdata
            {
                float4 vertex : POSITION;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
            };

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                return fixed4(1,0,0,1);
            }
            ENDCG
        }
    }
}
```

As you can see, anything related to textures and fog has been removed. What remains is responsible for rasterizing the triangles into pixels, by means of first calculating the position of the vertices. The rasterizing part is not visible, as it implemented within the GPU, and it's not programmable.

You might remember that we mentioned many coordinate systems in the previous chapter. Here in the vertex function, there is a translation of the vertex position from Object Space, straight to Clip Space. That means the vertex position has been projected from a 3D coordinate space to a different 3D coordinate space which is more appropriate to the next set of calculations that the data will go through. `UnityObjectToClipPos` is the function that does this translation. You don't need to understand this right now, but you'll encounter coordinate spaces again and again, so it pays to keep noticing them.

The next step (which happens automatically) is that that Clip Space vertex position is passed to the rasterizer functionality of the GPU (which sits between the vertex and fragment shaders). The output of the rasterizer will be interpolated values (pixel position, vertex color, etc.) belonging to a fragment.

This interpolated data, contained within the `v2f` struct, will be passed to the fragment shader. The fragment shader will use it to calculate a final color for each of the fragments.

Adding Properties

Hardcoding values is not a nice habit, so let's transform this red shader into a shader that can be any color we want. To do that, we need to reintroduce the property block. But instead of having a texture property, we want a color property.

The first step is adding this property block back to the file:

```
Properties
{
    _Color ("Color", Color) = (1,0,0,1)
}
```

A property block is made of `_Name ("Description", Type) = default value`. There are many different types of properties, including textures, colors, ranges, and numbers. We'll introduce more of them in the future. For now, let's go on with wiring this property to the shader itself. As it is now, the shader will compile, but the color you pick won't be used.

That's because we haven't declared and used the `_Color` variable yet. First add the declaration somewhere after the `CGPROGRAM` statement:

```
fixed4 _Color;
```

Then change the `return` statement in the fragment function so it's actually using the `_Color` variable:

```
fixed4 frag (v2f i) : SV_Target
{
    return _Color;
}
```

Now you can choose a color from the Material Inspector panel (see Figure 2-9).

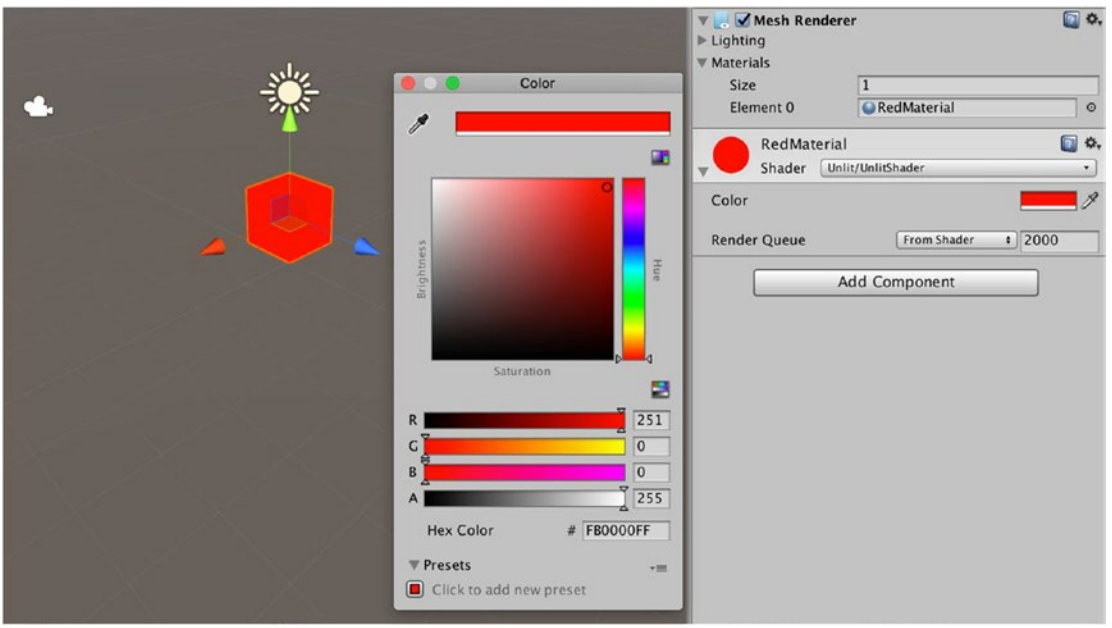

Figure 2-9. *The new color picker made available by the Color property*

Now you know how to wire a property into the shader code. This shader, like all Unity shaders, is mixing two languages. One is **Cg**, a shader language developed by NVIDIA, which is used between the `CGPROGRAM` and `ENDCG` statements. The other is ShaderLab, an extension of Cg developed and used only by Unity, which is everything outside of `CGPROGRAM`. That is the reason you had to declare the `_Color` variable, so the Cg part of the code would be aware of the property living on the ShaderLab side of things.

Now the shader name is not descriptive any more. Let's change it to `MonochromeShader`. To do that you have to change:

- The shader filename to `MonochromeShader`
- The shader path name to `Unlit/MonochromeShader`

You can choose whether to make them match or not, but when you have many shaders, having non-matching names is going to get confusing fast.

Summary

This chapter covered the shader editing workflow in Unity, including how the shader code maps to the rendering pipeline on the GPU. You made a very simple shader, which included both vertex and fragment functions, and you also learned a lot of ShaderLab syntax.

Next

Now that you have some shading coding experience under your belt, the next chapter covers how shaders fit within the graphics pipeline in more detail.

CHAPTER 3

The Graphics Pipeline

This chapter describes every step in the execution of shaders, including how they hook into the graphics pipeline.

By graphics pipeline we intend the steps needed to render a 3D scene into a 2D screen. As mentioned in the first chapter, shader execution is composed of many steps. Some of those steps are implemented completely in hardware in the GPU, and they are fixed, while others are programmable through shaders. This is because the modern GPU has evolved around the graphics pipeline.

3D renderers were first implemented completely in software, which was very flexible, but also very slow. Then 3D acceleration appeared, and more and more of the rendering process was implemented in the hardware, which as a consequence made it much more inflexible. To get some of that flexibility back, parts of the pipeline were made programmable with shaders.

This chapter shows how each part of a shader maps to some part of the graphics pipeline.

Why Learn the Basics of Graphics APIs

You might have heard of OpenGL, Metal, Vulkan, and Direct3D.

They are all graphics APIs, and they have emerged on different platforms, and at different times, but they all share the common objective of providing a set of tools to render a 3D scene without starting from scratch, while taking advantage of hardware acceleration.

While it is certainly possible to write a software renderer from scratch, it hasn't been a viable enough (which mainly means fast enough, in this case) option to ship a game for at least a decade. While writing a software renderer is an enormously educational experience, it's not needed to ship games. Nowadays you don't even need to use graphics APIs directly, since the available game engines such as Unity wrap them for you with an ease that no one who isn't wielding a large team of graphics programmer can have.

In general, you don't want to deal with graphics APIs directly, unless you're developing a game engine. But it's very useful, and sometimes even necessary, in shader development, to know what lies beneath. If you aren't aware of the graphics pipeline and the graphics APIs, you are powerless to optimize your shaders and unable to debug the tricky problems that sometimes arise in shader development.

A General Structure of the Graphics Pipeline

The precise way in which the graphics pipeline is implemented, in some specific GPU, can vary a lot. But the general principles can be distilled and simplified into a version that works for our explanatory purposes.

Keep in mind that, while the principles hold in this example, the stage names and how the stages are divided may vary between different graphics APIs. There are some steps that are always going to be needed, but they may be broken down differently, or called with different names, depending on the documentation you're reading. Figure 3-1 shows the stages of a rendering pipeline (which is another name for graphics pipeline), which are then listed and explained.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_3

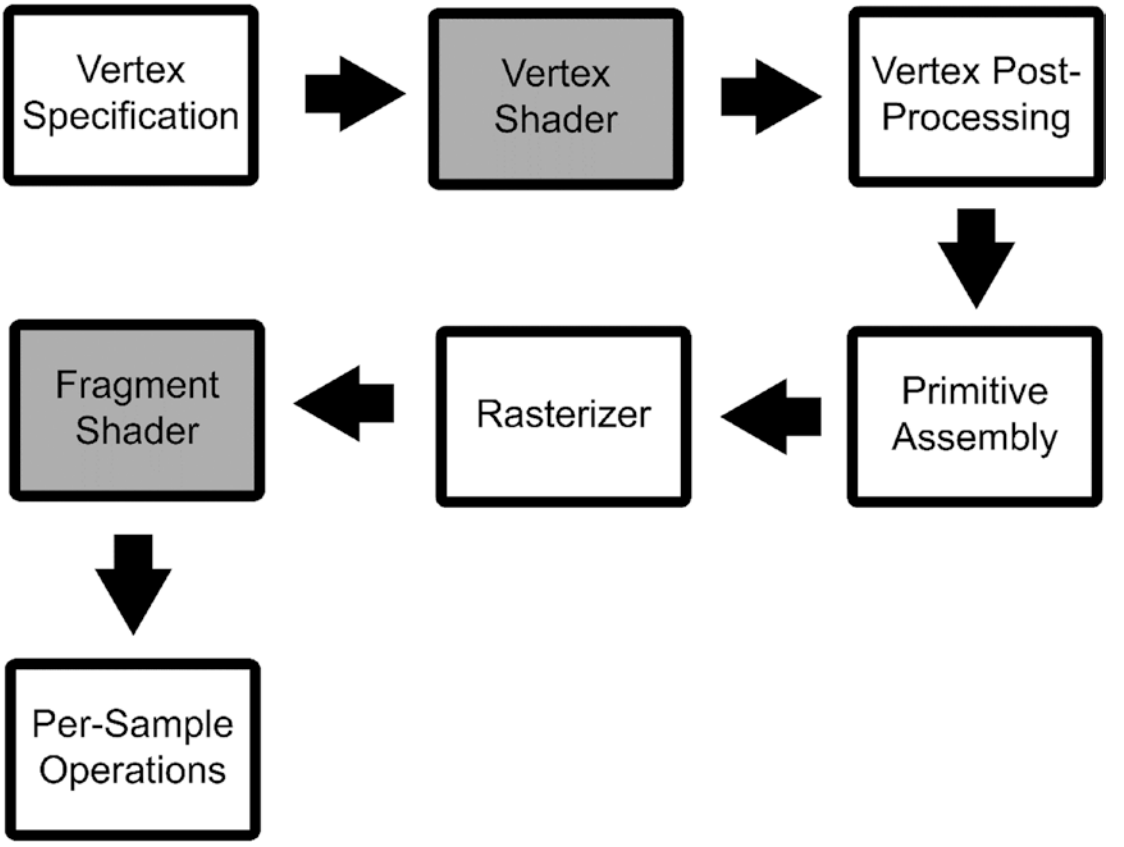

Figure 3-1. *An overview of the graphics pipeline*

The stages of an example graphics pipeline are as follows:

- The input assembly stage gathers data from the scene (meshes, textures, and materials) and organizes it to be used in the pipeline.
- The vertex processing stage gets the vertices and their info from the previous stage and executes the vertex shader on each of them. The main objective of the vertex shader used to be obtaining 2D coordinates out of vertices. In more recent API versions, that is left to a different, later stage.
- The vertex post-processing stage includes transformations between coordinate spaces and the clipping of primitives that are not going to end up on the screen.
- The primitive assembly stage gathers the data output by the vertex processing stages in a primitive and prepares it to be sent to the next stage.
- The rasterizer is not a programmable stage. It takes a triangle (three vertices and their data) and creates potential pixels (fragments) out of it. It also produces an interpolated version of the vertex attributes data for each of the fragments and a depth value.

- The fragment shader stage runs the fragment shader on all the fragments that the rasterizer produces. In order to calculate the color of a pixel, multiple fragments may be necessary (e.g., antialiasing).
- The output merger performs the visibility test that determine whether a fragment will be overwritten by a fragment in front of it. It also does other tests, such as the blending needed for transparency, and more.

This general overview is a mix of many different graphics pipelines, such as OpenGL and Direct3D 11. You might not find the same names in the specific one you want to use, but you'll see very similar patterns.

The Rasterizer

The rasterizer is an important part of rendering pipelines, and one that is normally not much discussed. It's a solved problem, so to speak, and it works quietly without attracting much attention.

It determines which pixels in the final image the triangle covers (see Figure 3-2). It also interpolates the different values that belong to each vertex (such as colors, UVs, or normals) over the pixels the triangle covers.

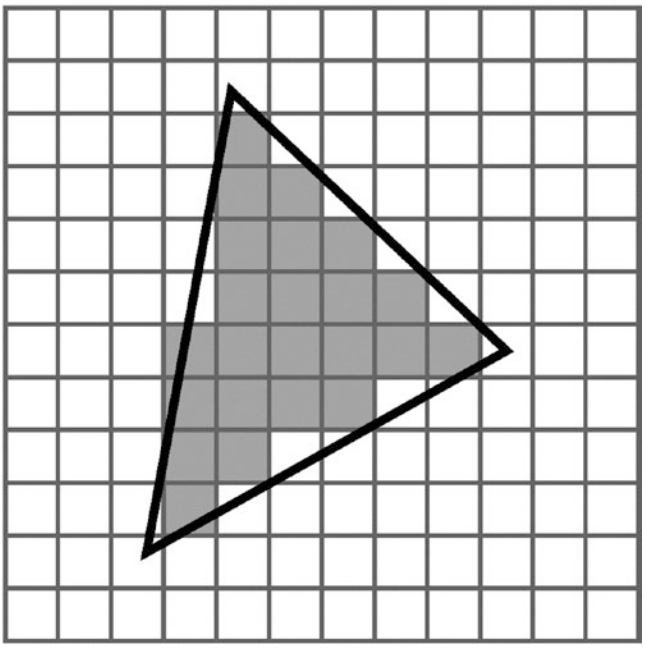

Figure 3-2. *The rasterizer determines which pixels a triangle covers*

The rasterizer is a fairly hidden bit of functionality and is easily overlooked. One of the clearest examples that can be devised to show you how it makes a difference is the interpolation of vertex colors. Imagine that you have a model (a simple quad, or even a triangle, will do) that has different vertex colors attached to its vertices. Figure 3-3 shows how that looks after it passes through the rasterizer.

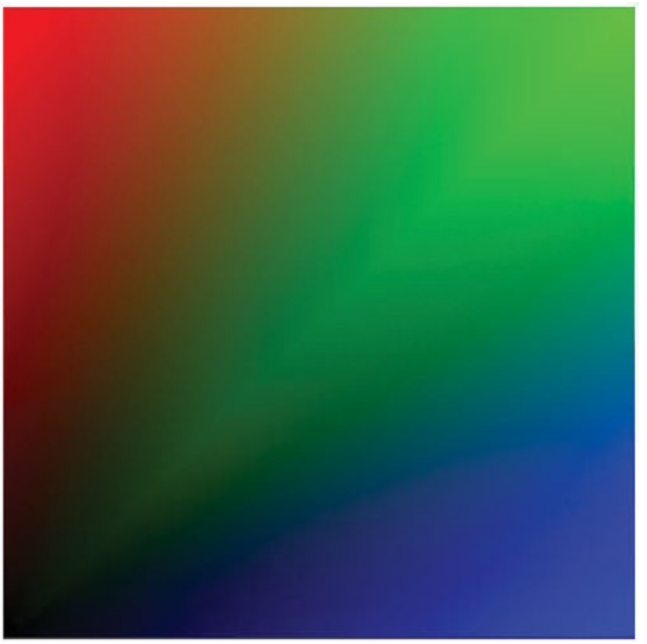

Figure 3-3. *Interpolation of vertex colors by the rasterizer*

Later in this chapter, you'll learn how to write the shader that will render this example.

The Structure of an Unlit Shader

As you saw in the last chapter, an Unlit shader revolves around the two shader functions and the two data structures used to pass information between them. Since their objective is to script parts of the graphics pipeline, they match some steps of the pipeline quite precisely. We're going to detail what they are in this section.

To summarize what you learned about Unlit shaders, Figure 3-4 shows an illustration representing their data flow. The vertex data is gathered into the `appdata` struct, which is passed to the vertex function. The vertex function fills in the members of the `v2f` data structure (which stands for vertex to fragment), which is passed as an argument to the fragment function. The fragment function returns the final color, which is a vertex of the four values—red, green, blue, and alpha.

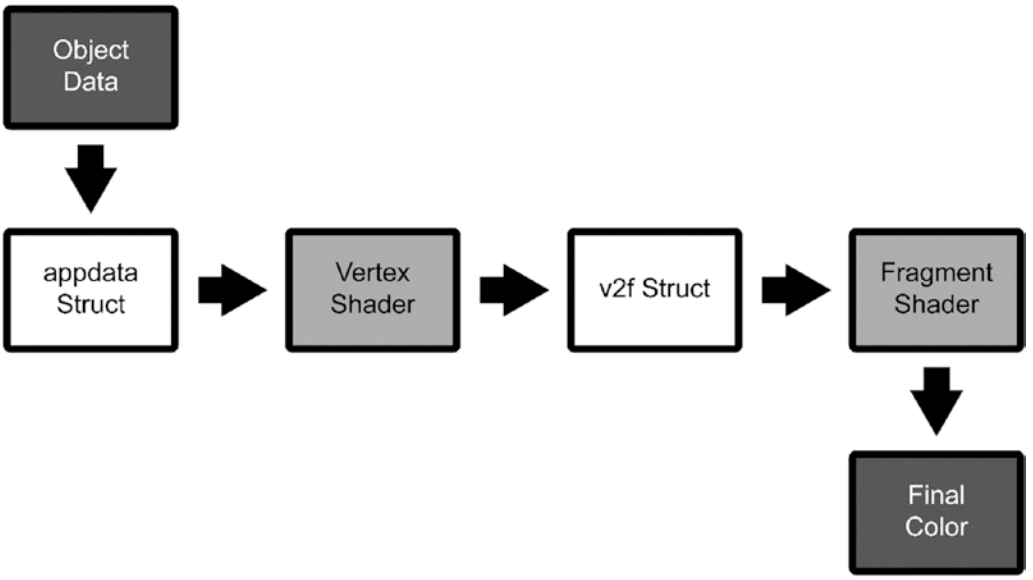

Figure 3-4. *The data flow of an Unlit shader*

In the last chapter, you wrote your first shader. It was very simple; it just rendered a model in a single color, without any lighting. We're going to use that example again to match some parts of the graphics pipeline to shaders, and then we'll also add to it, to better show the usefulness of the rasterizer.

Vertex Data Structure

The first structure was `appdata`, which corresponds to the input assembly stage. To freshen your memory, here is the `appdata` from the last shader:

```
struct appdata
{
    float4 vertex : POSITION;
};
```

`: POSITION` is a *shader semantic,* which is a string attached to a shader data structure that conveys information about the intended use of a member. By adding members with a semantic to `appdata`, we tell the input assembly stage what data we want out of what is available.

In this case, we only asked for the vertex position, which is the bare minimum you're always going to ask for. There is more data you can ask for, such as UV information to deal with texturing, or the vertex colors, should the model have them. The semantic and the data type **must** match. If you were to give it a single float for a `POSITION` semantic, the `float4` value would be truncated silently to `float`.

The data type chooses the "shape" of the variable, but the semantic chooses what goes inside it. Many kinds of semantic can fill a `float4`, but if we were to mistake which semantic we want, the shader would break at runtime, likely in subtle ways. Subtle shader breakage is one of the worst issues to track down once your shaders get complex, so be careful.

This is the only way you can gather data that changes per vertex, but you can pass other data in from properties and global properties that doesn't vary per vertex. OpenGL calls these per-vertex values, quite appropriately, *varying*, and the ones that you can pass globally, or a property in a shader, are called *uniform*.

Vertex Function

The next programmable stage is the vertex processing one, in which the vertex shader function is executed. It takes the `appdata` data structure (`appdata`) as an argument and returns the second type of data structure (`v2f`):

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    return o;
}
```

This is the bare minimum that needs to be done in a vertex shader: transforming the coordinates of the vertex to a coordinate space that can be used by the rasterizer. You do that by using the `UnityObjectToClipPos` function. For now, don't worry too much about coordinate spaces, as we explain them in detail in the next chapter.

Fragment Data Structure

Again, which members we include in the `v2f` data structure decides what data we can pass on from the vertex shader. This is the `v2f` struct from last chapter:

```
struct v2f
{
    float4 vertex : SV_POSITION;
};
```

It's very minimal and it only covers the 2D position of the vertex that was processed. We still need to get the semantic sort of right, but this data structure is less sensitive to mistakes. Keep in mind that semantics must not to be repeated. For example, you can't assign the `SV_POSITION` semantic to a second member as well.

Fragment Function

The next programmable stage is the fragment shading stage, where the fragment shader is executed on each fragment. In the very simple case from Chapter 2, the fragment shader was only using the position from the vertex.

```
float4 frag (v2f i) : SV_Target
{
    return _Color;
}
```

Actually, if you remember, that `float4` wasn't used anywhere in the code. But if you try to get rid of it, you'll find that the shader doesn't render anything at all. That value is used by the graphics pipeline, even if you don't see it reflected in the shader code.

You may notice that the `frag` function has an output semantic, which we didn't mention in the previous chapter. That is used for specific techniques that we are not going to cover in this book. You should stick with `SV_Target`, which means that it outputs one fragment color.

This simple shader shows you that the conversion from 3D scene space and 2D target render space works, because we can indeed render a 3D model into a 2D screen. But it doesn't showcase clearly the role of the rasterizer. Let's add vertex colors support to this shader. For this, you'll need to use a mesh that does have vertex colors. One is included in the example source code for this chapter.

Adding Vertex Colors Support

Basically, we are adding one varying, one extra value attached to the vertex, which will need to be passed to the rasterizer, which will interpolate it.

Appdata Additions

When adding a member to appdata that is supposed to be filled with the vertex colors of the mesh (if the mesh has them), we need to pay attention to the name and semantic. The best bet is to use `color` as the name of the member and `COLOR` as semantic. Using only the `COLOR` semantic with a differently named variable might not work, depending on your platform. Here's how the structure should look after adding this vertex color member:

```
struct appdata
{
    float4 vertex : POSITION;
    float4 color : COLOR;
};
```

v2f Additions

In `v2f`, you need to add the exact same member. As before, this data structure can be less nitpicky, but you change this formula at your own risk. Here's how the structure should look after adding this vertex color member:

```
struct v2f
{
    float4 vertex : SV_POSITION;
    float4 color : COLOR;
};
```

Assign the Color in the Vertex Function

Having prepared the structures with the appropriate members, we add one line to the vertex function. It assigns whatever is in `appdata`'s color member to the `v2f`'s color member. The "magic" is here; it makes the vertex color data go through the rasterizer, which is going to interpolate the colors appropriately:

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.color = v.color;
    return o;
}
```

Use the Color in the Fragment Function

As in the previous shader, the fragment function is only preoccupied with returning one color, but in this case we ignore our property (which we can remove completely) and use the interpolated varying that is contained in `v2f`:

```
fixed4 frag (v2f i) : SV_Target
{
    return i.color;
}
```

Final Result

Figure 3-5 shows the result of our interpolated vertex colors shader.

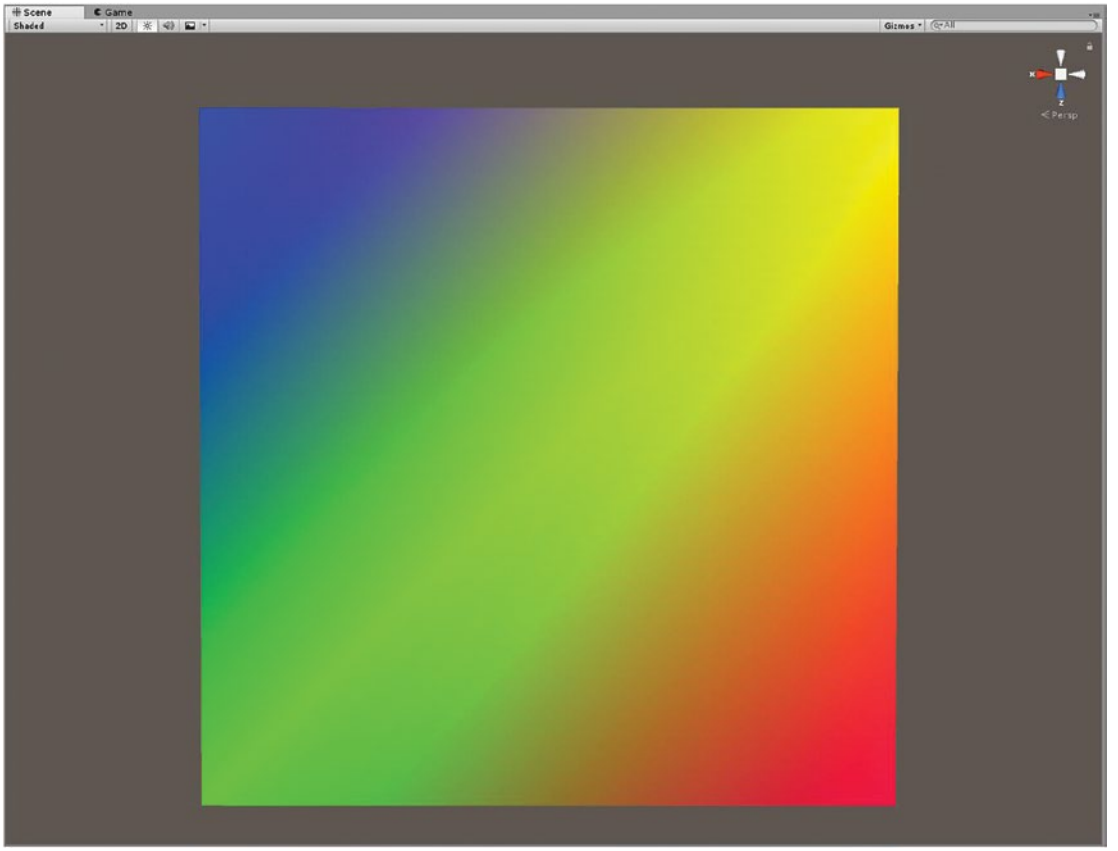

Figure 3-5. *The result of rendering this quad with vertex colors within a Unity scene*

Listing 3-1 shows is the final shader, after removing the `color` property.

Listing 3-1. The Final Shader That Showcases the Rasterizer's Contribution

```
Shader "Custom/RasterizerTestShader"
{
    SubShader
    {
        Tags { "RenderType"="Opaque" }

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float4 color : COLOR;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
                float4 color : COLOR;

            };
            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.color = v.color;
                return o;
            }
            float4 frag (v2f i) : SV_Target
            {
                return i.color;
            }
            ENDCG
        }
    }
}
```

Summary

This chapter explained the stages of the graphics pipeline, specifically which ones you can control through shader code. We demonstrated the otherwise hard to notice and uncelebrated role of the rasterizer with a new shader example.

Next

In the next chapter, we're going to tackle coordinate spaces in detail and connect them to where they are used within the graphics pipeline.

CHAPTER 4

Transforming Coordinate Spaces

In the past chapters, we mentioned transformations between coordinate spaces multiple times, but we deferred a more detailed explanation until this chapter.

As mentioned in the first chapter, you can theoretically choose what coordinate space each of your calculations is going to live in. In the case of real-time rendering, most of the choices have already been made for you by the graphics pipeline. It expects values (vertex positions, normals, etc.) to be in a certain coordinate space, at certain steps of the process.

One example of this from the previous chapter is when the vertex shader outputs the vertex position, it's supposed to be in Clip Space. That's why we set the vertex in the `v2f` data structure to `UnityObjectToClipPos(v.vertex)`.

This chapter goes through each of the spaces commonly used in the graphics pipeline. For each, we'll go through how to transform them and at what pipeline stages they're generally used.

Coordinate Spaces Who's Who

Let's start by introducing each of the coordinate spaces commonly used in real-time rendering. We will consider six: Object Space, World Space, Camera Space, Clip Space, Normalized Device Coordinates, and Screen Space.

This is mostly the order in which they are used within the graphics pipeline. The data passed to the vertex shader is in Object Space, some of it will need to be translated to World Space (to be used for lighting calculations), some of that will go on to being translated to Camera Space, and finally to Clip Space (vertex position), etc.

Object Space

The first coordinate space we're going to talk about is Object Space (see Figure 4-1). Object Space is a 3D coordinate system that has its origin commonly set at the base, or center, of the mesh that is being sent to the rendering pipeline in the input assembly stage. The origin is likely the pivot point in the 3D modeling software that was used to model the mesh.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_4

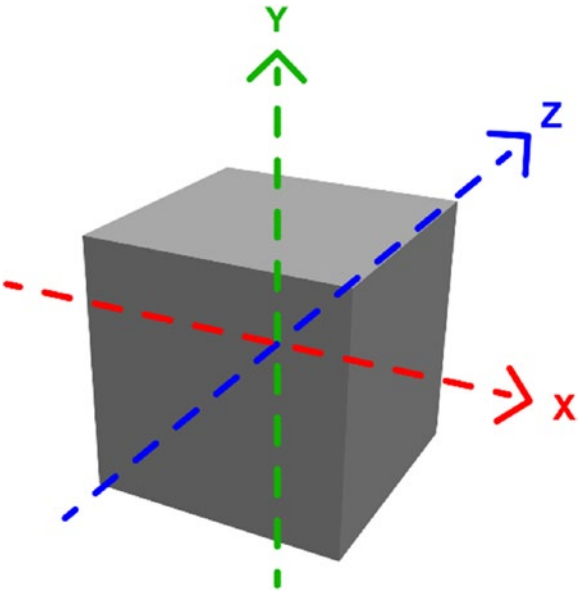

Figure 4-1. *Mesh in Object Space; note where the origin is placed*

It can also be called Local Space or Model Space. However, in Autodesk Maya, Local Space is used to refer to a different coordinate system, so beware of possible confusion there. When the vertex position is stored into `appdata`, in order to be passed to the vertex shader, it is in Object Space.

World Space

Moving on, next up is World Space. In World Space, the frame of reference is not the single mesh, but the entire scene (see Figure 4-2).

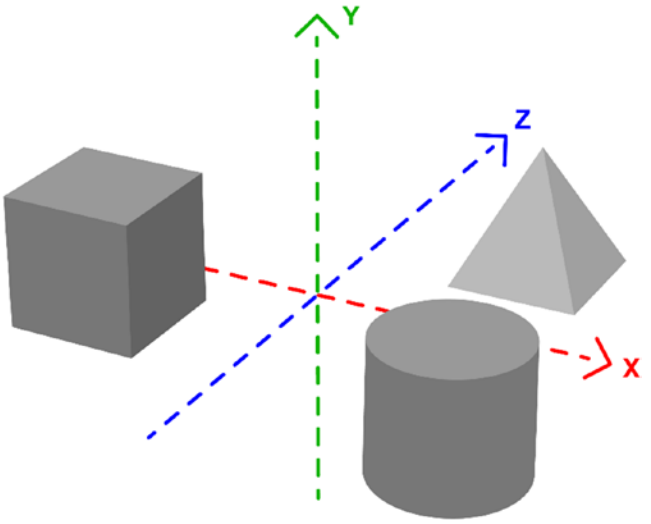

Figure 4-2. *A scene in World Space*

Where exactly the origin point falls in World Space depends on how you put the scene together. In Unity, this is the space in which your Unity scene lives, and the coordinates used in your GameObjects transforms should be in this coordinate space.

Transformation Between Spaces

To transform between different coordinate spaces, *matrix multiplications* are normally used. One convenient feature of matrices is that they can be composed. So, one matrix can get us all the way from Object Space to Clip Space.

Unity includes many built-in functions that implement the most commonly used transformations for some types of values, such as position, normals, light directions, etc. Here are some that will obtain a certain value in World Space:

- `float3 UnityObjectToWorldDir( in float3 dir )` takes a direction in Object Space and transforms it to a direction in World Space
- `float3 UnityObjectToWorldNormal( in float3 norm )` takes a normal in Object Space and transforms it to a normal in World Space; useful for lighting calculations
- `float3 UnityWorldSpaceViewDir( in float3 worldPos )` takes a vertex position in World Space and returns the view direction in World Space; useful for lighting calculations
- `float3 UnityWorldSpaceLightDir( in float3 worldPos )` takes a vertex position in World Space and returns the light direction in World Space; useful for lighting calculations

These functions hide the implementation details, and you only need to worry about passing them the right value and using them in the appropriate place.

As explained earlier, transforming from a coordinate space to another is done by using specific matrices. If you recoil when hearing the word "matrix," using these utility functions you'll be able to achieve a lot, without worrying about matrices too much.

Thanks to them, you mostly don't need to handle these transformations with matrices any more, even though that is still happening under the hood. Many functions that you can use to transform coordinates between spaces can be found in `UnityShaderVariables.cginc`, `UnityShaderUtilities.cginc`, and `UnityCG.cginc`.

You can get more hands on and use matrices directly, setting up the matrix multiplication yourself. In this use case, Unity includes many built-in matrices you can use for space transformations, but they have different names compared to the same options in OpenGL, etc.

Here are some of the built-in Unity matrices for transformation from and to Object Space:

- `unity_ObjectToWorld`, which is a matrix that transforms **from** Object Space to World Space
- `unity_WorldToObject`, the inverse of the above, is a matrix that transforms from World Space to Object Space

As an example, let's translate the vertex position from Object Space to World Space:

```
float4 vertexWorld = mul(unity_ObjectToWorld, v.vertex);
```

We're basically passing to the `mul` function our matrix and the value that we want transformed. You can also make up your own matrix, if none of the built-in ones will do what you need.

Camera Space

Next is Camera Space, also called Eye Space or View Space (see Figure 4-3). This coordinate space contains the same scene as the World Space coordinate system, but from the point of view of the camera that you are rendering from.

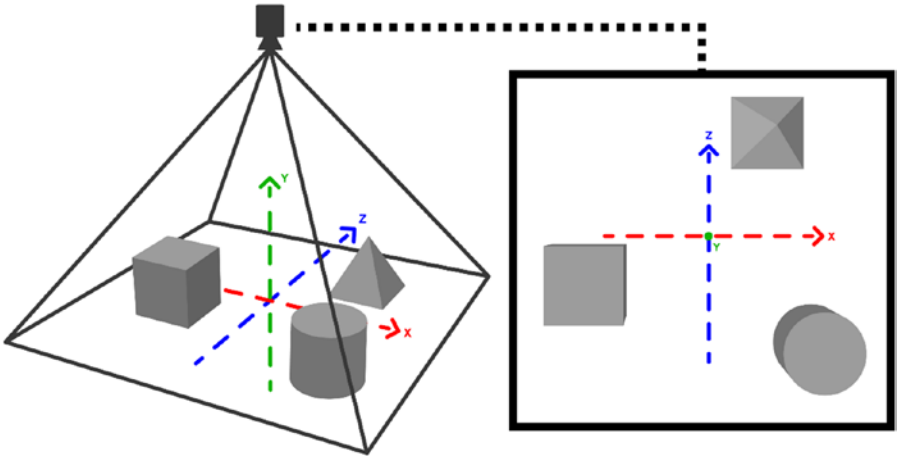

Figure 4-3. *Scene in View Space*

Camera Space is needed, because it's a necessary step to get to outputting Clip Space, but that is mostly taken care by the standard shader infrastructure.

There are a couple of built-in matrices for Camera Space:

- `unity_WorldToCamera`, which transforms from World Space to Camera Space
- `unity_CameraToWorld`, the inverse of the above, transforms from Camera Space to World Space

There is also one built-in function:

- `float4 UnityViewToClipPos( in float3 pos )` transforms a position from View Space to Clip Space

Clip Space

The next coordinate space we're going to talk about is Clip Space (see Figure 4-4).

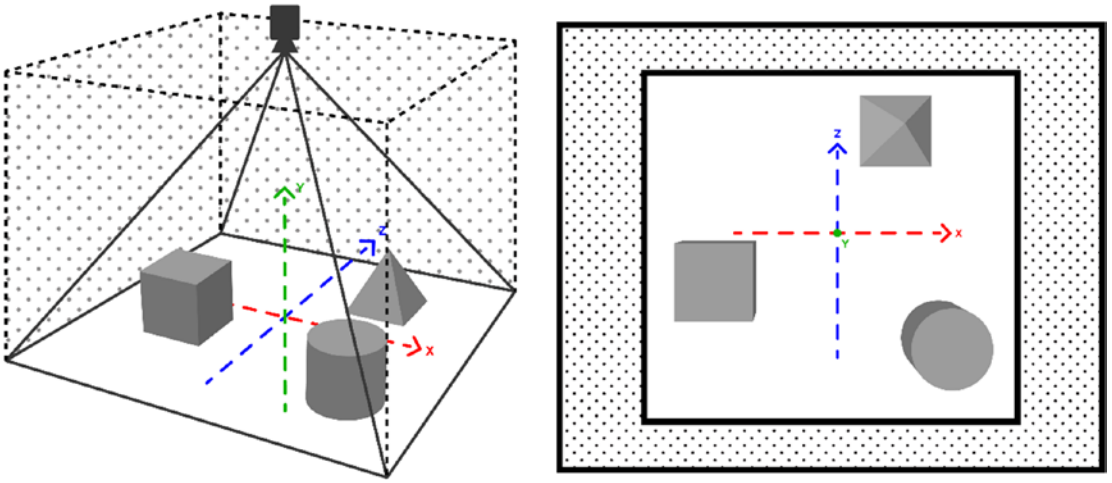

Figure 4-4. *The scene in Clip Space*

The vertex post-processing stage of the rendering pipeline includes *clipping*. Clipping removes any part of the primitives that are not within the boundaries of Clip Space. Clip Space coordinates range from -1 to 1.

There's one unintuitive bit of business going on here. You would expect the coordinate spaces we have listed up to now, Clip Space included, to have three coordinates—one for the x axis, one for the y axis, and one for the z axis. But in OpenGL (and other APIs), the 3D spaces we listed don't use three coordinates; they use four: (x, y, z, and **w**).

Where does this **w** coordinate come from, and why is it used? It turns out that there is an issue with using Cartesian spaces for 3D rendering: two parallel lines cannot meet each other, which makes it impossible to represent perspective. This additional coordinate is necessary for the workaround to this problem, which is called *homogeneous coordinates*. We keep around this additional coordinate **w**, and then, at the opportune moment (going from Clip Space to Normalized Device Coordinates), we divide all the others by it. Doing this, we can represent perspective.

So, Object, World, View, and Clip Space represent 3D spaces using these four coordinates. In all but Clip Space, w is 1. Then, w is changed with a number different from 1 by the matrix used to transform from View to Clip Space. That matrix is called the *projection* matrix in OpenGL. Then the **w** coordinate is used to determine whether a vertex should be clipped or not.

To set up the projection matrix, you need to use the information from the viewing volume (aka, the *frustrum*). The frustrum varies with what type of projection you want to use; common ones are perspective projection and orthographic projection.

In perspective projection (see Figure 4-5), the frustrum is composed of a near and a far plane, where the near plane is smaller than the far one, because objects located farther from the camera will appear smaller in perspective. The field of view defines the proportion between the near and far plane. Changing it changes how much of the scene we're going to render.

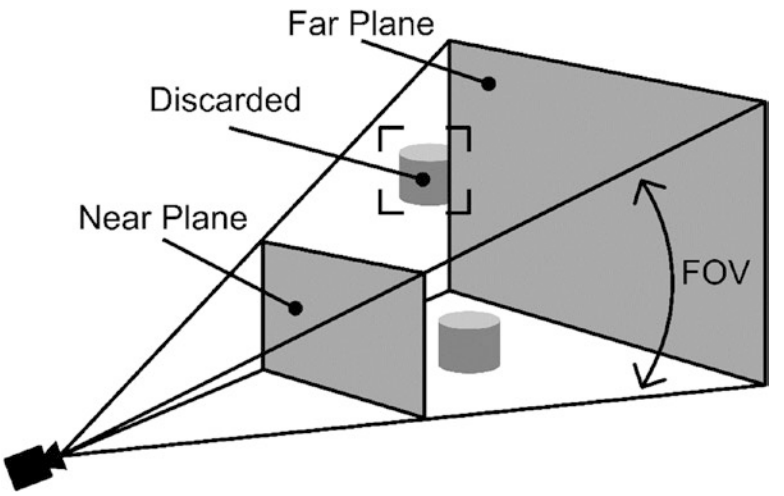

Figure 4-5. *A frustrum used for perspective projection*

There are no built-in matrices for Clip Space, but there are some built-in functions for it:

- `float4 UnityWorldToClipPos( in float3 pos )`, which transforms a position from World Space to Clip Space
- `float4 UnityViewToClipPos( in float3 pos )`, which transforms a position from View Space to Clip Space
- `float4 UnityObjectToClipPos(in float3 pos)`, which transforms a vertex position from Object Space to Clip Space

Normalized Device Coordinates

Next up are the *Normalized Device Coordinates* (NDC). This is a 2D space that is independent of the specific screen or image resolution. Coordinates in NDC are obtained by diving Clip coordinates by **w**, a process called *perspective division*. Again, NDC coordinates range from -1 to 1 in OpenGL. NDC uses three numbers instead of two, as you'd expect it to, but in this case the z coordinate is used for the depth buffer, rather than being a homogeneous coordinate.

Screen Space

Finally, Screen Space (see Figure 4-6) is the coordinate space of the 2D render target. That may be a screen buffer, or a separate render target, or an image. It is obtained by transforming and scaling NDC into viewport resolution. Finally, these screen coordinates of the vertices are passed to the *rasterizer*, which will use them to produce fragments.

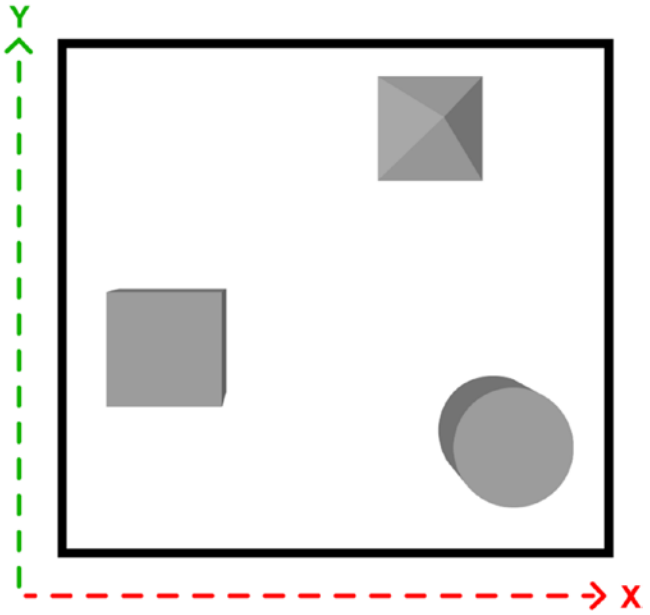

Figure 4-6. *The scene in Screen Space*

Underneath Built-In Functions

The book has mentioned many built-in functions so far, and now we look at how one of them works in more detail. From the last chapter's example shader:

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.color = v.color;
    return o;
}
```

`UnityObjectToClipPos` is a function that stands for `mul(UNITY_MATRIX_MVP,*)` which is a matrix multiplication, going from Object Space to Clip Space. Matrices are commonly combined by multiplying them by each other, so `MVP` stands for Model Matrix * View Matrix * Projection Matrix. In other words, we're fast-forwarding from Object Space through World and View Space to get to Clip Space coordinates using just this one line.

In the next chapter, you see more examples of useful applications of transforming values between coordinate spaces.

Where to Find the Shader "Standard Library" Code

What I mean by standard library is a ZIP file that you can download from the Unity web site that contains all the shader includes and the shader code in Unity. You can also find that code inside your Unity install, but its position may be subject to change.

To get this code, go to the Unity web site and find the link to download older versions of Unity. That will give you a list of Unity versions and different download options. What you're looking for is the "Built In Shaders" option (see Figure 4-7). That will download a ZIP file with all the includes mentioned up to now, and more.

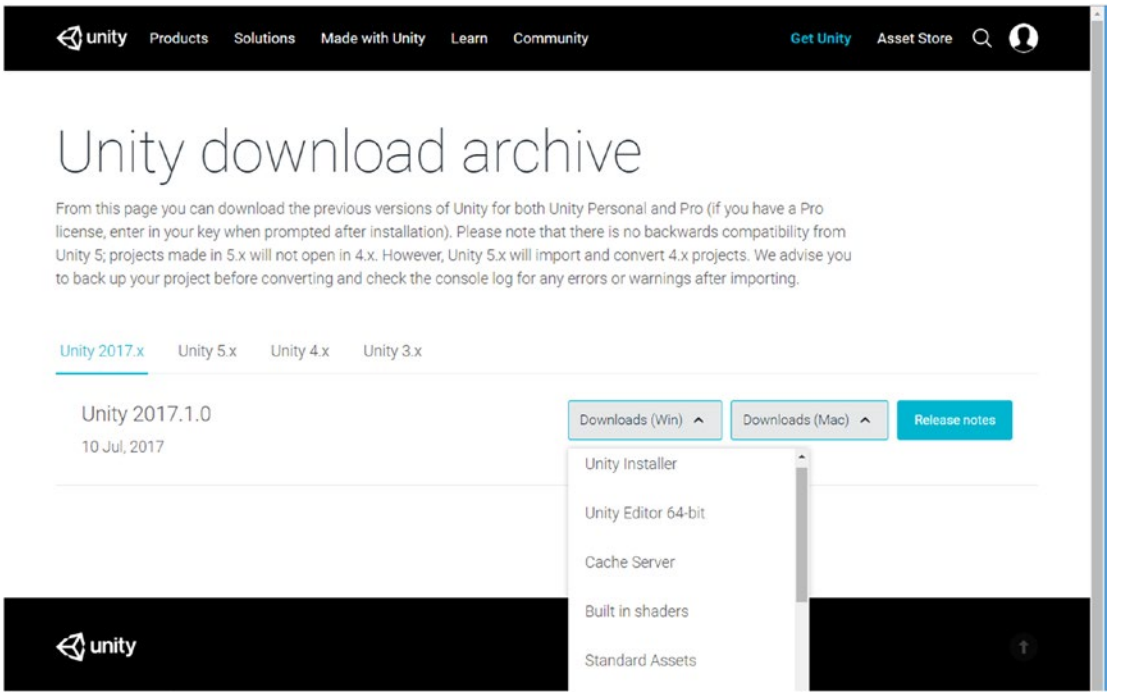

Figure 4-7. *Downloading the Unity shader source code*

Summary

This chapter presented the various coordinate spaces commonly used in the graphics pipeline and in shading, and discussed ways to transform between them. It also showed which functions are already included in Unity to convert between them.

Next

The next chapter begins the journey into lighting shaders, starting from the basics.

CHAPTER 5

Your First Unity Lighting Shader

In the previous chapters, we introduced the shader editing workflow, got started with the simplest possible shader, and explained the graphics pipeline and the role of coordinate space transformations in it.

This chapter gets started on the main topic of this book: *lighting*. There is an infinite variety of shaders that you can write, without ever needing to worry about lighting. But the heart of this book is lighting calculations and how they help making your game look great and distinctive. To get started, we'll take the `MonochromeShader` shader and extend it with some very approximated lighting calculations.

This type of approximated lighting was used before the advent of *physically based rendering*. It's cheaper, simpler, but also uglier. You might recognize it as what used to be the typical "video game look". That said, it'll introduce you to lighting through short and simple shaders, and will also help you understand how shading has evolved with time.

Lighting Shaders

A *lighting shader* includes all the calculations needed to simulate light hitting a surface. While most lighting shaders are based, at least loosely, on the rendering equation, a few years ago there wasn't enough computational power in GPUs to do much more than very loose approximations of it. Until about 2010, when the physically based model started to trickle down from the movie rendering community, these were the terms we would use to refer to different parts of the lighting calculations:

- *Diffuse* is the subset of a surface with irregular microfacets, that reflects light in many different directions
- *Specular* is the subset of the surface that has aligned microfacets, and reflects light in a few, similar directions
- *Ambient* is the minimum light intensity in the scene, so places where the direct light doesn't reach won't end up just black

These explanations are in terms of microfacet theory, which is something we can do now, since physically based rendering introduced it in real-time shading. But before that, diffuse and specular were quite hard to pin down precisely. That's because in physical reality there is no such clean division between specular and diffuse. The variability in directions of the microfacets of a surface controls how smooth or rough that surface looks. A rougher surface will look more diffuse; a smoother surface will look more specular. Diffuse to specular is a continuum, not a binary choice.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_5

What Is an Approximation

An *approximation* is a value or quantity that is nearly, but not exactly, correct. In our case, an approximation means an alternative way of calculating some parts of the rendering equation. Approximations can be less or more computationally intensive. As of yet, it's impossible to calculate the rendering equation in real-time, due to the presence of an integral in it:

$$L_0(x,\omega_0) = L_e(x,\omega_0) + \int_\Omega f(x,\omega_i \rightarrow \omega_0) \cdot L_i(x,\omega_i) \cdot (\omega_i \cdot n) dw$$

meaning that the group of calculations after the $\int_\Omega$ is going to be repeated for each direction w in the hemisphere above the point on the surface called x. As you may suspect, each direction means a lot of directions. Solving this integral in real-time has so far eluded us, although we're starting to see things like somewhat-real-time raytracing being implemented on GPUs.

Using a cheaper approximation may mean that your game can run at 60 fps, but it will also mean that you lost some potential rendering fidelity. Not all games focus on fidelity. Most of the time, graphics need only be good enough. But in this book we focus on fidelity, as that's the yardstick by which physically based rendering measures results.

Diffuse Approximation

Let's try to define diffuse in more practical terms. Think of light hitting a surface, and then being reflected in any possible direction (see Figure 5-1). Each reflection direction is equally probable.

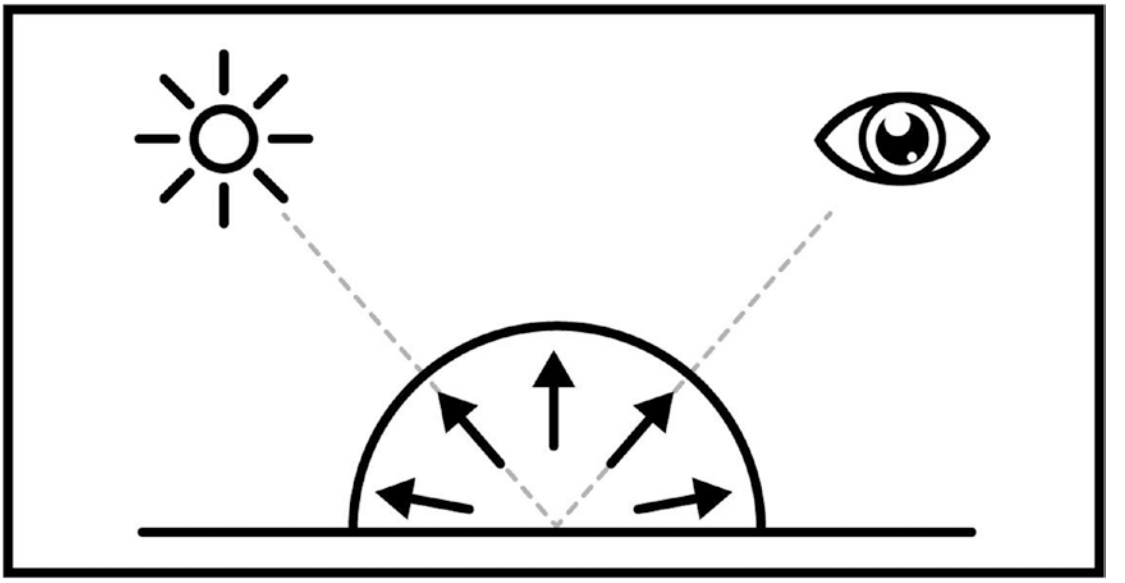

Figure 5-1. *Light being reflected off a surface in a diffused way*

This approximation only concerns itself with the direction, color, and intensity of the light. It doesn't take into account the nature of the surface at the microscopic level. The objective is to determine how much light is hitting, and being reflected off, each pixel of the rendered model, without delving into a more detailed simulation. Imagine it as a function—you pass it the direction, color, and intensity of the light, as well as the unlit color of the surface, and it will give you the color of the final image at that point.

It's possible to implement it in the vertex shader, which is computationally cheaper, since usually there are fewer vertices than pixels. This works because the rasterizer will interpolate the values, but as a side effect that creates pretty visible artifacts.

Specular Approximation

Think of a light hitting a surface and being reflected in just a few directions (see Figure 5-2). At this level of approximation, specular light often appears as an almost white small circle. Again, this approximation is concerned only with the direction, color and intensity of the light.

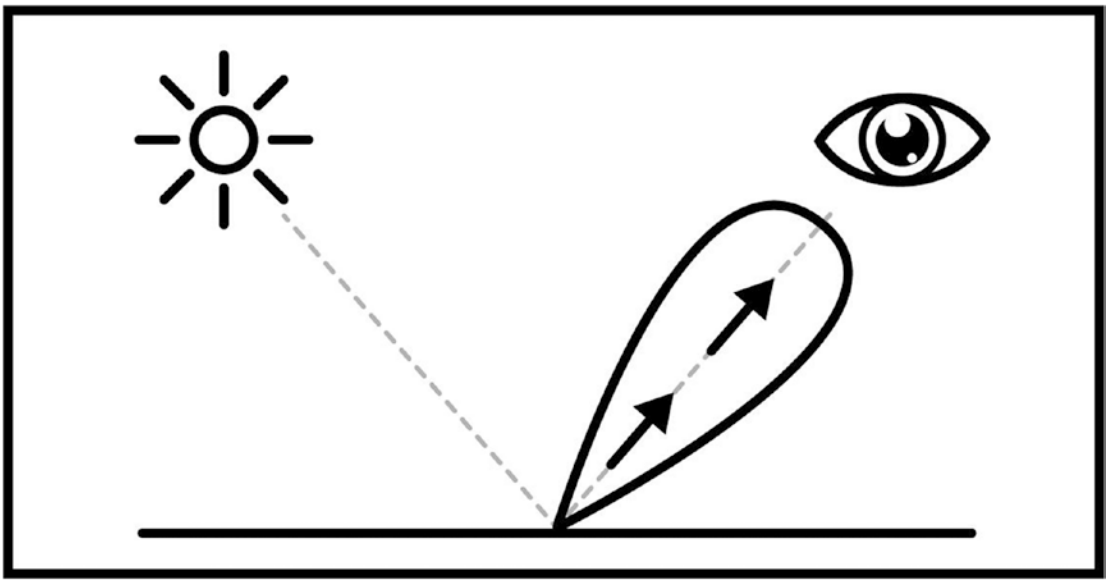

Figure 5-2. *Light being reflected off a surface in a specular way*

The fact that light is being reflected only in a few directions makes specular *view-dependent,* meaning that if your point of view moves, the specular term will change. Specular is trickier to simulate in lightmaps, because you need to bake the direction, and other information, in the lightmap as well.

Diffuse and Specular Combined

Both diffuse and specular terms will show up in the same surface, most of the time. Metals have a lower diffuse component and a much higher specular one, but they still have a diffuse component. When you combine them, you have the typical graph commonly used to represent lighting calculations, as shown in Figure 5-3.

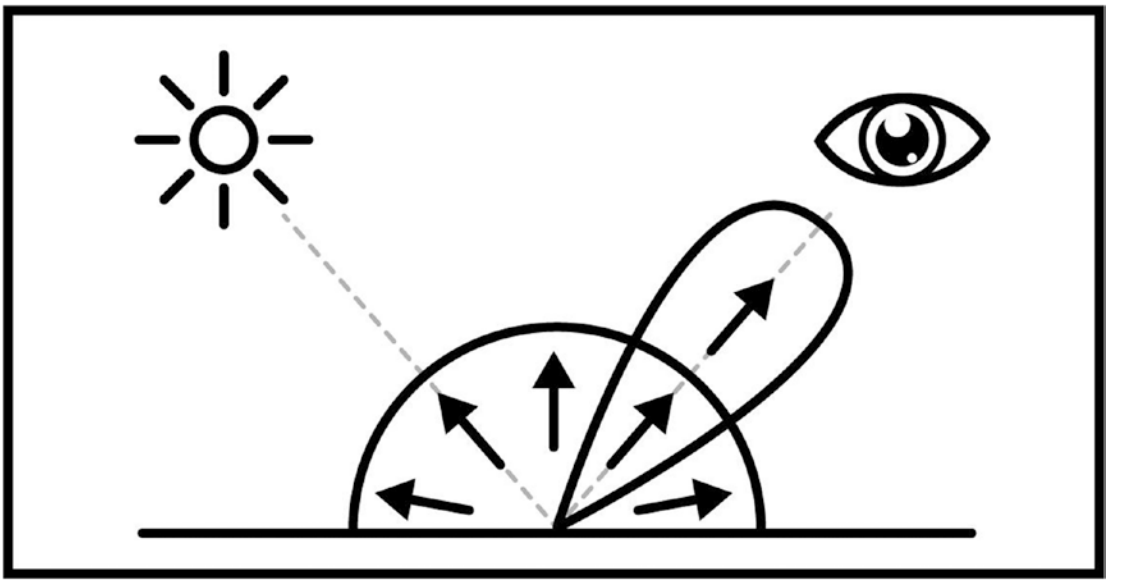

Figure 5-3. *Graph of how light is reflected off a surface, with a specular and a diffuse lobe*

Calculating Basic Lighting

Now that we've defined specular and diffuse approximations, let's attempt to implement them.

Diffuse

If you remember, back in Chapter 1, we talked about how the angle at which the light ray hits the surface is important. That angle is called the *angle of incidence,* and the bigger it is, the less light the surface will receive from the ray. For angles bigger than 90 degrees, it won't receive any light at all (see Figure 5-4).

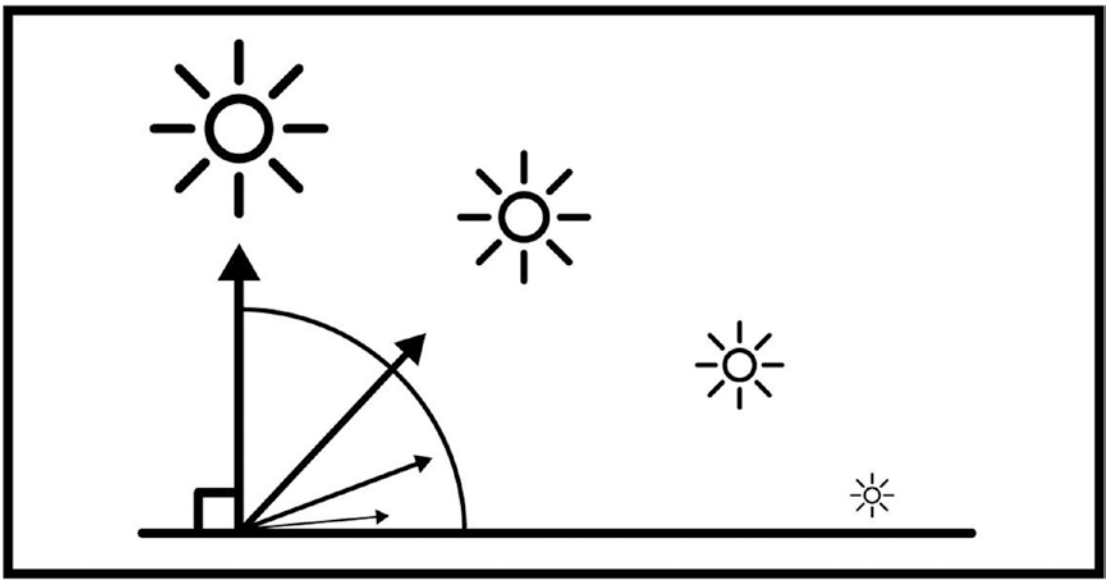

Figure 5-4. *Angle of incidence and its effect on the amount of reflected light*

To calculate the amount of brightness/light a surface will receive, we can use the cosine of the angle of incidence.

To calculate lighting in our shader, we need to calculate that angle from the surface normal direction and the light direction. Those two are vectors, which means they are groups of more than one numerical value. Vectors in shader languages can typically contain two, three, or four elements. The operation we need here is called a *dot product.* Both cosine and dot product are functions included in every shader language. Translated to code, this concept is shown in Listing 5-1.

Listing 5-1. Two Ways of Calculating Brightness (Magnitudes of Normal and LightDir Are 1)

```
float brightness = cos( angle_of_incidence ) // brightness from the angle of incidence
float brightness = dot( normal, lightDir )  // brightness calculated from the Normal and
                                                Light directions

float3 pixelColor = brightness * lightColor * surfaceColor // final value of the surface color
```

The result from this operation, multiplied by the color of the light and the color of the surface, will give you a crude but effective approximation of lighting. This basic diffuse is also known as *Lambert*, or *Lambertian Reflectance:*

$$LightDir \bullet Normal \times Color \times Intensity_{Light}$$

Meaning that the dot product of the light direction, with the normal direction, multiplied by the light color and intensity, will give you the color of the pixel at that point.

Now that we've introduced the theory and implementation of a Diffuse term, we can put this into practice by adding them to our cube scene.

Your First Lighting Unity Shader

Let's extend the monochrome shader with a Diffuse term.

Implementing a Diffuse Term

Let's create a new material, called `DiffuseMaterial`. Duplicate the `MonochromeShader` and call the duplicate `DiffuseShader`. Remember to change the shader path to `Custom/DiffuseShader`, or you will have two overlapping shader path names.

We need to make quite a few changes to the old shader. First, we need to change the tags section to this:

```
Tags { "LightMode" = "ForwardBase" }
```

This means that this pass will be used for the first light pass of the forward renderer. If we only have one `ForwardBase` pass, any lights after the first one won't contribute to the final result. If we want them to, we need to add another pass and set its tags to the following:

```
Tags { "LightMode" = "ForwardAdd" }
```

We're not going to worry about the second pass for now. Moving on, we need to add another `include` to the appropriate place in the file:

```
#include "UnityLightingCommon.cginc"
```

`UnityLightingCommon.cginc` is a file that contains many useful variables and functions that can be used in lighting shaders. With this, the chores are over, so now we're going to get into the meat of the implementation.

First, keep in mind that the Normal and Light directions need to be in one of the coordinate spaces covered in Chapter 4. Thinking about it, we shouldn't use Object Space, because the light is outside the model we're rendering. The appropriate space to use for these lighting calculations is World Space.

First, we need to get the Normal information from the renderer; therefore, we need to add a slot for that normal to `appdata`, the data structure that contains the information we ask from the renderer. This is how `appdata` should look:

```
struct appdata
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
```

Notice that we're also telling it that we want a normal by adding the `NORMAL` semantic to the declaration; otherwise, there'd be no way for the renderer to understand what we want.

This vertex function will need to calculate the Normal direction in World Space. Fortunately, there is a handy function called `UnityObjectToWorldNormal` (mentioned in Chapter 4) that takes the Object Space Normal direction we just passed to the vertex shader through `appdata` and translates it to World Space.

```
v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
```

```
    float3 worldNormal = UnityObjectToWorldNormal(v.normal); //calculate world normal
    o.worldNormal = worldNormal; //assign to output data structure
    return o;
}
```

Then we need to assign it to the output structure. To do that, we need to add that slot, using the `TEXCOORD0` semantic to tell it to use a slot that fits a vector of three or four values.

```
struct v2f
{
    float4 vertex : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
};
```

Now we can use that information to calculate our Lambert diffuse. We can get the light color from the variable `_LightColor0`, which comes from the extra include file, and the World Space light position of the first light in the scene from the variable `_WorldSpaceLightPos0`.

When you want to access a subset of a vector, you can add **r**, **g**, **b**, **a**, or **x**, **y**, **z**, **w** to it, after a dot. That means accessing the numbers contained by the vector directly, somewhat like you would access the value at an index of a c array with `[0]`, `[1]`, etc. This is called a *swizzle operator,* and it can do more things than array indexing, like rearranging the values themselves by changing the order of the letters.

In the fragment shader, we need to first normalize the `worldNormal`, as the result of a transform may not be a vector of magnitude 1. Then we calculate the dot product of normal and light direction, taking care not to let it become negative. That's what the `max` function does.

```
float4 frag (v2f i) : SV_Target
{
    float3 normalDirection = normalize(i.worldNormal);

    float nl = max(0.0, dot(normalDirection, _WorldSpaceLightPos0.xyz));
    float4 diffuseTerm = nl * _Color * tex * _LightColor0;

    return diffuseTerm;
}
```

Finally, we multiply that dot product with the color of the surface and the color of the light. This is the end; your shader should be complete. For your convenience, Listing 5-2 shows all the code for this shader.

Listing 5-2. Our Diffuse Shader So Far

```
Shader "Custom/DiffuseShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,0,0,1)
    }
    SubShader
    {
        Tags { "LightMode" = "ForwardBase" }
        LOD 100
```

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

            struct appdata
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
                float3 worldNormal : TEXCOORD0;
            };

            float4 _Color;
            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                float3 worldNormal = UnityObjectToWorldNormal(v.normal);
                o.worldNormal = worldNormal;
                return o;
            }
            float4 frag (v2f i) : SV_Target
            {
                float3 normalDirection = normalize(i.worldNormal);

                float nl = max(0.0, dot(normalDirection, _WorldSpaceLightPos0.xyz));
                float4 diffuseTerm = nl * _Color * _LightColor0;

                return diffuseTerm;
            }
            ENDCG
        }
    }
}
```

To check out your work, apply this shader to the cube in the scene. Assign this shader to its material, and then assign the material to your cube (see Figure 5-5).

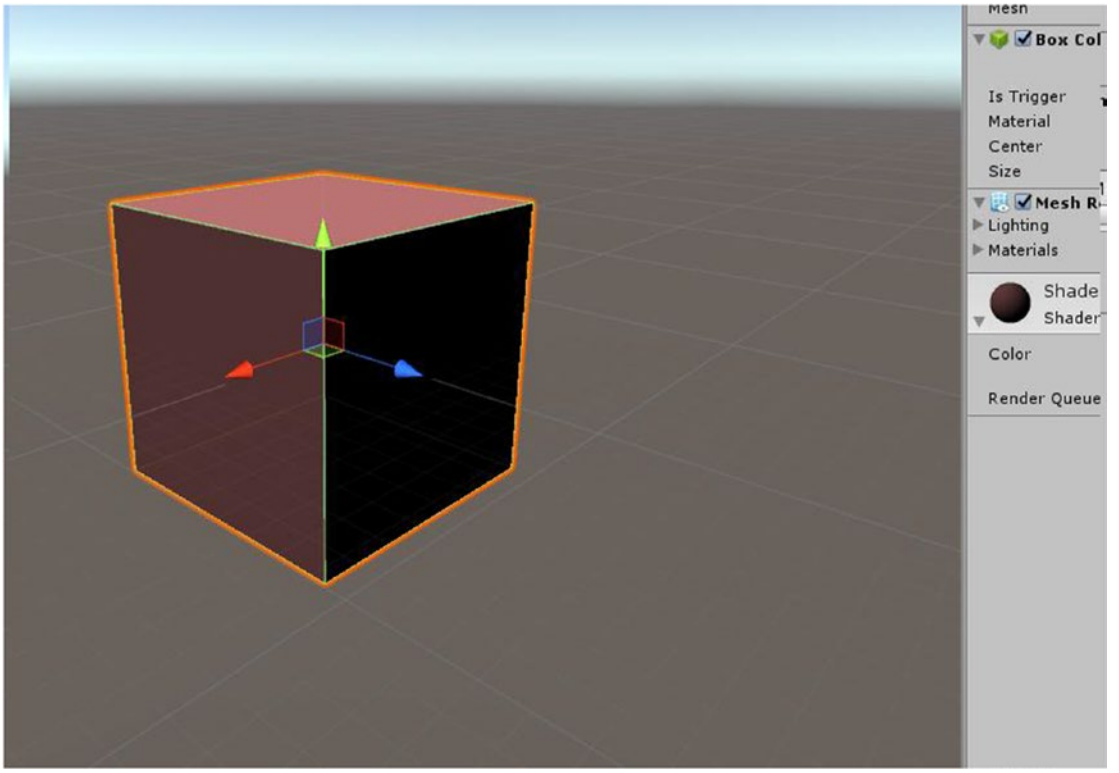

Figure 5-5. *A cube shader with our Lambert diffuse*

To better judge the result, you might want a more complex model. Let's look at how to import a model. In the source supplied with this book, you'll find a model called `duck.fbx`. Add a folder called `Models` to your project, and then drag the duck model into it. This will most likely create a folder with a bogus material inside; delete that, as we need complete control of our materials, and it's best to keep the scene tidy. Unity attempts to create a material for every imported model, but the results of this automatic conversion are often not great.

Drag the duck model in the scene, then drag `DiffuseMaterial` on the resulting GameObject, and you should be able to enjoy the view of a Lambert-shaded duck (see Figure 5-6).

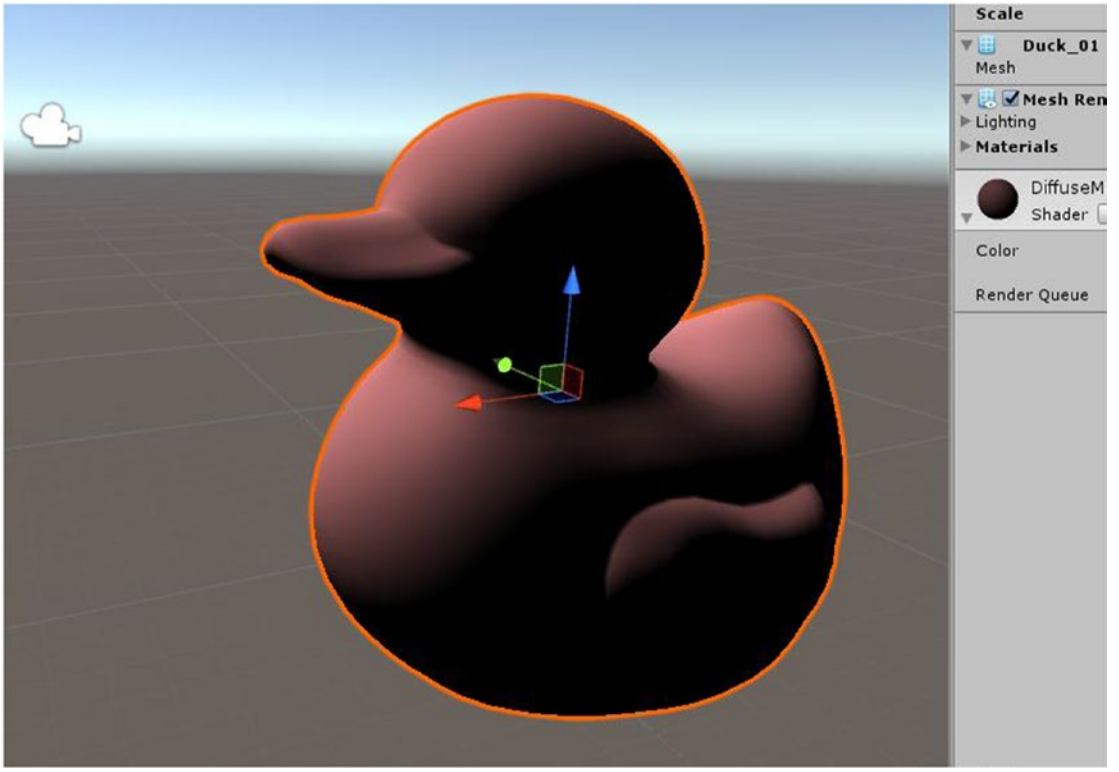

Figure 5-6. *A duck model shaded with our Lambert diffuse*

Adding a Texture Property

In order to make the duck look better, we can add a texture property. Let's add to the properties:

```
_DiffuseTex ("Texture", 2D) = "white" {}
```

Then, you need to add a slot for a texture coordinate in `appdata`:

```
float2 uv : TEXCOORD0;
```

In `v2f`, since we are already using the `TEXCOORD0` semantic for the world normal, we need to change that one to `TEXCOORD1` and then add:

```
float2 uv : TEXCOORD0;
```

This happens because we're asking the GPU to give us interpolated texture UVs in that data structure. There is a finite number of texture interpolators we can access, and that's different depending on the GPU in your machine. If you work with mobile GPUs and try to pass too many vectors in a data structure, you might encounter a compiler error.

Let's add the variables for the texture:

```
sampler2D _DiffuseTex;
float4 _DiffuseTex_ST;
```

In the vertex function, we're going to add this:

```
o.uv = TRANSFORM_TEX(v.uv, _MainTex);
```

This is the macro that scales and offsets texture coordinates. This way, any changes in the material properties regarding scales and offsets will be applied here. It's also the reason why we're declaring `_DiffuseTex_ST`, because it's needed by `TRANSFORM_TEX`. Now we're going to change the fragment function. We need to add a line to sample the texture, and then we need to use this texture with the diffuse calculations and the already existing `_Color` property.

```
float4 frag (v2f i) : SV_Target
{
    float3 normalDirection = normalize(i.worldNormal);

    float4 tex = tex2D(_DiffuseTex, i.uv);

    float nl = max(_0.0, dot(normalDirection, _WorldSpaceLightPos0.xyz));
    float4 diffuseTerm = nl * _Color * tex * _LightColor0;

    return diffuseTerm;
}
```

We obtain the final color by multiplying the `_Color` with the texture sample color, and the dot product of normal and light directions. Now you should make a new directory called `Textures`, drag in the `Duck_DIFF.tga` texture from the source code, and assign it to the material property we just added, through the inspector. Et voila, your duck will now look much more like a duck (see Figure 5-7).

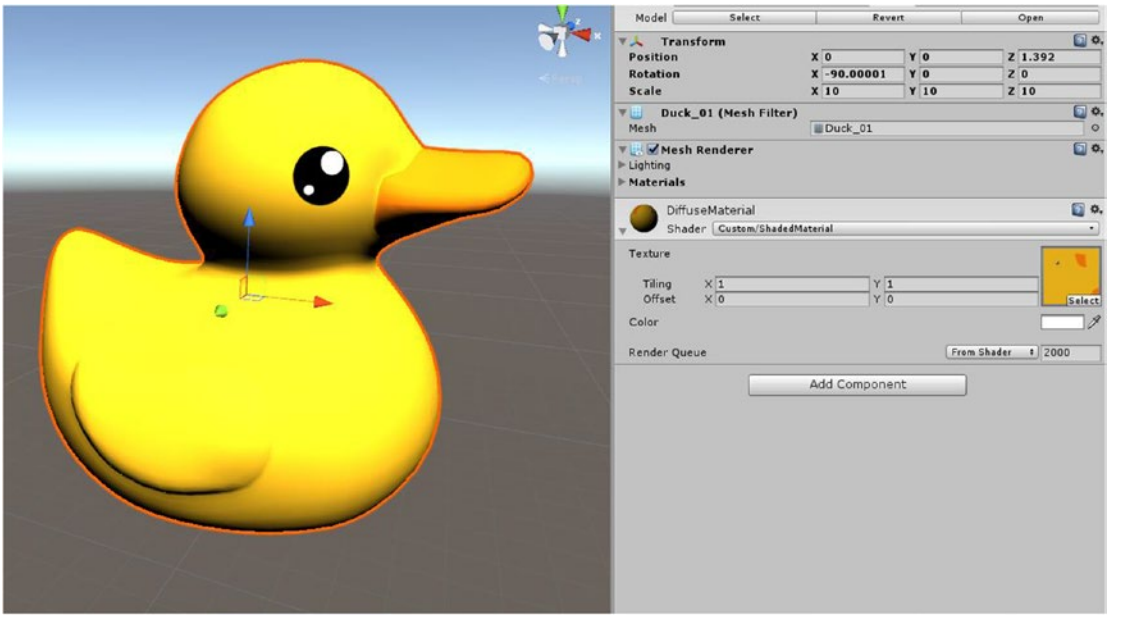

Figure 5-7. *Our duck benefiting from appropriate textures*

Adding an Ambient Value

As mentioned, an ambient is basically a cutoff value, under which we shouldn't let our diffuse value drop. At the moment, it's hard-coded as `0` in the fragment shader:

```
float nl = max(0.0, dot(normalDirection, _WorldSpaceLightPos0.xyz));
```

However, we should probably add a property for it, so that we can change it without changing the code. This also introduces you to another type of property, ranges:

```
_Name ("Description", Range (min, max)) = number
```

So let's call this property `_Ambient`, with a range of 0 to 1, and a default value of 0.25. Then let's create, as usual, a variable for it, called `float _Ambient`. We will put this variable in place of the `0` as the first argument of the `max` function, where we calculate our `nl` variable.

Now you can play with the slider for this property and see in real-time how changing it influences the final effect (see Figure 5-8).

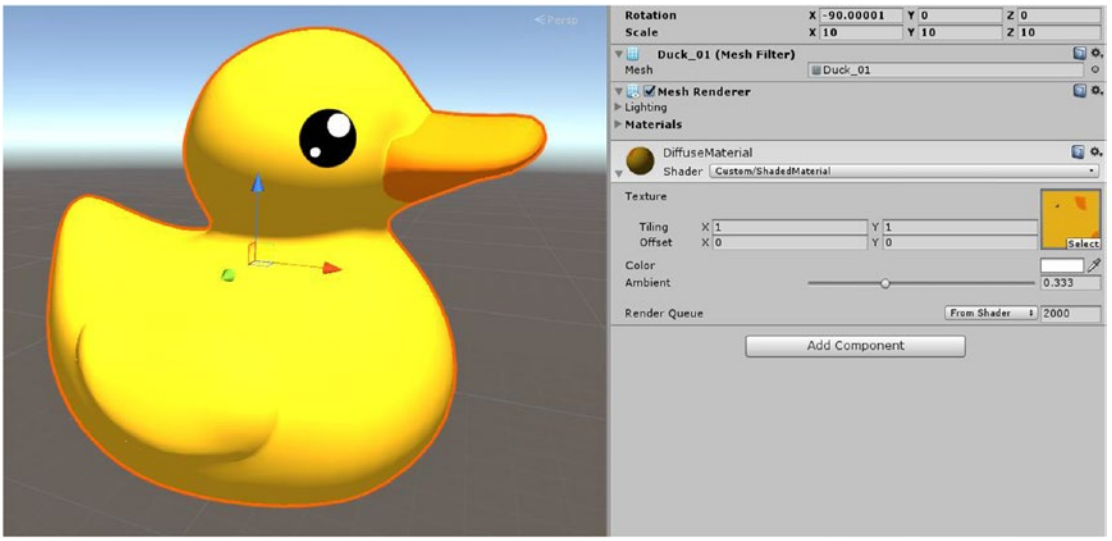

Figure 5-8. *Adding an Ambient slider*

Listing 5-3 shows the complete code for this shader.

Listing 5-3. Our DiffuseShader with Added Properties

```
Shader "Custom/DiffuseShader"
{
    Properties
    {
        _DiffuseTex ("Texture", 2D) = "white" {}
        _Color ("Color", Color) = (1,0,0,1)
        _Ambient ("Ambient", Range (0, 1)) = 0.25
    }
    SubShader
    {
        Tags { "LightMode" = "ForwardBase" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"
            #include "UnityLightingCommon.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float2 uv : TEXCOORD0;
            };
```

```
            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
                float3 worldNormal : TEXCOORD1;
            };

            sampler2D _DiffuseTex;
            float4 _DiffuseTex_ST;
            float4 _Color;
            float _Ambient;
            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _DiffuseTex);
                float3 worldNormal = UnityObjectToWorldNormal(v.normal);
                o.worldNormal = worldNormal;
                return o;
            }
            float4 frag (v2f i) : SV_Target
            {
                float3 normalDirection = normalize(i.worldNormal);

                float4 tex = tex2D(_DiffuseTex, i.uv);

                float nl = max(_Ambient, dot(normalDirection, _WorldSpaceLightPos0.xyz));
                float4 diffuseTerm = nl * _Color * tex * _LightColor0;

                return diffuseTerm;
            }
            ENDCG
        }
    }
}
```

Summary

This chapter introduced the basics of lighting, explained some lighting formulas, and expanded our knowledge of shader writing and ShaderLab properties by implementing diffuse lighting. We also introduced how to deal with *assets*, a subject that will come back in a later chapter.

Next

In the next chapter, we're going to add a specular component to this diffuse shader.

CHAPTER 6

Specular Implementation

In the last chapter, we explained the basic theory of lighting for rendering and implemented a diffuse shader from scratch, within an Unlit shader. In this chapter, you're going to learn how to add a Specular term to that shader.

Calculating Basic Lighting (Part II)

In the past chapter, we went through the theory for the diffuse approximation; now it's Specular approximation's turn.

Specular

You will only see specular if your point of view happens to be lined up with the direction of the specular. That makes it *view-dependent*. You can bake diffuse lighting in standard lightmaps, but specular requires you to use some tricks in baking or calculate it in real-time.

One of the simplest formulation of specular we can use is called *Phong:*

$$R = 2 \times (N \cdot L) \times N - L$$

This is the *reflection direction*. It can be obtained by multiplying the dot product of Normal and Light directions with two and the Normal direction, and then subtracting the Light direction. In many shader languages there is a function for this, generally called *reflect*.

Listing 6-1. An Implementation of Phong

```
float3 reflectionVector = reflect (-lightDir, normal);
float specDot = max(dot(reflectionVector, eyeDir), 0.0);
float spec = pow(specDot, specExponent);
```

To implement Phong (see Listing 6-1), you need to first calculate the mirror reflection direction, then calculate the dot product of the reflection direction, and the view direction—as we said, specular is view-dependent. Then you elevate this value to the power of the exponent that you choose in the shader properties. That controls the specular intensity. You'll see in a future chapter how this approach, while loosely based on physical reality, is actually violating many rules of *physically based shading*. When you correct it to conform to PBS principles, even a simple Phong feels much more realistic.

Now we've introduced the theory and implementation of a Specular term, let's put this into practice by adding it to the diffuse shader.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_6

Your First Lighting Unity Shader (Part II)

In this section, you're going to take the `DiffuseShader` and add a specular term to it.

Implementing a Specular

Let's create a new material, called `SpecularMaterial`. Duplicate the `DiffuseShader` and call the duplicate `SpecularShader`. Remember to change the shader path to `Custom/SpecularShader`, or you will have two overlapping shader path names.

The structure of the shader is unchanged, so no worries there. First, add two new properties to the property block: `_SpecColor` and `_Shininess. _SpecColor` is the color of the specular and uses white as the default. `_Shininess` is the intensity of the specular and it is one number:

```
Properties
{
    _DiffuseTex ("Texture", 2D) = "white" {}
    _Color ("Color", Color) = (1,0,0,1)
    _Ambient ("Ambient", Range (0, 1)) = 0.25
    _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
    _Shininess ("Shininess", Float) = 10
}
```

Next, add a member to the `v2f` struct. You need to calculate an extra value in the vertex shader, and then pass them the fragment shader through `v2f`. That extra value is the world space vertex position, which we're going to call `vertexWorld`:

```
struct v2f
{
    float2 uv : TEXCOORD0;
    float4 vertexClip : SV_POSITION;
    float4 vertexWorld : TEXCOORD1;
    float3 worldNormal : TEXCOORD2;
};
```

You do this because you need to calculate the light direction in the fragment shader. You could do it in the vertex shader, and that would be called being *vertex-lit.* But the result, unsurprisingly, is going to look better if you do the lighting calculations in the fragment shader.

Now, fill in that value in the vertex shader:

```
v2f vert (appdata v)
{
    v2f o;
    o.vertexClip = UnityObjectToClipPos(v.vertex);
    o.vertexWorld = mul(unity_ObjectToWorld, v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _DiffuseTex);
    float3 worldNormal = UnityObjectToWorldNormal(v.normal);
    o.worldNormal = worldNormal;
    return o;
}
```

In this line, we're using a matrix multiplication to transform the local space vertex position to the world space vertex position. `unity_ObjectToWorld` is the matrix you need for this transformation; it's included in the standard library.

Now you're ready to start adding the lines you need to the fragment function. There are a few values that you need to calculate, such as the normalized world space normal, the normalized view direction, and the normalized light direction (see Listing 6-2).

Listing 6-2. Values Needed for Specular Calculations

```
float3 normalDirection = normalize(i.worldNormal);
float3 viewDirection = normalize(UnityWorldSpaceViewDir(i.vertexWorld));
float3 lightDirection = normalize(UnityWorldSpaceLightDir(i.vertexWorld));
```

For best results, all the vectors need to be normalized after a transform. Depending on the situation, you might get away with avoiding that, but you risk bad artifacts in your lighting. Note that all those values are in the same coordinate space: *World Space*.

Here we're using various utility functions. You could use `mul` and the appropriate matrices, but Unity 2017.x is substituting those with the utility functions anyway. For example, if you were to use `mul(UNITY_MATRIX_MVP, v.vertex)` after saving and loading in Unity, you'd find it exchanged to `UnityObjectToClipPos(v.vertex)` and this message added to the top of the file:

```
// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'
```

Hence, it's pretty pointless not to use the utility functions in the first place. Moving on, after the diffuse implementation, you need to translate those pseudo-code lines from the Specular explanation into valid Unity shader code (see Listing 6-3).

Listing 6-3. Specular Calculations

```
float3 reflectionDirection = reflect(-lightDirection, normalDirection);
float3 specularDot = max(0.0, dot(viewDirection, reflectionDirection));
float3 specular = pow(specularDot, _Shininess);
```

First you find the `reflectionDirection` using the `reflect` function. You need to negate the `lightDirection`, so it goes from the object to the light. Then you calculate the *dot product* between `viewDirection` and `reflectionDirection`, which is the same sort of operation that you used to calculate how much light is being reflected off a surface in the diffuse term.

In the diffuse, it was between the normal and the light directions. Here, it's between the mirror reflection direction and the view direction, because the specular term is *view-dependent*. Note that again, the dot product value cannot be negative. You can't have negative light.

Then you need to add the specular to the final output. For the diffuse, you multiply it by the color of the surface. The equivalent of that for the specular is multiplying by the specular color.

```
float4 specularTerm = float4(specular, 1) * _SpecColor * _LightColor0;
```

As you may have noticed, we are not multiplying the specular by the surface color, as you do for the diffuse. That would basically make it disappear. There are physical principles underlying this, which we're going to explain when we introduce physically based shading.

Listing 6-4 shows the complete fragment shader.

Listing 6-4. The Complete Fragment Shader, Including a Diffuse and a Specular Term

```
float4 frag (v2f i) : SV_Target
{
    float3 normalDirection = normalize(i.worldNormal);
    float3 viewDirection = normalize(UnityWorldSpaceViewDir(i.vertexWorld));
    float3 lightDirection = normalize(UnityWorldSpaceLightDir(i.vertexWorld));

    // sample the texture
    float4 tex = tex2D(_DiffuseTex, i.uv);

    //Diffuse implementation (Lambert)
    float nl = max(0, dot(normalDirection, lightDirection));
    float4 diffuseTerm = nl * _Color * tex * _LightColor0;

    //Specular implementation (Phong)
    float3 reflectionDirection = reflect(-lightDirection, normalDirection);
    float3 specularDot = max(0.0, dot(viewDirection, reflectionDirection));
    float3 specular = pow(specularDot, _Shininess);
    float4 specularTerm = float4(specular, 1) * _SpecColor * _LightColor0;

    float4 finalColor = diffuseTerm + specularTerm;
    return finalColor;
}
```

Finally, the whole shader is shown in Listing 6-5, for your convenience.

Listing 6-5. Complete Shader with a Diffuse and Specular Term

```
Shader "Custom/SpecularShader"
{
    Properties
    {
        _DiffuseTex ("Texture", 2D) = "white" {}
        _Color ("Color", Color) = (1,0,0,1)
        _Ambient ("Ambient", Range (0, 1)) = 0.25
        _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
        _Shininess ("Shininess", Float) = 10
    }
    SubShader
    {
        Tags { "LightMode" = "ForwardBase" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"
            #include "UnityLightingCommon.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
```

```
            float3 normal : NORMAL;
            float2 uv : TEXCOORD0;
        };

        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertexClip : SV_POSITION;
            float4 vertexWorld : TEXCOORD2;
            float3 worldNormal : TEXCOORD1;
        };

        sampler2D _DiffuseTex;
        float4 _DiffuseTex_ST;
        float4 _Color;
        float _Ambient;
        float _Shininess;
        v2f vert (appdata v)
        {
            v2f o;
            o.vertexClip = UnityObjectToClipPos(v.vertex);
            o.vertexWorld = mul(unity_ObjectToWorld, v.vertex);
            o.uv = TRANSFORM_TEX(v.uv, _DiffuseTex);
            float3 worldNormal = UnityObjectToWorldNormal(v.normal);
            o.worldNormal = worldNormal;
            return o;
        }
        float4 frag (v2f i) : SV_Target
        {
            float3 normalDirection = normalize(i.worldNormal);
            float3 viewDirection = normalize(UnityWorldSpaceViewDir(i.vertexWorld));
            float3 lightDirection = normalize(UnityWorldSpaceLightDir(i.vertexWorld));
            // sample the texture
            float4 tex = tex2D(_DiffuseTex, i.uv);

            //Diffuse implementation (Lambert)
            float nl = max(_Ambient, dot(normalDirection, lightDirection));
            float4 diffuseTerm = nl * _Color * tex * _LightColor0;
            //diff.rbg += ShadeSH9(half4(i.worldNormal,1));
            //Specular implementation (Phong)
            float3 reflectionDirection = reflect(-lightDirection, normalDirection);
            float3 specularDot = max(0.0, dot(viewDirection, reflectionDirection));
            float3 specular = pow(specularDot, _Shininess);
            float4 specularTerm = float4(specular, 1) * _SpecColor * _LightColor0;

            float4 finalColor = diffuseTerm + specularTerm;
            return finalColor;
        }
        ENDCG
    }
  }
}
```

Figure 6-1 shows how this looks when applied to our duck.

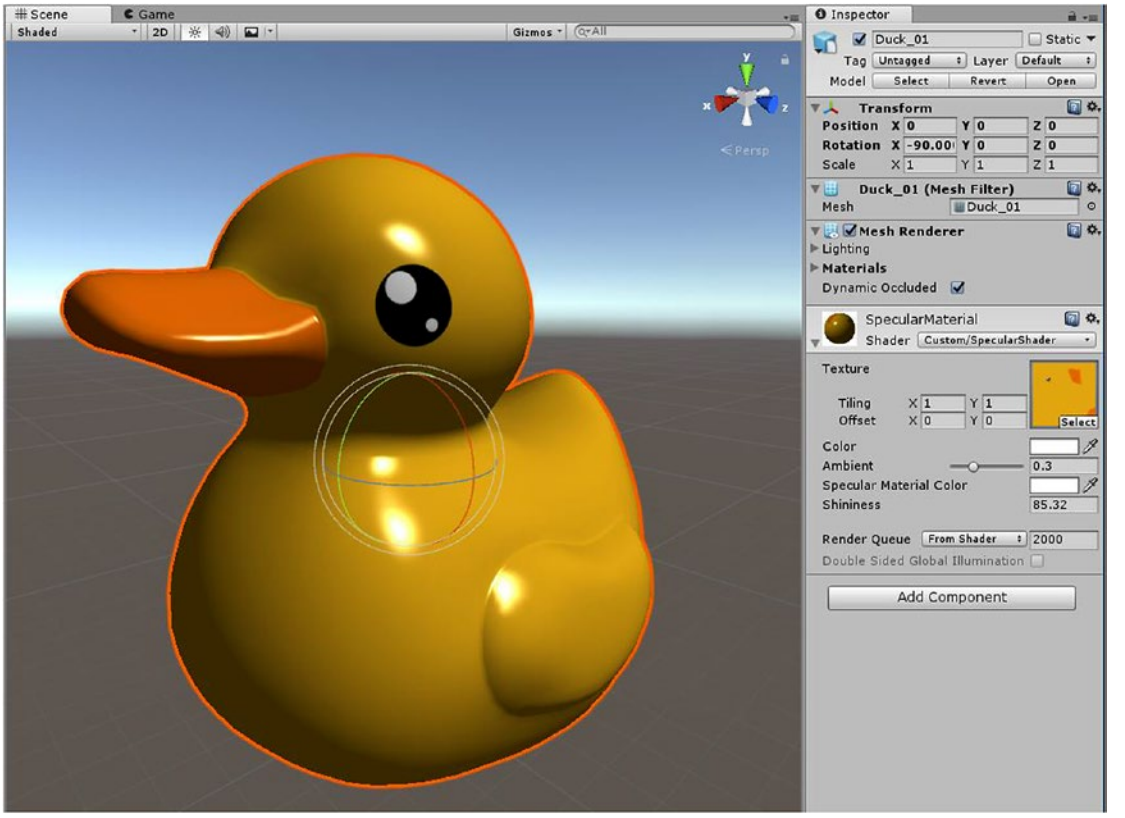

Figure 6-1. *The specular shader applied to the duck model from before*

This concludes the introduction to the non-physically based Phong specular. Up to now, you have supported only one light in the scene, but by adding a `ForwardAdd` pass, you can support any number of lights in a scene. Let's add a `ForwardAdd` pass to this shader.

Supporting More Than One Light

One thing you need to be careful of is to ensure that `ForwardAdd` is a separate pass, because you don't want to do things like add the ambient more than once. Copy the specular shader and change the shader path to `Custom/SpecularShaderForwardAdd`.

You need to move tags and other info that's currently residing within the sub-shader, to within the pass, and copy and paste the current pass. Add `#pragma multi_compile_fwdbase` after the other pragmas in the `ForwardBase` pass (see Listing 6-6).

Listing 6-6. Set Up for the ForwardBase Pass

```
Pass
{
    Tags { "LightMode" = "ForwardBase" }

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #pragma multi_compile_fwdbase
```

Then we should change the tag to the second pass `ForwardAdd`, add the line `Blend One One` after the tags, and add the pragma `#pragma multi_compile_fwdadd` (see Listing 6-7).

`ForwardAdd` tells the compiler it should use this pass for any light after the first, and `Blend One One` sets up the blending mode. Blending modes are basically similar to the layer modes in Photoshop. We have different layers, rendered by different passes, and we want to blend them together in a way that makes sense. Keep in mind that the blending modes are much simpler than those available in Photoshop. The formula is `Blend SrcFactor DstFactor`; we used one for both factors which means that the colors are blended additively.

The pragmas are to take advantage of the automatic multi-compile system, which compiles all the variants of the shader needed for the specific pass to work.

Listing 6-7. Set Up for the ForwardAdd Pass

```
Pass
{
    Tags { "LightMode" = "ForwardAdd" }
    Blend One One

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #pragma multi_compile_fwdadd
```

This is already enough to have lights after the first influence the result. The finishing touch is to remove `_Ambient` as the minimum in the `ForwardAdd` pass, which means we won't add the ambient twice or more (see Listing 6-8).

Listing 6-8. Avoiding Re-Adding the Ambient Term into the ForwardAdd Pass

```
//Diffuse implementation (Lambert)
float nl = max(0.0, dot(normalDirection, lightDirection));
```

Let's try it out by adding a different light to the project, maybe using two different colors to keep the two lights apart, and then changing the shader applied to the duck to the new one shown in Figure 6-2.

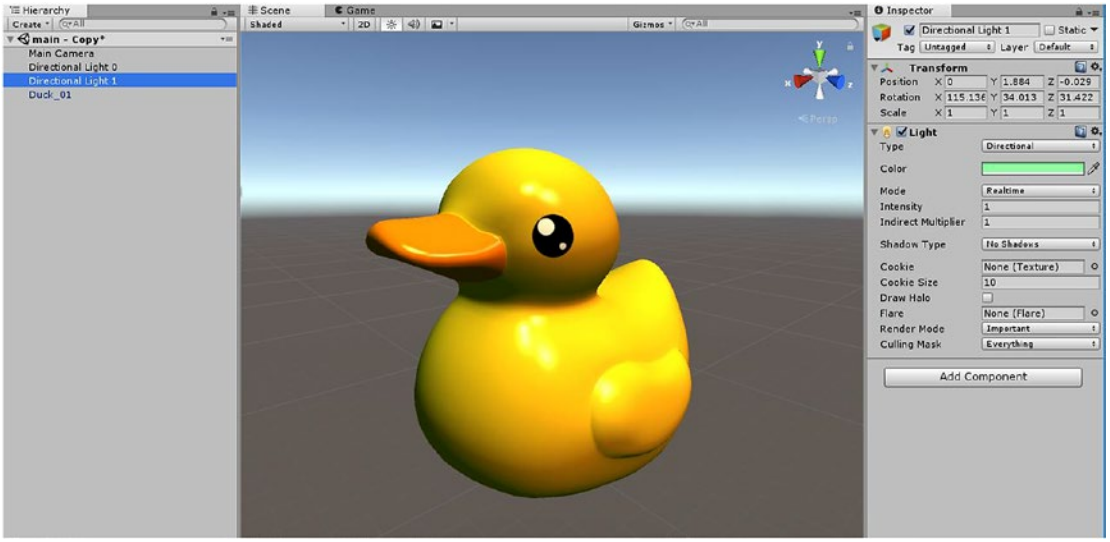

Figure 6-2. *Our duck in a scene with two lights*

For your convenience, Listing 6-9 shows the resulting complete shader, as you can see it's quite long because the code is duplicated.

Listing 6-9. The Complete Shader, with Support for Multiple Lights

```
Shader "Custom/SpecularShaderForwardAdd"
{
    Properties
    {
        _DiffuseTex ("Texture", 2D) = "white" {}
        _Color ("Color", Color) = (1,0,0,1)
        _Ambient ("Ambient", Range (0, 1)) = 0.25
        _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
        _Shininess ("Shininess", Float) = 10
    }
    SubShader
    {
        Pass
        {
            Tags { "LightMode" = "ForwardBase" }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma multi_compile_fwdbase

            #include "UnityCG.cginc"
            #include "UnityLightingCommon.cginc"
```

```
        struct appdata
        {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float2 uv : TEXCOORD0;
        };

        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertexClip : SV_POSITION;
            float4 vertexWorld : TEXCOORD2;
            float3 worldNormal : TEXCOORD1;
        };

        sampler2D _DiffuseTex;
        float4 _DiffuseTex_ST;
        float4 _Color;
        float _Ambient;
        float _Shininess;
        v2f vert (appdata v)
        {
            v2f o;
            o.vertexClip = UnityObjectToClipPos(v.vertex);
            o.vertexWorld = mul(unity_ObjectToWorld, v.vertex);
            o.uv = TRANSFORM_TEX(v.uv, _DiffuseTex);
            float3 worldNormal = UnityObjectToWorldNormal(v.normal);
            o.worldNormal = worldNormal;
            return o;
        }
        float4 frag (v2f i) : SV_Target
        {
            float3 normalDirection = normalize(i.worldNormal);
            float3 viewDirection = normalize(UnityWorldSpaceViewDir(i.vertexWorld));
            float3 lightDirection = normalize(UnityWorldSpaceLightDir(i.vertexWorld));
            float4 tex = tex2D(_DiffuseTex, i.uv);

            float nl = max(_Ambient, dot(normalDirection, lightDirection));
            float4 diffuseTerm = nl * _Color * tex * _LightColor0;
            float3 reflectionDirection = reflect(-lightDirection, normalDirection);
            float3 specularDot = max(0.0, dot(viewDirection, reflectionDirection));
            float3 specular = pow(specularDot, _Shininess);
            float4 specularTerm = float4(specular, 1) * _SpecColor * _LightColor0;

            float4 finalColor = diffuseTerm + specularTerm;
            return finalColor;
        }
        ENDCG
    }
    Pass
    {
```

```
        Tags { "LightMode" = "ForwardAdd" }
        Blend One One

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #pragma multi_compile_fwdadd

        #include "UnityCG.cginc"
        #include "UnityLightingCommon.cginc"

        struct appdata
        {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
            float2 uv : TEXCOORD0;
        };

        struct v2f
        {
            float2 uv : TEXCOORD0;
            float4 vertexClip : SV_POSITION;
            float4 vertexWorld : TEXCOORD2;
            float3 worldNormal : TEXCOORD1;
        };

        sampler2D _DiffuseTex;
        float4 _DiffuseTex_ST;
        float4 _Color;
        float _Ambient;
        float _Shininess;
        v2f vert (appdata v)
        {
            v2f o;
            o.vertexClip = UnityObjectToClipPos(v.vertex);
            o.vertexWorld = mul(unity_ObjectToWorld, v.vertex);
            o.uv = TRANSFORM_TEX(v.uv, _DiffuseTex);
            float3 worldNormal = UnityObjectToWorldNormal(v.normal);
            o.worldNormal = worldNormal;
            return o;
        }
        float4 frag (v2f i) : SV_Target
        {
            float3 normalDirection = normalize(i.worldNormal);
            float3 viewDirection = normalize(UnityWorldSpaceViewDir(i.vertexWorld));
            float3 lightDirection = normalize(UnityWorldSpaceLightDir(i.vertexWorld));
            float4 tex = tex2D(_DiffuseTex, i.uv);

            float nl = max(0.0, dot(normalDirection, lightDirection));
            float4 diffuseTerm = nl * _Color * tex * _LightColor0;
            float3 reflectionDirection = reflect(-lightDirection, normalDirection);
            float3 specularDot = max(0.0, dot(viewDirection, reflectionDirection));
```

```
                float3 specular = pow(specularDot, _Shininess);
                float4 specularTerm = float4(specular, 1) * _SpecColor * _LightColor0;

                float4 finalColor = diffuseTerm + specularTerm;
                return finalColor;
            }
            ENDCG
        }
    }
}
```

Summary

In this chapter, you implemented the simplest Specular version, Phong. Then you added support for multiple lights in the shader, which resulted in a very long shader due to code duplication.

Next

The next chapter introduces *surface shaders,* which are supposed to spare you some of the translate-between-spaces work, and will save you a lot of lines of code when supporting multiple lights.

CHAPTER 7

Surface Shaders

In the previous two chapters, we explained the basic theory of lighting for rendering and implemented a diffuse and a specular shader from scratch, within an Unlit shader. In this chapter, we're going to translate the Unlit shader from last chapter into a Surface shader, which will save a fair bit of code.

What Is a Surface Shader?

A *Surface shader* is a type of shader unique to Unity, which is meant to be used for shaders that calculate surface lighting models. From this chapter on, we're only going to use Surface shaders for lighting.

Their main advantage is that they hide a fair bit of boilerplate code. For example, thinking about the previous chapter's shader, you'll remember that you had to basically copy and paste the entire shader to make the shader support more than one light. That is pretty cumbersome. Surface shaders fix that, and the price is some loss in flexibility. It may happen that you do actually need to use an Unlit shader due to needing more precise control over the `ForwardBase` pass, but for our purposes, Surface shaders will do nicely.

The structure of Surface shaders differs from Unlit shaders. In Unlit shaders, we use two shader functions (vertex, and fragment), two data structures (one for the input to the vertex function, the other for the output), and if you want to support more than one light you need to write two passes, `ForwardAdd` and `ForwardBase`. In a Surface shader, the vertex function is optional, so you still use two data structures but they have different purposes, and you don't specify the fragment function at all, but you have to write a surface function instead. Also you can optionally write your own lighting model function.

The Default Surface Shader

Let's create a new Surface shader by right-clicking on the Project pane and choosing Create ➤ Shader ➤ Standard Surface Shader. Listing 7-1 shows what we get.

Listing 7-1. The Default Surface Shader

```
Shader "Custom/SurfaceShader" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
```


C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_7

```
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        #pragma surface surf Standard fullforwardshadows

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        // Add instancing support for this shader. You need to check 'Enable Instancing' on
        materials that use the shader.
        // See https://docs.unity3d.com/Manual/GPUInstancing.html for more information about
        instancing.
        // #pragma instancing_options assumeuniformscaling
        UNITY_INSTANCING_CBUFFER_START(Props)
            // put more per-instance properties here
        UNITY_INSTANCING_CBUFFER_END

        void surf (Input IN, inout SurfaceOutputStandard o) {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

As promised, no vertex function and no fragment function, but a new surface function and a new pragma.

Pragmas

We still have a block of properties, but the `vert` and `frag` pragmas are not around anymore. In their place, there is a single `surface` pragma. The `surface` pragma takes as the first argument the surface function, which is your responsibility to write in your shader (we're going to explain in a bit exactly what it is in a bit), then the lighting model you're going to use, and any options.

The `surf` pragma included in this default file is:

```
#pragma surface surf Standard fullforwardshadows
```

`surf` is the surface function, `Standard` is the lighting model, and `fullforwardshadows` is an option.

If you want to use a vertex function different from the default one, you can. You can write your custom vertex function within the Surface shader, specifying the vertex input and output data structures (which usually are called `appdata` and `v2f`, for historical reasons, but you call them whatever you like), and finally pass it to the `surf` pragma this way:

```
#pragma surface surf Lambert vertex:vert
```

`surf` is again the surface function, `Lambert` is a built-in lighting model, and `vertex:vert` specifies the vertex function.

New Data Structures

Let's ignore for now the parts about instancing and examine the surface function. It takes a data structure called `Input`, which is included in this shader, and one called `SurfaceOutputStandard`, which has `inout` as a type qualifier. That means it's an input, but also an output, and you won't need two different data structures for that. This `SurfaceOutputStandard` data structure will then be sent to the lighting function (`Standard`, `BlinnPhong`, `Lambert`, or you can write a custom one).

The `Input` structure in this shader only includes the UVs, taking part of the role that was reserved for the vertex output function, `v2f`:

```
struct Input {
    float2 uv_MainTex;
};
```

You may notice that there are no include files listed, but `SurfaceOutputStandard` comes from the usual include files, particularly from `UnityPBSLighting.cginc` (see Listing 7-2).

Listing 7-2. The Data Struct SurfaceOutputStandard

```
struct SurfaceOutputStandard
{
    fixed3 Albedo;      // base (diffuse or specular) color
    fixed3 Normal;      // tangent space normal, if written
    half3 Emission;
    half Metallic;      // 0=non-metal, 1=metal
    half Smoothness;    // 0=rough, 1=smooth
    half Occlusion;     // occlusion (default 1)
    fixed Alpha;        // alpha for transparencies
};
```

The objective of this data structure is to pass information to the lighting function.

The Surface Function

The `surf` function is used to prepare the necessary data and then assign it to the data structure (see Listing 7-3).

Listing 7-3. The Default surf Function

```
void surf (Input IN, inout SurfaceOutputStandard o) {
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
    o.Albedo = c.rgb;
    // Metallic and smoothness come from slider variables
    o.Metallic = _Metallic;
    o.Smoothness = _Glossiness;
    o.Alpha = c.a;
}
```

As you can see, in this default shader four of the seven members of the `SurfaceOutputStandard` data structure are being filled in, and some of those are used by most lighting models, while some others are specific to the `Standard` lighting model used by Unity.

`Albedo` is the name used in Unity for the color of the surface, which generally comes from the diffuse texture. `Alpha` is not really used unless you're drawing a transparent mesh. `Normal` takes data that comes from Normal maps, and `Emission` is used if the mesh is supposed to emit light. Everything else is specific to the Standard lighting model.

What's a Lighting Model?

The best way to understand that is with an example. The Standard lighting function is very complex, so let's look at one we already are familiar with, the `Lambert` lighting function (see Listing 7-4) taken from `Lighting.cginc` (but liberally edited so it fits in one function).

Listing 7-4. Lambert Implemented as a Lighting Model Function

```
inline fixed4 LightingLambert (SurfaceOutput s, UnityGI gi)
{
    fixed4 c;
    UnityLight light = gi.light;
    fixed diff = max (0, dot (s.Normal, light.dir));

    c.rgb = s.Albedo * light.color * diff;
    c.a = s.Alpha;

    #ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
        c.rgb += s.Albedo * gi.indirect.diffuse;
    #endif

    return c;
}
```

This is a lighting model function. Any custom lighting models that you can write yourself would follow this same pattern. It returns a `fixed4` and takes one `SurfaceOutput` and one `UnityGI` structure. `SurfaceOutput` is similar to `SurfaceOutputStandard`, it just has fewer members, because `Lambert` is a simpler lighting model. `UnityGI` is a data structure used to pass around the indirect light calculated by

the global illumination system. Global Illumination is basically a much better way to solve the problem of calculating indirect light, which in the past chapter we crudely solved with a simple Ambient value.

You don't need to worry about global illumination yet. The important member of `UnityGI` for this topic is `light`, which is another data structure, `UnityLight. UnityLight` includes the light direction and the light color. You should recognize the calculation we used in Chapter 5 to implement `Lambert`: the dot product and the multiplication with the light and surface colors.

A lighting function is supposed to simulate the behavior of light on a surface. To do that, as you've learned from the Diffuse and Specular approximations, it's going to need some bits of information. Namely the light direction, the normal direction, the surface and light colors, and possibly the view direction.

`SurfaceOutput` and its cousin `SurfaceOutputStandard` both contain the normal and the color (albedo) of the surface as members. The light direction and color are obtained from the `UnityGI` data structure. In other words, a lighting function gets passed all the data needed to calculate lighting, either with the input data structure or by other arguments.

`Lambert` doesn't need a view direction, but if it did, the function signature would be:

```
half4 Lighting<Name> (SurfaceOutput s, half3 viewDir, UnityGI gi);
```

Using these function signatures will make the compiler recognize your function as a lighting model function, and you'll be able to use it in the surface pragma.

With this, we've looked at all the pieces need to write a Surface shader, but how do they fit together exactly?

Data Flow of a Surface Shader

The execution model of a Surface shader is somewhat unintuitive. You should take into consideration that, behind the scenes, a Surface shader is compiled to something very similar to an Unlit shader. At that point, everything boils down again to a vertex shader and a fragment shader.

Surface shaders remove some flexibility in order to save you time and lines of code. They do that by wrapping the fragment shader away, behind an interface. They break it down into a surface function and lighting model function. Furthermore, Unity gives you various pre-made lighting functions, so you only really need to worry about the surface function.

But the focus of this book is writing physically based lighting model functions and making them work with the rest of the Unity shader infrastructure. So, you'll get pretty familiar with lighting functions later on.

For now, be aware that the flow of data goes starts from the *optional* vertex function, for which you can make input and output data structures, or stick to those included in the standard library. Then it goes from the vertex function to the `Input` struct, which is passed to the surface function. In the surface function, you fill in a data structure that contains most of the data needed to calculate the lighting, which is generally named `SurfaceOutput` or similar. That struct is passed to the lighting function, which finally returns a color (see Figure 7-1).

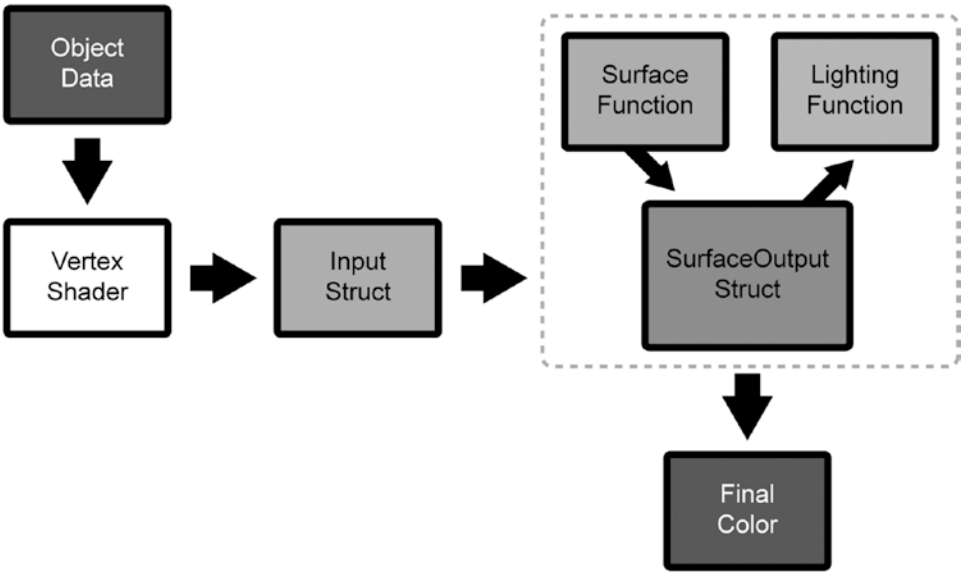

Figure 7-1. *The data flow of Surface shaders*

Editing a Surface Shader

Now that you know what a Surface shader is and how it can help you, let's go through a few examples of custom Surface shaders, which are using the Unity Standard lighting function. The Standard lighting model is physically based. In the next chapters we'll explain exactly what that means and show what code makes it physically based.

Add a Second Albedo Map

One of the most common tasks that using the Standard lighting model won't solve is when you need more textures than it gives you. Let's add a second albedo texture and then lerp between the two according to a slider.

First, let's add the second texture and the slider value (see Listing 7-5).

Listing 7-5. Adding the Properties for a Second Albedo Texture

```
Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _SecondAlbedo ("Second Albedo (RGB)", 2D) = "white" {}
        _AlbedoLerp ("Albedo Lerp", Range(0,1)) = 0.5
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
```

Then, as usual, we need to declare those as variables. After the `input` struct will do (see Listing 7-6). You may think we need to add another set of UVs for the second texture to the `Input` struct, but if the texture has the same UVs (and it should, because it's for the same model), you can recycle the same set of UVs for both.

Listing 7-6. Adding the Variables Declaration for a Second Albedo Texture

```
sampler2D _MainTex;
sampler2D _SecondAlbedo;
half _AlbedoLerp;
```

Now we need to add the appropriate lines to the `surf` function. We're basically processing our shader inputs (textures, values, etc.) in advance of sending the information to the lighting function. So, we need to look up the second texture, with the same UVs, and then assign the result of lerping the two to the albedo output (see Listing 7-7).

Listing 7-7. Sampling the Second Albedo Texture and Lerping Between the Two Albedo Textures

```
void surf (Input IN, inout SurfaceOutputStandard o) {
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
    fixed4 secondAlbedo = tex2D (_SecondAlbedo, IN.uv_MainTex);
    o.Albedo = lerp(c, secondAlbedo, _AlbedoLerp)  * _Color;
    // Metallic and smoothness come from slider variables
    o.Metallic = _Metallic;
    o.Smoothness = _Glossiness;
    o.Alpha = c.a;
}
```

That's all we need. Let's look at the final result; the second texture is a different version of the first texture, which is filled with noise. Lerping between them has the result shown in Figure 7-2.

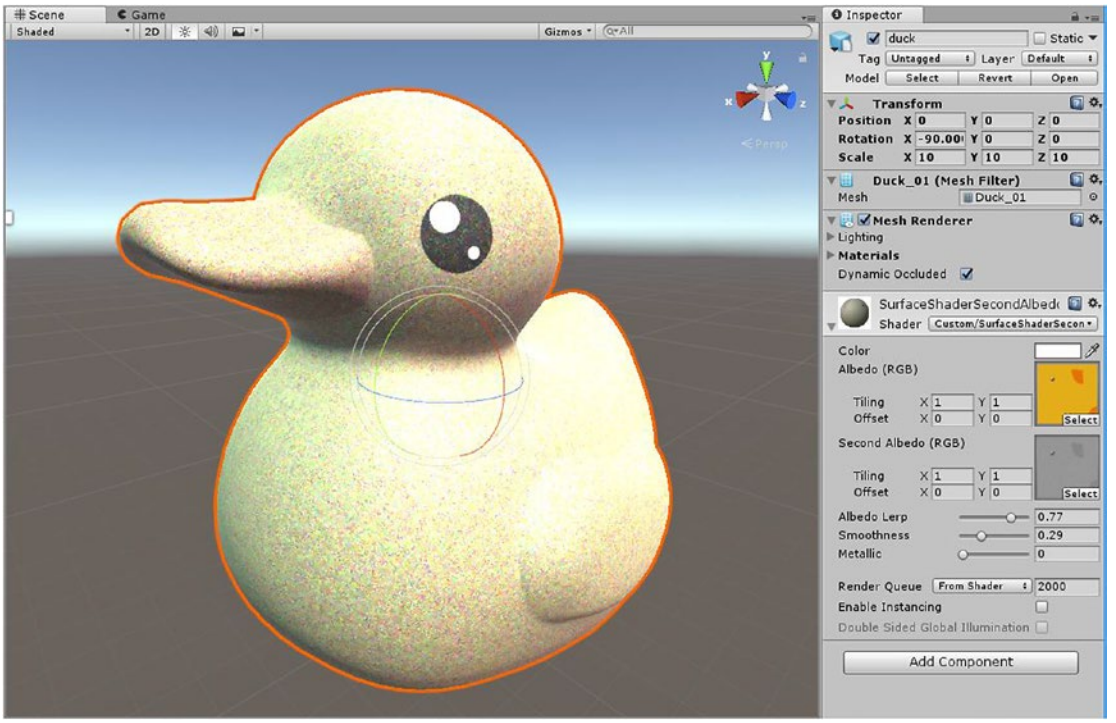

Figure 7-2. *The result of lerping between two different albedo textures*

You can see both the slight yellow tint coming from the first texture, and the noise coming from the second texture, showing up in the final result. Listing 7-8 shows the complete shader for your convenience.

Listing 7-8. Surface Shader That Lerps Between Two Albedo Textures

```
Shader "Custom/SurfaceShaderSecondAlbedo" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _SecondAlbedo ("Second Albedo (RGB)", 2D) = "white" {}
        _AlbedoLerp ("Albedo Lerp", Range(0,1)) = 0.5
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        CGPROGRAM
        #pragma surface surf Standard fullforwardshadows
        #pragma target 3.0

        sampler2D _MainTex;
        sampler2D _SecondAlbedo;
        half _AlbedoLerp;

        struct Input {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        UNITY_INSTANCING_CBUFFER_START(Props)
        UNITY_INSTANCING_CBUFFER_END

        void surf (Input IN, inout SurfaceOutputStandard o) {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
            fixed4 secondAlbedo = tex2D (_SecondAlbedo, IN.uv_MainTex);
            o.Albedo = lerp(c, secondAlbedo, _AlbedoLerp)  * _Color;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

If you need more fine-grained control, you can use a texture instead of a slider, which would work very similarly. You'd need a third texture lookup, and then extract one value out of the texture channels from the mask texture, and then use it as the value that controls the `lerp`.

Add a Normal Map

Another very common task is dealing with normal maps. Let's add a normal map to the default shader we started the chapter with. First, as usual, let's add a property for the normal map, as shown in Listing 7-9.

Listing 7-9. Adding the Normal Map Property

```
Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _NormalMap("Normal Map", 2D) = "bump" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
}
```

Then again, declare the variable, and add the appropriate handling to the `surf` function. In this case we need both to sample the texture (again the main texture UVs will do) and use the `UnpackNormal` function on it. Then we need to assign the result to the Normal member of the surface output data structure, as shown in Listing 7-10.

Listing 7-10. Declaring the Variable for the Normal Map and Unpacking the Normal Map

```
sampler2D _NormalMap;

void surf (Input IN, inout SurfaceOutputStandard o) {
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
    o.Normal = UnpackNormal (tex2D (_NormalMap, IN.uv_MainTex));
    o.Albedo = c.rgb;
    // Metallic and smoothness come from slider variables
    o.Metallic = _Metallic;
    o.Smoothness = _Glossiness;
    o.Alpha = c.a;
}
```

What we're doing with the `UnpackNormal` function here in a Surface shader would require two extra vertex shader output members (binormal and tangent in World Space) and another few lines in the fragment function, if we were to do this within an Unlit shader. So some effort is saved by using Surface shaders to deal with normal maps. The result of supporting a normal map can be seen in Figure 7-3.

Figure 7-3. *Our normal map, showing through*

For your convenience, the complete shader is shown in Listing 7-11.

Listing 7-11. The Complete Custom Surface Shader with Added Normal Map

```
Shader "Custom/SurfaceShaderNormalMap" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _NormalMap("Normal Map", 2D) = "bump" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        CGPROGRAM
        #pragma surface surf Standard fullforwardshadows
        #pragma target 3.0

        sampler2D _MainTex;
        sampler2D _NormalMap;

        struct Input {
            float2 uv_MainTex;
        };
```

```
        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        UNITY_INSTANCING_CBUFFER_START(Props)
        UNITY_INSTANCING_CBUFFER_END
        void surf (Input IN, inout SurfaceOutputStandard o) {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Normal = UnpackNormal (tex2D (_NormalMap, IN.uv_MainTex));
            o.Albedo = c.rgb;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Making Sure Shadows Work

You might have noticed that all the shaders have a fallback value that we haven't mentioned yet. The fallback is the name of a different shader that's going to be used to render the shadows for meshes that use the shader. If your fallback shader is missing or broken, the shadows for the mesh will also be broken.

If the shadows of some of your meshes are missing, this is one of the things you should check.

Use Different Built-In Lighting Models

We've been using the Standard lighting model, but we can easily switch to a different one if we want to. Let's use the `BlinnPhong` one instead. To do that, first change the `surf` pragma to this:

```
#pragma surface surf BlinnPhong fullforwardshadows
```

The `BlinnPhong` light function takes the `SurfaceOutput` data structure, instead of the `SurfaceOutputStandard`, so let's change the signature of the `surf` function to this:

```
void surf (Input IN, inout SurfaceOutput o) {
```

`BlinnPhong` has no concept of glossiness and metallic, so we should remove them from the properties, the variable declarations, and the `surf` function. Then we need to add as properties the `Gloss` and `Specular` values that the `BlinnPhong` lighting model function uses, as shown in Listing 7-12.

Listing 7-12. The Built-In BlinnPhong Lighting Model Function

```
inline fixed4 UnityPhongLight (SurfaceOutput s, half3 viewDir, UnityLight light)
{
    half3 h = normalize (light.dir + viewDir);

    fixed diff = max (0, dot (s.Normal, light.dir));

    float nh = max (0, dot (s.Normal, h));
    float spec = pow (nh, s.Specular*128.0) * s.Gloss;
```

```
    fixed4 c;
    c.rgb = s.Albedo * light.color * diff + light.color * _SpecColor.rgb * spec;
    c.a = s.Alpha;

    return c;
}
```

Specular, SpecColor, and Gloss are used in the BlinnPhong implementation. Traditionally in Unity the alpha of the albedo texture has been use to provide the gloss, and the specular has been declared as shininess in the properties. The SpecColor also needs to be added to the properties, but not to the declarations, as that happens by default. Listing 7-13 shows the final BlinnPhong Surface shader.

Listing 7-13. The Complete BlinnPhong Custom Shader

```
Shader "Custom/SurfaceShaderBlinnPhong" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
        _Shininess ("Shininess", Range (0.03, 1)) = 0.078125
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        CGPROGRAM
        #pragma surface surf BlinnPhong fullforwardshadows
        #pragma target 3.0

        sampler2D _MainTex;
        float _Shininess;

        struct Input {
            float2 uv_MainTex;
        };

        fixed4 _Color;

        UNITY_INSTANCING_CBUFFER_START(Props)
            // put more per-instance properties here
        UNITY_INSTANCING_CBUFFER_END

        void surf (Input IN, inout SurfaceOutput o) {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = _Shininess;
            o.Gloss = c.a;
            o.Alpha = 1.0f;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Figure 7-4 shows the result of applying the final `BlinnPhong` Surface shader to our duck scene.

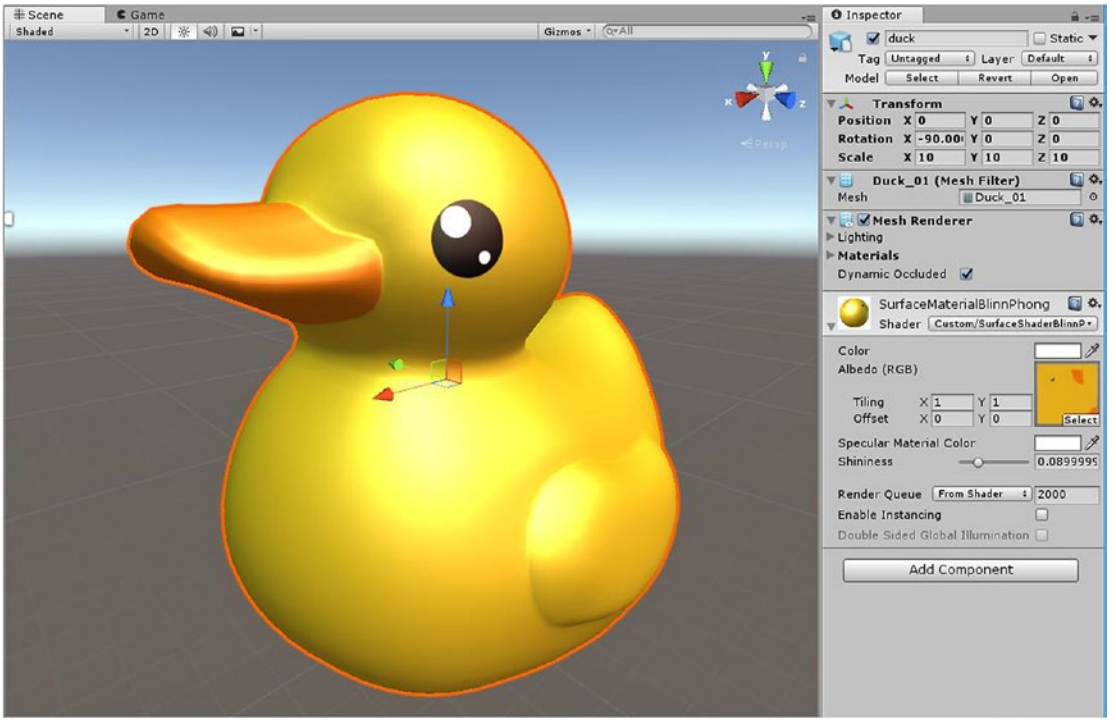

Figure 7-4. *The BlinnPhong shader applied to our usual scene*

Writing a Custom Lighting Model

Writing lighting functions is going to be our main focus from now on, so let's get started on that. Unity lacks a built-in Phong lighting model, because it prefers to use `BlinnPhong`. We can use the knowledge we already have about Phong to implement it as a custom lighting model.

Lighting Model Function Signatures

As mentioned earlier, custom lighting model functions have to conform to one of a few possible signatures. There are four of them. Two of them are used in the Forward renderer, one is diffuse only, and one is view-dependent, so it can be used for speculars. Here is the diffuse one:

```
half4 Lighting<Name> (SurfaceOutput s, UnityGI gi);
```

Here is the view-dependent one:

```
half4 Lighting<Name> (SurfaceOutput s, half3 viewDir, UnityGI gi);
```

The other two are for the current deferred renderer and the legacy deferred renderer. We're not going to cover deferred rendering in this book, because using a deferred renderer limits the data we can use in our shaders, and the more interesting lighting models require extra data that might not be available. All the principles you're learning will work with deferred renderers anyway.

Here are the two function signatures used by the deferred renderers:

```
half4 Lighting<Name>_Deferred (SurfaceOutput s, UnityGI gi, out half4 outDiffuseOcclusion,
out half4 outSpecSmoothness, out half4 outNormal);
half4 Lighting<Name>_PrePass (SurfaceOutput s, half4 light);
```

We're not going to use them, but it's good to be able to recognize them.

The SurfaceOutput Data Structure

So, moving on, Phong is definitely view-depended, hence we need to use the second type signature. We also need to know what member `SurfaceOutput` has, as shown in Listing 7-14.

Listing 7-14. The SurfaceOutput Data Structure

```
struct SurfaceOutput {
    fixed3 Albedo;
    fixed3 Normal;
    fixed3 Emission;
    half Specular;
    fixed Gloss;
    fixed Alpha;
};
```

We're not going to need `Emission` (only used when the object is supposed to emit light). The `Normal` that we get from this data structure is already in World Space, and so are the light direction and the `viewDir` that we get from the function signature.

The Surface Function

The surface function is very simple, because we only need to pass it the albedo and the alpha (see Listing 7-15).

Listing 7-15. The Surface Function for Our Phong Custom Surface Shader

```
void surf (Input IN, inout SurfaceOutput o) {
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
    o.Albedo = c.rgb;
    o.Alpha = 1.0f;
}
```

Properties Block

As you might remember, Phong uses a `SpecColor` and a `Shininess` value, so we need to add them to the properties and declare `Shininess` as a variable. Again, `SpecColor` doesn't need to be declared as a variable (see Listing 7-16).

Listing 7-16. The Properties Block for Our Phong Custom Surface Shader

```
Properties {
    _Color ("Color", Color) = (1,1,1,1)
    _MainTex ("Albedo (RGB)", 2D) = "white" {}
    _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
    _Shininess ("Shininess", Range (0.03, 128)) = 0.078125
}
```

The Custom Lighting Function

Now, this is the heart of this shader. We are adapting our Phong implementation from a few chapters ago. Keep in mind that all the directions passed into the lighting function are already in World Space. You need to take some care with using correct sources, as there is a mix of data structures passed in, plain values passed in, and variables in the properties that are used straight from there.

We're removing the ambient, because from a Surface shader, it's easier to access Unity's global illumination functionality.

Listing 7-17 shows the final lighting model function.

Listing 7-17. Our Custom Phong Lighting Model Function

```
inline fixed4 LightingPhong (SurfaceOutput s, half3 viewDir, UnityGI gi)
{
    UnityLight light = gi.light;

    float nl = max(0.0f, dot(s.Normal, light.dir));
    float3 diffuseTerm = nl * s.Albedo.rgb * light.color;
    float3 reflectionDirection = reflect(-light.dir, s.Normal);
    float3 specularDot = max(0.0, dot(viewDir, reflectionDirection)); //no more ambient
    float3 specular = pow(specularDot, _Shininess);
    float3 specularTerm = specular * _SpecColor.rgb * light.color.rgb;

    float3 finalColor = diffuseTerm.rgb + specularTerm;

    fixed4 c;
    c.rgb = finalColor;
    c.a = s.Alpha;

    #ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
        c.rgb += s.Albedo * gi.indirect.diffuse;
    #endif

    return c;
}
```

Note that block at the end, which is adding the global illumination value to the final color. In order for that to work, we need to provide an extra function, called `LightingPhong_GI`. For the time being, let's use the same GI function as the `BlinnPhong` (see Listing 7-18).

Listing 7-18. The Necessary Global Illumination Function for Our Custom Phong Lighting Model

```
inline void LightingPhong_GI (SurfaceOutput s, UnityGIInput data, inout UnityGI gi)
{
    gi = UnityGlobalIllumination (data, 1.0, s.Normal);
}
```

The last touch is to update the `surf` pragma to use our new lighting function:

```
#pragma surface surf Phong fullforwardshadows
```

The global illumination data is must stronger than our ambient, so the result is a bit too bright, as shown in Figure 7-5.

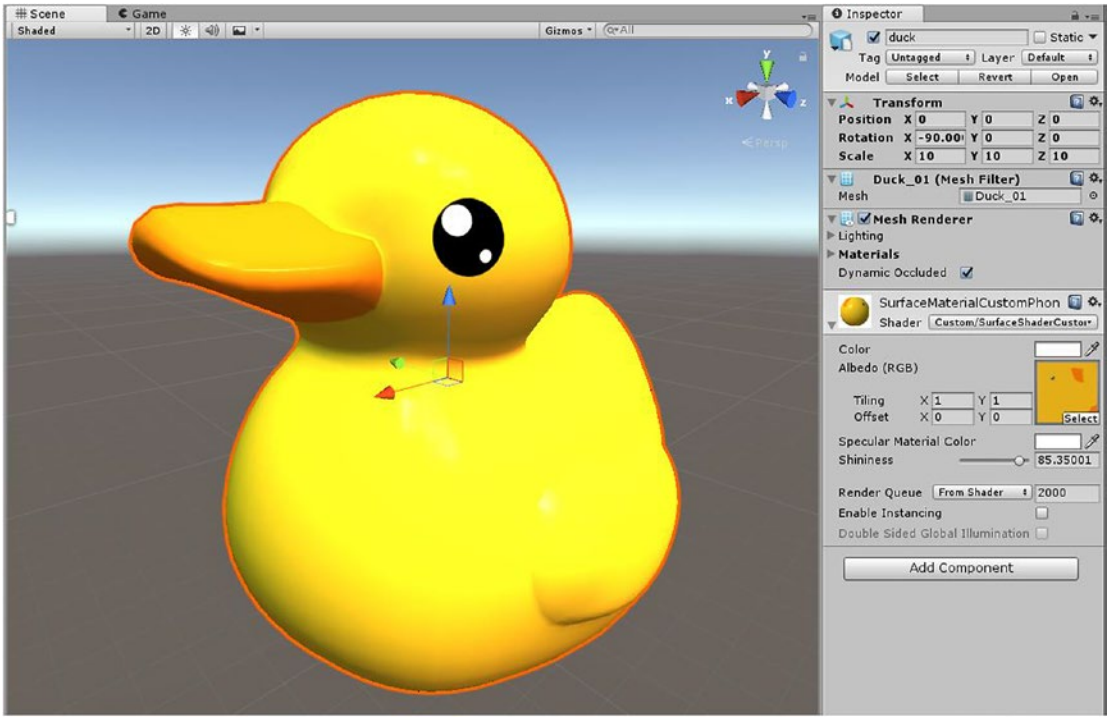

Figure 7-5. *Our custom Phong lighting model with global illumination*

If we remove the global illumination line and add back an ambient, we can see that the result is the same as with the Phong Unlit shader (see Figure 7-6).

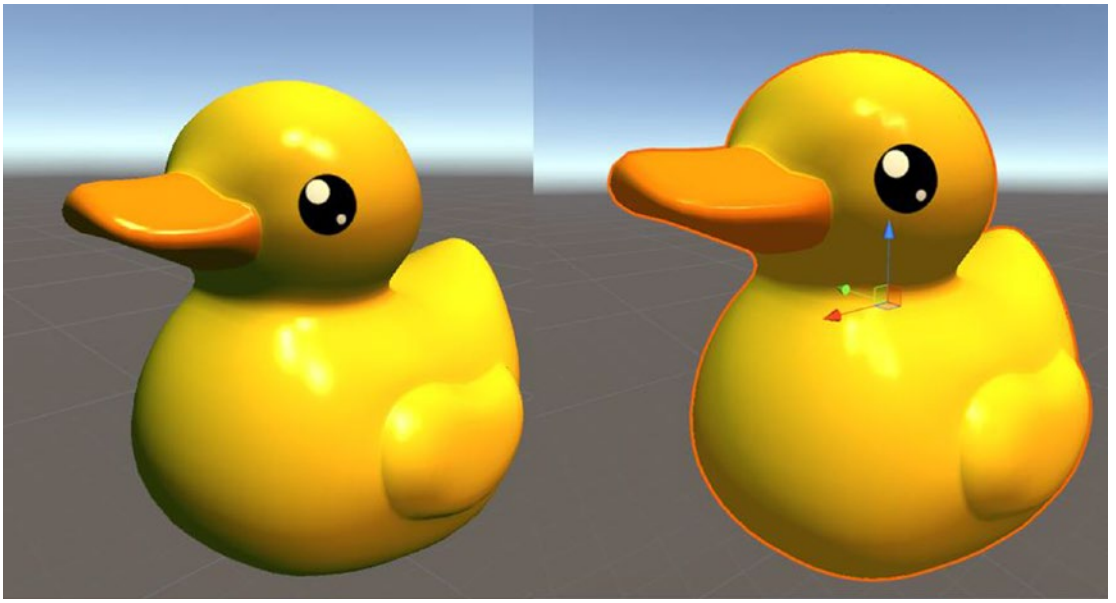

Figure 7-6. *On the left is the Unlit Phong shader; on the right is the Phong Surface shader without global illumination*

What's happening is by using the global illumination, we're mixing non-physically based parts with physically based parts, and that's a recipe for trouble. Next, we're going to go through what physically based means and how to stick to it. For your convenience, Listing 7-19 shows the final shader, with global illumination.

Listing 7-19. The Complete Shader Including the Phong Custom Lighting Model Function

```
Shader "Custom/SurfaceShaderCustomPhong" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
        _Shininess ("Shininess", Range (0.03, 128)) = 0.078125
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        CGPROGRAM
        #pragma surface surf Phong fullforwardshadows
        #pragma target 3.0

        sampler2D _MainTex;
        float _Shininess;
        fixed4 _Color;
```

```
        struct Input {
            float2 uv_MainTex;
        };

        UNITY_INSTANCING_CBUFFER_START(Props)
        UNITY_INSTANCING_CBUFFER_END

        inline void LightingPhong_GI (SurfaceOutput s, UnityGIInput data, inout UnityGI gi)
        {
            gi = UnityGlobalIllumination (data, 1.0, s.Normal);
        }

        inline fixed4 LightingPhong (SurfaceOutput s, half3 viewDir, UnityGI gi)
        {
            UnityLight light = gi.light;

            float nl = max(0.0f, dot(s.Normal, light.dir));
            float3 diffuseTerm = nl * s.Albedo.rgb * light.color;

            float3 reflectionDirection = reflect(-light.dir, s.Normal);
            float3 specularDot = max(0.0, dot(viewDir, reflectionDirection));
            float3 specular = pow(specularDot, _Shininess);
            float3 specularTerm = specular * _SpecColor.rgb * light.color.rgb;

            float3 finalColor = diffuseTerm.rgb + specularTerm;

            fixed4 c;
            c.rgb = finalColor;
            c.a = s.Alpha;

            #ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
                c.rgb += s.Albedo * gi.indirect.diffuse;
            #endif

            return c;
        }

        void surf (Input IN, inout SurfaceOutput o) {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Alpha = 1.0f;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Summary

This chapter dissected Surface shaders and discussed what they're useful for. You saw a few useful examples of how you can customize them. Most importantly, the chapter explained what a custom lighting model function is and showed how to port the Phong implementation from the Unlit shader to a custom lighting function within a Surface shader.

Next

We're going to delve into the physically based principles, explain them in detail, and start to put them into practice.

PART II

Physically Based Shading

This part of the book covers Physically Based Shading from the fundamental concepts to how to implement custom lighting models.

You will learn the *microfacet* theory, get to know many different custom lighting functions, including handy ways to visualize and analyze them and how to implement them in Unity.

You'll learn how to hack your custom lighting functions into the Unity standard shader, so you can take advantage of its extensive functionality, including specular reflections and global illumination.

You'll also learn how to implement advanced techniques not included in the usual type of custom lighting function, such as translucency. This part includes an overview on how to implement reflections, or how Unity's reflection probes work underneath.

CHAPTER 8

What Is Physically Based Shading?

Physically based shading is similar to shading as we used to do before, but now the lighting calculations are based on the physics of light with some precision. Correctness is defined as realistic behavior. You can check if the lighting implemented in your shader is correct by rendering the same scene using a physically based offline renderer.

This chapter discusses the principles of the physics of light that are interesting to us. We presented some information about the behavior of light when hitting a surface in Chapter 1, and now we're going to expand on that.

Light Is an Electromagnetic Wave

To cover this topic adequately, we have to delve briefly into physics. Starting from the nature of light, which is an electromagnetic wave, which can for some purposes be conceptually simplified to a ray. What that means for us is that there is some behavior of light that can be explained by thinking of it as a wave, such as diffraction, and other characteristics for which we can just think of light as rays, such as mirror reflection.

In general, when a surface's irregularities are larger than the width of the light's wavelength, it's safe to not worry about the nature of light as a wave. Most of the behaviors that we are interested in can be explained from a higher level of abstraction, by thinking of light as rays. Recall our previous simplification of the behavior of light for shading—diffuse and specular. Now we're going to re-explain them in more realistic, less abstract, terms.

Microfacet Theory Overview

We said that diffuse reflection happens when the microfacets on a surface are oriented in many different directions, and specular is when they are oriented in a few similar directions. That was a big simplification to get you started, so stay tuned for a more physically accurate explanation.

To begin with, *microfacet theory* is applicable when the irregularities of a surface are bigger than light's wavelength, but small compared to the scale we're observing from. Think of whether a surface is flat as a macroscale concept, whether it has microfacets as a microscale concept, and then at wavelength scale, the important thing is whether irregularities are bigger or smaller than the wavelength of the light.

Microfacet theory is useful as a guide to mathematically derive a statistical representation of where incoming light ends up going after hitting a surface. We shade on vertices or on pixels, and it's very unlikely for us to render a surface at such a magnification that we would need to worry about single rays. A statistical model is enough for our purposes.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_8

Refraction and Other Beasts

There is another behavior of light that we haven't covered yet, because it's not generally represented directly in games. We said a ray of light can be *reflected* or *absorbed*, and we mentioned how light can go through semitransparent layers, such as water, but we didn't say that what allows it to get to the layer below is called *refraction* or *transmission*.

When a ray hits a surface, some part of its light is absorbed, some is reflected, and some is refracted. And how much of the ray light goes to each is important. By "hitting," I actually mean a slightly more complex concept. Each material has an index of refraction, which represents how fast light can go through the material.

Light travels in a straight line in a medium that has a constant index of refraction, and it bends while travelling if the index of refraction changes slightly. But when passing between mediums with very different indices of refraction, it *scatters*, breaking up into multiple directions.

Absorption (see Figure 8-1) needs to happen (or not happen) to give surfaces their particular albedo color. Thinking of light as a wave, the light carries a spectrum of colors, and absorption by the surface removes every part of the spectrum, except the color that we see on the surface. That part of the spectrum is reflected away, to our eyes. For our purposes, the spectral nature of light can be simplified to RGB colors.

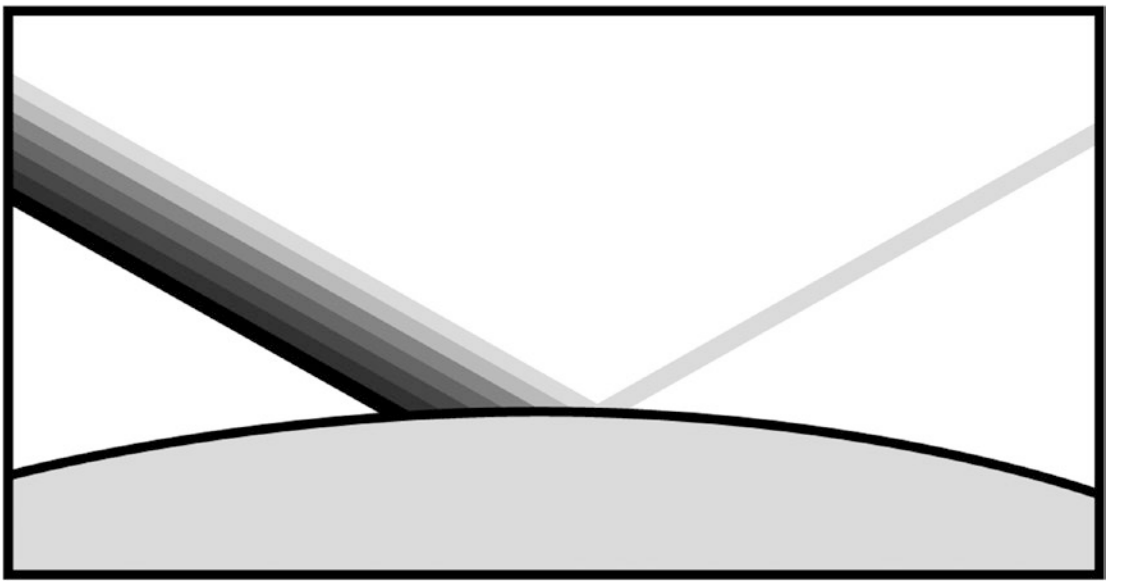

Figure 8-1. *Absorption, as shown in Chapter 1*

Reflection has been our focus up to now. As we said, when a ray hits a surface, part of its energy is sent in a particular direction, the reflection direction. But we haven't really covered how much of the energy coming in gets reflected. We've mainly worried about in what direction it goes, so far. Keeping track of the proportion of energy that goes into reflection is important for physically based shading.

Since we don't model the microgeometry in our 3D models, we have to treat it statistically, as mentioned. For every pixel we are shading, light is being reflected in multiple directions, forming a cone (see Figure 8-2).

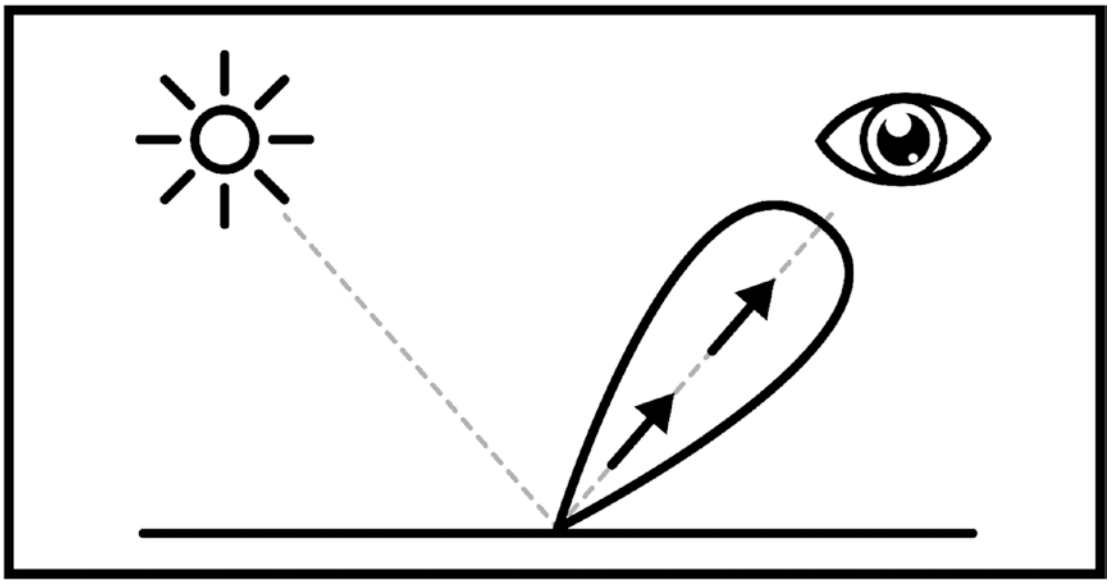

Figure 8-2. *Mirror reflection reflecting rays into a cone, generally called a lobe*

To keep track of whether the reflected light is an appropriate quantity, we need to compare it to the incoming light. We'll see in a bit to which parts of the rendering equation this maps to. Incoming light is generally measured as coming from a hemisphere around the point, such as in Figure 8-3.

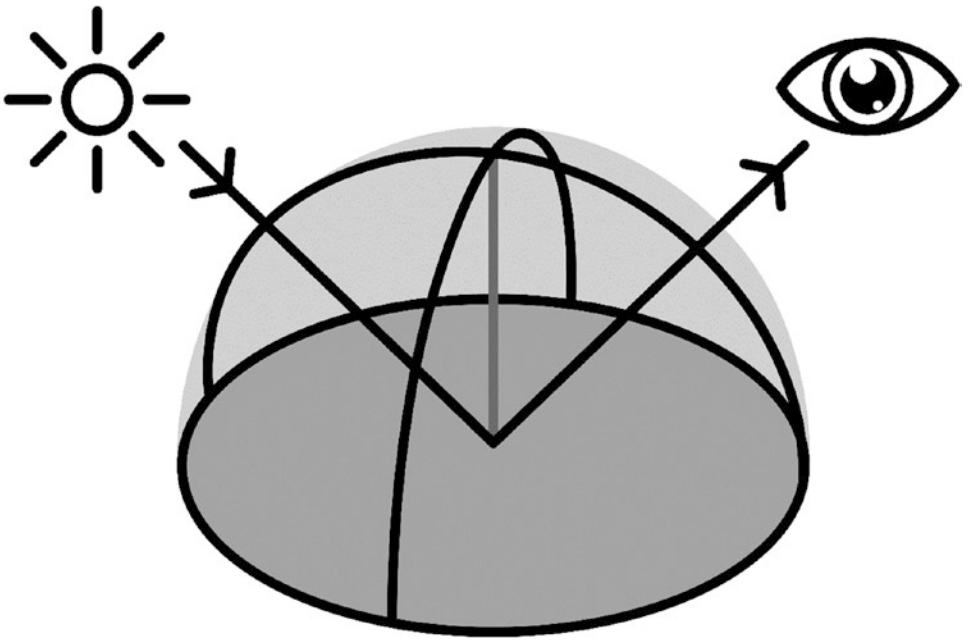

Figure 8-3. *Mirror reflection, showing incoming light from a hemisphere*

This hemisphere is basically half a sphere that's aligned around the normal of the surface. Within the rendering equation, the incoming light from every direction within this hemisphere is added up, in one of the most time-consuming parts of these calculations. Some of this light will be direct, which we can easily track and calculate, but another part of it will be indirect. That is light that has bounced around before getting to that point. That needs to heavily approximated, and it generally falls down to either *lightmapping* or *global illumination*. In the case of mirror reflection, this phenomenon is what makes possible to see the whole scene reflected on a reflective object. That needs to be simulated with yet another technique, such as *reflection probes* in Unity, or just plain cube maps.

Moving on, we have the previously overlooked *refraction* (aka, *transmission*), last but not least (see Figure 8-4). What happens with refraction is slightly more complex than mirror reflection, as it depends on the physical characteristics of the material the light is passing through. Each homogeneous material that light is passing through has an index of refraction. Every time this exiting and entering from a homogenous substance happens, light gets bent.

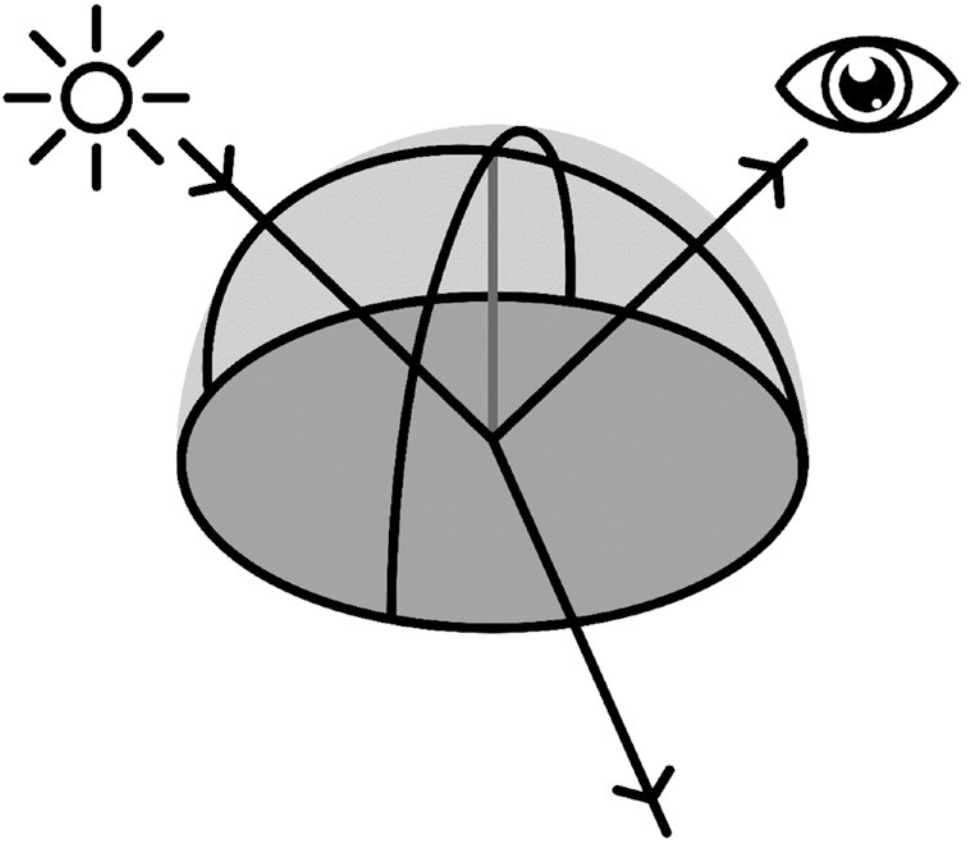

Figure 8-4. *Refraction and mirror reflection, also showing the hemisphere of incoming light directions*

When light exits a material and enters a new one, the angle changes. Depending on how much their indexes of refraction differ, the more the angle of refraction will change. From a faster to a slower material, say from air to water, the direction of the light bends toward the normal. Also, keep in mind light will change direction again when exiting the material.

The refraction direction is usually calculated using *Snell's law,* but unless we are simulating an unmistakably refractive material, such as water or glass, we're not going to worry about every single ray of light, and every single direction of refraction. As we said, we'd need to render at a truly ridiculous resolution to have a one-pixel/one-light-ray correspondence. In reality, every pixel contains so many rays of light, that it only makes sense to deal with them statistically, especially since we have to render this in real-time.

For the purposes of refraction, you can divide materials into metals and non-metals (dielectrics). Metals absorb all the refracted light, which is why we observe much lower diffuse component on metals. Non-metals scatter it around, until it's either absorbed or it exits the material again, but away from where it first entered.

This is the physical origin of the diffuse approximation. It can be simplified as a diffuse when the exit point is within the same pixel as the entrance point. If it's outside of that pixel, you need to use subsurface scattering techniques to simulate it adequately. Therefore, the distance from the camera is an important factor of whatever something can be shaded as diffuse, or should be shaded with subsurface techniques.

Fresnel Reflectance

As mentioned, keeping track of how much energy (as light intensity) is incoming and how much is going to end up outgoing from our point on the surface is important in keeping shading physically based. In the Specular approximation, we didn't worry about making sure that the specular wasn't brighter than the incoming light. This is one of the reasons why Phong doesn't feel realistic.

The **ratio** between how much incoming light is reflected and how much is refracted changes depending on the light angle, the normal, and the view angle. A glancing viewing angle will have much more reflection than a straight down one.

As an example, think of being on a boat and looking straight down. There will be little reflection, and you'll be able to see what's below the water. If, on the other hand, you look toward the horizon, you will see a lot of reflection happening, and won't see anything underneath the water.

If you graph the *Fresnel Reflectance* by the angle of incidence (see Figure 8-5) from 0 degrees to about 45, most materials hardly change at all. They start from a certain Fresnel value and stick to it. From 45 degrees to 75, they start to change, most raising toward maximum reflectance. At 90 degrees, almost every material is completely reflective.

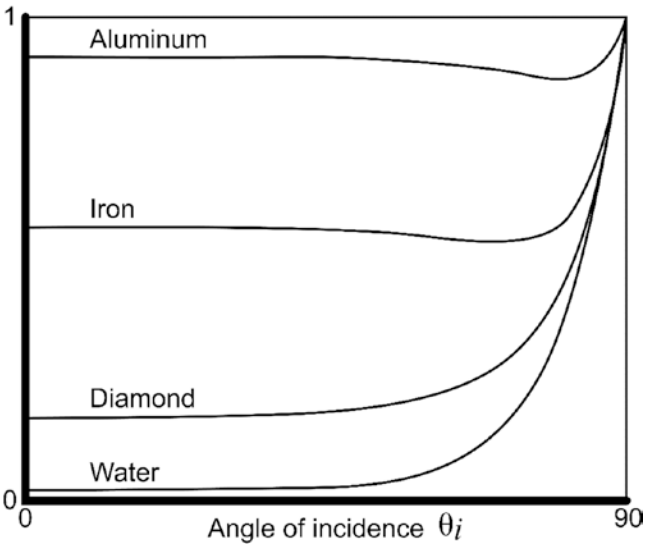

Figure 8-5. *A graph of the changes in Fresnel reflectance in materials, depending on the angle of incidence*

The color of the surface when the angle of incidence is at 0 degrees is maintained for most of the angle of incidence values, so it's convenient to simplify, and take it as the specular color of the material in general. You probably have seen material charts around that list the specular colors various materials should use. Unity's is at `https://docs.unity3d.com/Manual/StandardShaderMaterialCharts.html`.

Specular colors are mostly necessary for metals. Dielectrics tend to have less precise, darker, and monochromatic specular colors. Since metals don't have any subsurface scattering (they absorb all the light that's not reflected) if they didn't have specular color, they wouldn't have any color.

We commonly calculate this split between reflection and refraction using the Fresnel equations, commonly implemented in code through the *Schlick approximation***.** This approximation is supposed to behave similarly to those real physical world values in the graph we described.

$$F_{schlick}(F_0, l, h) = F_0 + (1 - F_0)(1 - (l \cdot h))^5$$

F_o is the specular color, meaning the color at 0 degrees of the angle of incidence. The original Fresnel formula use instead of specular color, spectral indices of refraction, which are much less convenient to deal with.

How to Measure Light

As you might have realized (maybe with some horror) by now, rendering is inextricably linked with physics. Our intuitive approximations were the state of the art a decade ago, but that has moved on. You can't really be innovative in this era of physically based shading, without having a working knowledge of the physics of light, and likely of Calculus as well.

As I write I cannot know your personal objectives for this book. You might only want to make your indie game look better. You may want to make your own physically based renderer. You may want to become a graphics researcher. Depending on your objective, you might need to delve deeper into physics, but I'm trying here to give you at least a good idea of how everything fits together. This includes introducing you to most used units of measure for light. Measuring light (and other electromagnetic radiation) in physics is called *radiometry*. You need to know at least a few units of measure that concern light to be able to read the rendering equation.

Solid Angle

A solid angle (see Figure 8-6) is expressed in a unit called a *steradian*, for which the symbol is *sr*. It's the projection of a shape onto a unit sphere. This hemisphere should be familiar to you from before.

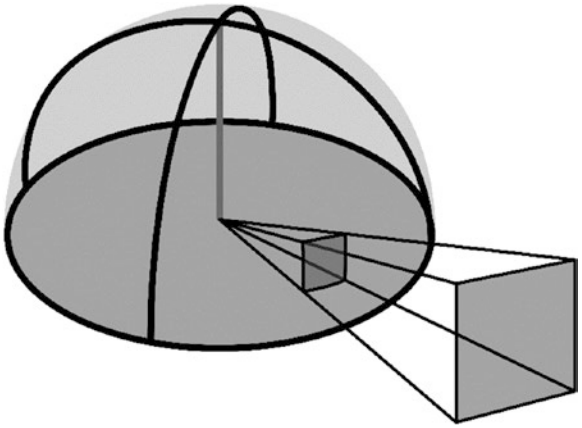

Figure 8-6. *Solid angle*

Power

The rate of transmission of energy coming from any direction, and going through a surface, is *power* (see Figure 8-7). You'll find power represented as *W*.

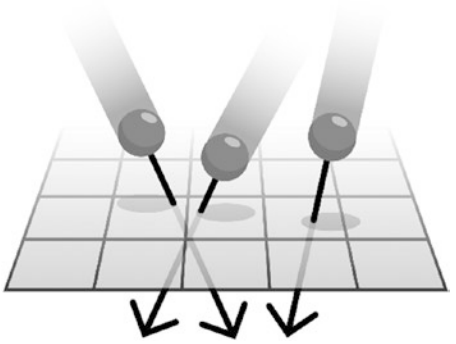

Figure 8-7. *Power*

Irradiance

Irradiance is power coming from all directions, going through a point (see Figure 8-8). In formulas, it's normally represented as *E*. You'll find it represented as W/m^2.

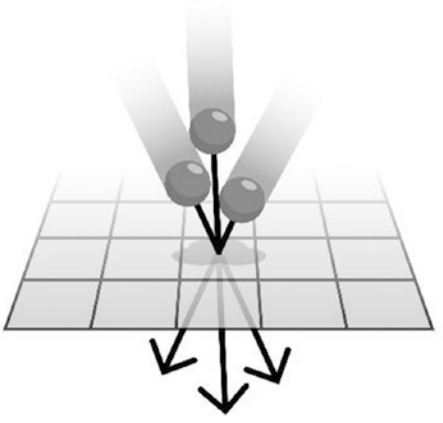

Figure 8-8. *Irradiance*

Radiance

Radiance is the magnitude of light along a single ray (see Figure 8-9). It is normally represented by L in formulas, with L_i being the incoming radiance and L_0 the outgoing radiance. You'll find it represented as $W/(m^2sr)$.

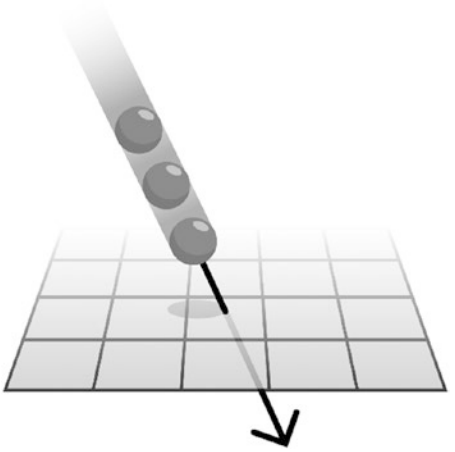

Figure 8-9. *Radiance*

The measures build on one another, as you can see by radiance $W/(m^2sr)$. The final one uses all the previous ones—power (W), solid angle (sr), and irradiance $W/(m^2)$. There are more units needed to measure light, but these should do for our purposes.

How to Represent a Material

Now we know what units we can use to measure light, so we can tackle how to represent a material for the purposes of rendering. Representing how light behaves when hitting a material is something that we generally try to do with as few parameters as possible.

All the lighting models we have looked at up to now (Lambert, Phong, and BlinnPhong) needed at the minimum the direction of the normal and the direction of the light. Lighting models that are view-dependent also at minimum require the view direction.

So far, we've used the expression *lighting model* to mean an implementation or a formula that simulates the behavior of light when hitting a certain surface. But there are more technical terms to describe the same concepts, depending on what subset of light behaviors you want to model with it, and we're now going to introduce them. You'll find that the common mathematical representation of a lighting model feels different than the ones we have implemented and looked at up to now.

Bidirectional Reflectance Distribution Function (BRDF)

A BRDF is a function that defines how light is reflected on a surface. All of the lighting models that we've implemented up to now are BRDFs, meaning that they mainly concern themselves with reflectance. This means they can't represent phenomena such as subsurface scattering or translucency.

They are functions that take as arguments the direction of incoming light and the direction of outgoing light, but they are in the form of spherical polar coordinates (see Figure 8-10). This coordinate system enables us to represent these directions with only four numbers. In this system, everything is measured in angles from the point of origin. In different reference material, you might find different symbols representing different angles (even switched around, just to make it more confusing) but in this book, we'll use θ for the azimuth angle and ϕ for the zenith angle. This is the convention followed by computer graphics literature.

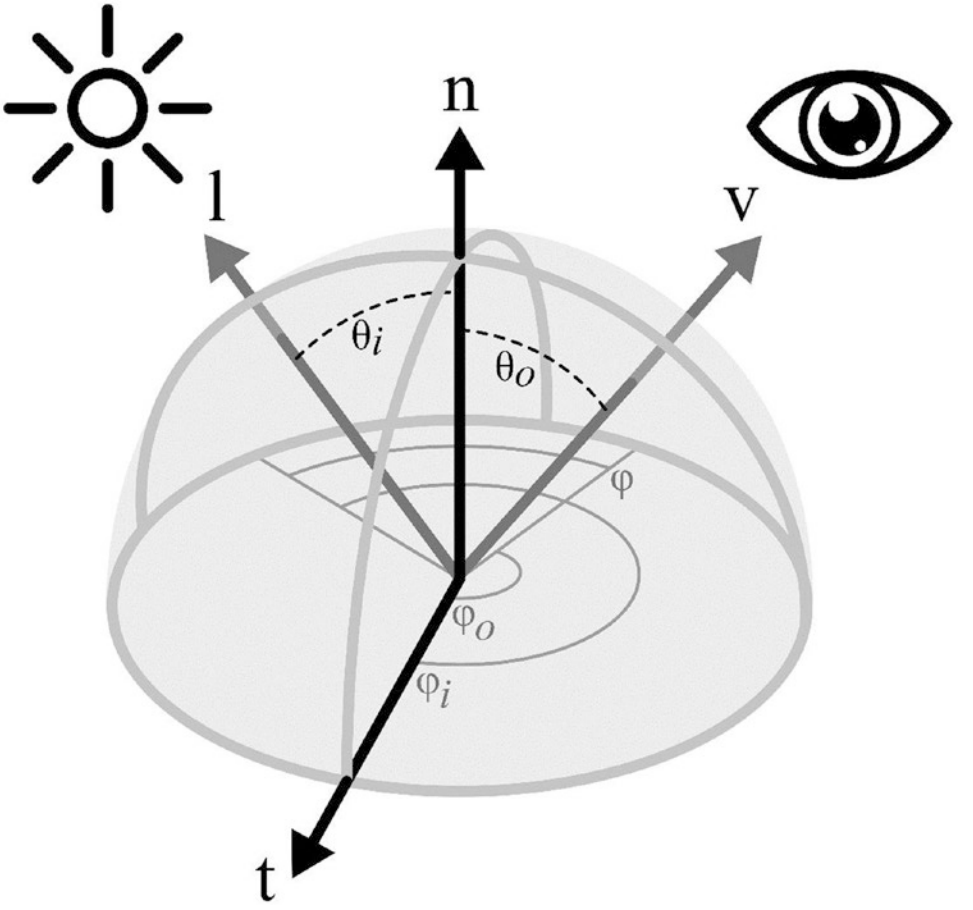

Figure 8-10. *Incoming light direction, view direction, and normal direction represented in spherical polar coordinates*

In Figure 8-10, there are four named angles. Out of those, θ_i and ϕ_i define the incoming light direction, while θ_0 and ϕ_0 define the outgoing light direction (or view direction). ϕ starts from the tangent vector **t**. We're going to use this kind of coordinates again in the future, in order to analyze the behavior of different BRDFs, so keep an eye out for them.

The original definition of BRDF is:

$$f_r(\omega_i,\omega_r)=\frac{dL_r(\omega_r)}{dE_i(\omega_i)}=\frac{dL_r(\omega_r)}{L_i(\omega_i)\cos\theta_i\omega_i}$$

As we showed in the previous section, L is radiance and E is irradiance. ω_i and ω_r are the incoming and outgoing (reflected) light directions.$cos\theta_i$ is the angle between the incoming light direction and the surface normal.

There are other types of functions, not as well defined as the BRDF, that can deal with subsurface scattering, etc. They are BSDF (Bidirectional Scattering Distribution Function) and BSSRDF (Bidirectional scattering-surface reflectance distribution function), but they are outside our real-time rendering scope.

There are some properties that a well behaved physically based BRDF must have, they are positivity, reciprocity, and energy conservation.

Positivity

It means that the BRDF cannot return a negative number, there is no negative light.

Reciprocity

The BRDF must have the same value when ω_i and ω_r are swapped.

Energy Conservation

In the simplest terms, the outgoing light cannot exceed the incoming light, unless the excess of light has been emitted from the object.

Of these three, we'll encounter energy conservation most, as it makes a lot of difference for the perceived realism of the BRDF result.

Microfacet Theory

Microfacet theory is a way to mathematically derive BRDFs to describe surface reflection, specifically for surfaces that have microsurface irregularities and are not completely smooth and flat at every scale. We assume that we have a surface with surface irregularities smaller than the scale of observation, but large compared to the wavelength of light. Each point is locally a perfect mirror; we're ignoring nanoscale irregularity.

Such perfect mirrors reflect light (ignoring refraction) into one outgoing direction, which depends on the light direction and the microfacet normal. For this light to get into our viewpoint, these microfacets need to be oriented so that they reflect into the direction of view.

To achieve that the direction of their normal has to be the half vector. The half vector is halfway between the direction of the light and the direction of the view. The half vector is obtained by adding `lightDir` and `viewDir`, and then normalizing the result.

Shadowing and masking (see Figure 8-11) stop some microfacets from contributing to the outgoing light. Shadowing is when irregularity creates a peak, which shadows some microfacets. Masking is when the reflection going out is blocked by a peak. In reality, this blocked light will continue to bounce, and might make it out of the surface eventually. But microfacet theory simplifies this away.

Figure 8-11. *Respectively shadowing and masking approximations used by the theory, as opposed to reality (light bounces)*

From these assumptions, we can derive this equation, which describes a microfacet specular BRDF:

$$f(l,v) = \frac{F(l,h)G(l,v,h)D(h)}{4(n \cdot l)(n \cdot v)}$$

It's useful to break it up into components, discussed next.

Fresnel

$$F(l,h)$$

This is the amount of light that's reflected rather than refracted, for a given substance, based on the light angle and normal. We explained Fresnel earlier, and this shows you where its place is within an actual BRDF.

Normal Distribution Function

$$D(h)$$

It tells us how many of the microfacets are pointing in the half vector direction, as those are the only ones we care about, because they contribute outgoing light. The distribution function determines the size and shape of the highlights. Some NDFs that you're likely to encounter are GGX, BlinnPhong, and Beckmann.

Geometry Function

$$G(l,v,h)$$

This tells us how many facets are not rendered useless by shadowing and masking. It is necessary for the BRDF to be energy conserving. Often, Smith's shadowing function is used as a geometry function.

Microfacet theory is very useful, but it can't simulate all phenomena. It ignores diffraction and interference, which depend on the nature of light as a wave.

The Rendering Equation (Part II)

In the first chapter, I showed you the rendering equation. I didn't do it to scare you off, I promise! It was to get you used to seeing it. Now we're going to map each part of the equation to what it does.

$$L_0(x,\omega_0) = L_e(x,\omega_0) + \int_\Omega f(x,\omega_i \to \omega_0) L_i(x,\omega_i)(\omega_i \cdot n) dw$$

First, we have outgoing light:

$$L_o(x, \omega_0)$$

This is what we want to obtain from the equation.
Then we have emitted light:

$$L_e(x, \omega_0)$$

Which is the light emitted by the surface we are shading. It's out of the BRDF, because it's not reflected or refracted light, and out of the integral, so it's only added once.
Then we have the integral:

$$\int_\Omega [...] dw$$

Which is going to repeat and add up everything within it, for each solid angle in the hemisphere (dw)
Within the integral, we have the BRDF:

$$f(x,\omega_i \to \omega_0)$$

It takes the incoming light ω_i and returns the reflected and refracted light ω_0.
Then we have the incoming light:

$$L_i(x, \omega_i)$$

This is the incoming light to the point on the surface that we're shading.
Then we have the normal attenuation:

$$(\omega_i \cdot n)$$

Which you know as $n \cdot l$ from earlier lighting model implementations.

Hacks Real-Time Rendering Needs

The integral part of the rendering equation cannot yet be executed in real-time. As mentioned multiple times, indirect light is simulated with many techniques—spherical harmonics, global illumination, lightmapping, and image-based lighting (reflection probes, cubemaps, etc.).

One important point to know is that if you implement a new lighting model, you may need to change how global illumination is calculated and how cubemaps are processed.

With reflection probes, a cubemap is captured from the scene, and to use image based lighting, the cubemap needs to be processed according to the BRDF that you're using. In practice, whatever processing is built-in within Unity may be sufficiently compatible for it not to be worth changing with a formulation for your BRDF.

There are open source tools that can be modified and used to process cubemaps for your custom BRDF, such as cmftStudio and the old CubeMapGen. You likely could also implement this processing as a compute shader within Unity.

HDR and Tone Mapping

When your camera is not HDR, the scale of your light values cannot be realistic. You definitely want to use HDR camera and RenderTextures as much as possible.

The consequence of that is that unless you're only targeting HDR screens, at some point you need to convert those HDR values back to LDR. This is done with tone mapping as a post-processing effect. Tone mapping can influence the style of your game, so choosing the best tone mapping operator is important. You could think of it as what film stock does to a movie. The light that is captured to film (which was HDR) has been converted analogically to LDR, with patterns specific to the composition of the film stock.

Linear Color Space

Have you ever wondered about what space you're doing your calculations in? Chances are that after starting this book, you have, more than once. An obvious problem with rendering that apparently nobody noticed for a long time, is that we were doing our calculations in a non-Linear Space.

The colors shown on the LDR screens are in Gamma Space. Gamma Space is a non-Linear Space where the midpoint between 0 and 255 is not 128, but 187 (see Figure 8-12). Hence all calculations involving colors were subtly wrong. The shift to using Linear Space is another staple of physically based shading. It's relatively easy to take care of, especially on a platform for which Unity supports Linear Space outright.

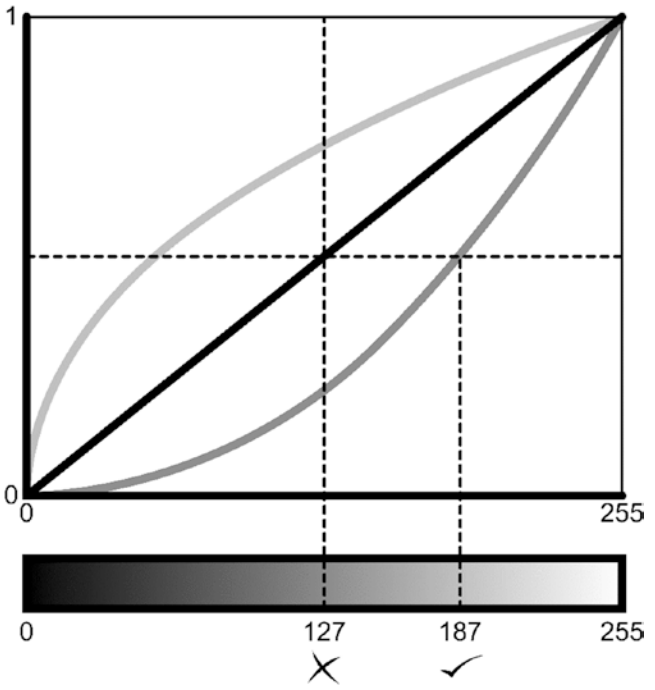

Figure 8-12. *Linear and Gamma color spaces middle points compared*

Why Is Physically Based Shading Useful?

At this point, you might wonder if physically based shading (PBS) is worth all the fuss and all the math. If you desire realism in your graphics, PBS has no comparison.

While this may not sound interesting for you if you want to implement some artistic shaders that don't necessarily need any realism at all, PBR has other advantages. For example, using a physically based lighting model, material's settings for a scene are supposed to behave well with any lighting setup, drastically reducing the amount of fiddling artists need to do when changing the lighting of a scene. You can still stylize your rendering, either by using post-processing or by tweaking the lighting model.

Summary

This chapter briefly covered most of the physical concepts you need to know in order to understand how physically based shading actually works. Hopefully this wasn't too overwhelming. You may feel like a lot of concepts are bouncing about in your head, but don't worry, it's almost impossible to take this in all at once. Unless you have previously studied the physics of light in some depth, you're bound to need more time. Some BRDF implementations in the next chapters will help you put into practice the concepts from this chapter.

Next

The next chapter shows you how to create your first physically based lighting model in an Unity Surface shader, putting what you've learned into practice.

CHAPTER 9

Making a Shader Physically Based

Among the many things you learned in the last chapter, there are the three rules your implemented BRDFs need to follow In order to be considered physically based. One way to put this knowledge into practice is to review the custom Phong surface shader, to find whether it breaks those rules. The shader followed the original formulation of Phong, which is fairly old, so it's quite likely.

Analyzing Phong

Listing 9-1 shows the custom Phong lighting functions. Let's check for positivity, reciprocity, and energy conservation.

Listing 9-1. The Phong Custom Lighting Functions

```
inline fixed4 LightingPhong (SurfaceOutput s, half3 viewDir, UnityGI gi)
{
    UnityLight light = gi.light;

    float nl = max(0.0f, dot(s.Normal, light.dir));
    float3 diffuseTerm = nl * s.Albedo.rgb * light.color;

    float3 reflectionDirection = reflect(-light.dir, s.Normal);
    float3 specularDot = max(0.0, dot(viewDir, reflectionDirection));
    float3 specular = pow(specularDot, _Shininess);
    float3 specularTerm = specular * _SpecColor.rgb * light.color.rgb;

    float3 finalColor = diffuseTerm.rgb + specularTerm;

    fixed4 c;
    c.rgb = finalColor;
    c.a = s.Alpha;

    #ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
       c.rgb += s.Albedo * gi.indirect.diffuse;
    #endif

    return c;
}
```


C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_9

Checking for Positivity

You can make sure that the BRDF output is always positive, with the little trick of putting a max function starting from 0 in front of the final color. For the purposes of this chapter that'd be okay, but generally it's better if the BRDF mathematically guarantees that.

Checking for Reciprocity

Reciprocity is not easy to check from the shader code. In general, any BRDF published as physically based should have this property. Dig into the research papers on the BRDF if you want to check that, or alternatively if you're handy with programs such as Mathematica, or even with hand calculations, you can do the math and check for yourself. Phong is not reciprocal, so this needs some work.

Checking for Energy Conservation

With the right normalization factor, even Phong can be energy conserving. An energy conserving specular will be brighter when it's concentrated in a smaller area, and dimmer when it's spread around more. Generally, this would correspond with how rough the material is, in a BRDF based on microfacet theory.

The Modified Phong

Luckily for us, the work of making Phong physically based was done by Lafortune and Willems, in their paper from 1994. The normalization factor we want is:

$$\frac{n+2}{2\pi}$$

You can see the modified, normalized Phong function implemented in Listing 9-2.

Listing 9-2. Modified Phong as a Custom Lighting Function

```
inline fixed4 LightingPhongModified (SurfaceOutput s, half3 viewDir, UnityGI gi)
{
    const float PI = 3.14159265358979323846;
    UnityLight light = gi.light;

    float nl = max(0.0f, dot(s.Normal, light.dir));
    float3 diffuseTerm = nl * s.Albedo.rgb * light.color;

    float norm = (n + 2) / (2 * PI);
    float3 reflectionDirection = reflect(-light.dir, s.Normal);
    float3 specularDot = max(0.0, dot(viewDir, reflectionDirection));
    float3 specular = norm * pow(specularDot, _Shininess);
    float3 specularTerm = specular * _SpecColor.rgb * light.color.rgb;

    float3 finalColor = diffuseTerm.rgb + specularTerm;

    fixed4 c;
    c.rgb = finalColor;
```

```
    c.a = s.Alpha;

    #ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
       c.rgb += s.Albedo * gi.indirect.diffuse;
    #endif

    return c;
}
```

Et voila; we have implemented a physically based BRDF. Possibly that was a bit anticlimactic. It boiled down to multiplying the specular for the *normalization factor*. The normalization factor prevents the BRDF from returning more light than it received in the first place.

You'll get to play with the interesting BRDFs later on, but for now let's worry about checking whether it feels physically based when you look at it, and learning how you can make a convenient shader. This custom lighting function fits well inside the old custom Phong surface shader from a few chapters ago (see Listing 9-3).

Listing 9-3. The Custom/Modified Phong Surface Shader

```
Shader "Custom/ModifiedPhong" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _SpecColor ("Specular Material Color", Color) = (1,1,1,1)
        _Shininess ("Shininess (n)", Range(1,1000)) = 100
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        #pragma surface surf PhongModified fullforwardshadows

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        half _Shininess;
        fixed4 _Color;

        inline void LightingPhongModified_GI (
            SurfaceOutput s,
            UnityGIInput data,
            inout UnityGI gi)
        {
            gi = UnityGlobalIllumination (data, 1.0, s.Normal);
        }
```

```
        inline fixed4 LightingPhongModified (SurfaceOutput s, half3 viewDir, UnityGI gi)
        {
            const float PI = 3.14159265358979323846;
            UnityLight light = gi.light;

            float nl = max(0.0f, dot(s.Normal, light.dir));
            float3 diffuseTerm = nl * s.Albedo.rgb * light.color;

            float norm = (_Shininess + 2) / (2 * PI);
            float3 reflectionDirection = reflect(-light.dir, s.Normal);
            float3 specularDot = max(0.0, dot(viewDir, reflectionDirection));
            float3 specular = norm * pow(specularDot, _Shininess);
            float3 specularTerm = specular * _SpecColor.rgb * light.color.rgb;

            float3 finalColor = diffuseTerm.rgb + specularTerm;

            fixed4 c;
            c.rgb = finalColor;
            c.a = s.Alpha;

            #ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
            c.rgb += s.Albedo * gi.indirect.diffuse;
            #endif

            return c;
        }

        UNITY_INSTANCING_CBUFFER_START(Props)
        UNITY_INSTANCING_CBUFFER_END

        void surf (Input IN, inout SurfaceOutput o) {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            o.Specular = _Shininess;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

What you are looking for is a reduction of specular brightness when the specular is spread out, and an increase of specular brightness when it's concentrated on a small surface. Since we have a pre-normalization version of Phong lying around, we can check that easily. As you can see in Figure 9-1, the original Phong (on the right) keeps the same brightness, while the normalized Phong (on the left) is less bright when the specular occupies a bigger area and brighter when it occupies a smaller one, as energy conservation should do.

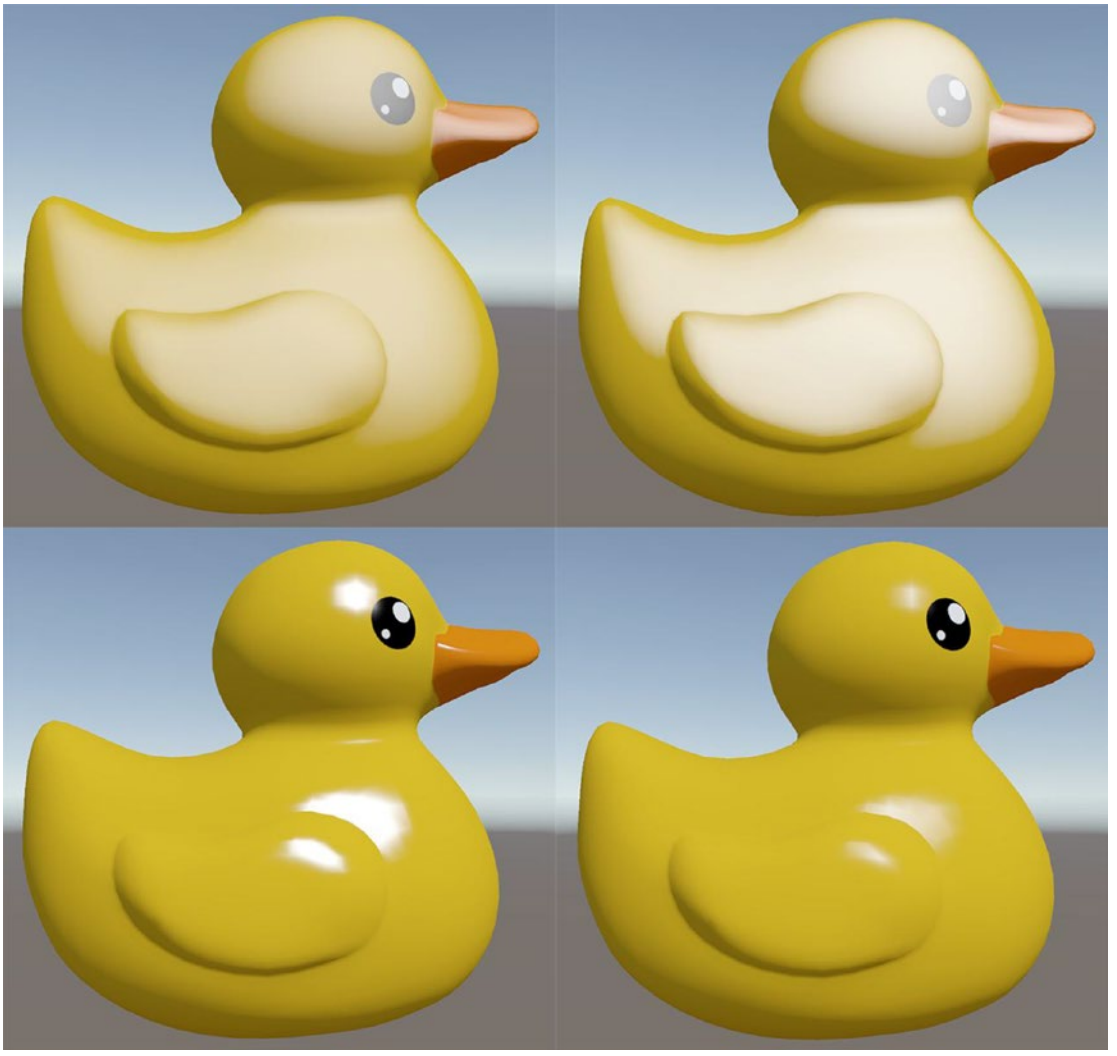

Figure 9-1. *The left side is the normalized Phong and the right is the plain one; first row is n=1, second row is n=80*

Another handy way to see the difference is to plot the graph of the BRDF's behavior comparatively. You will see more about that in Chapter 11, where we introduce a program to analyze BRDFs, but Figure 9-2 shows a little sneak peek into the future.

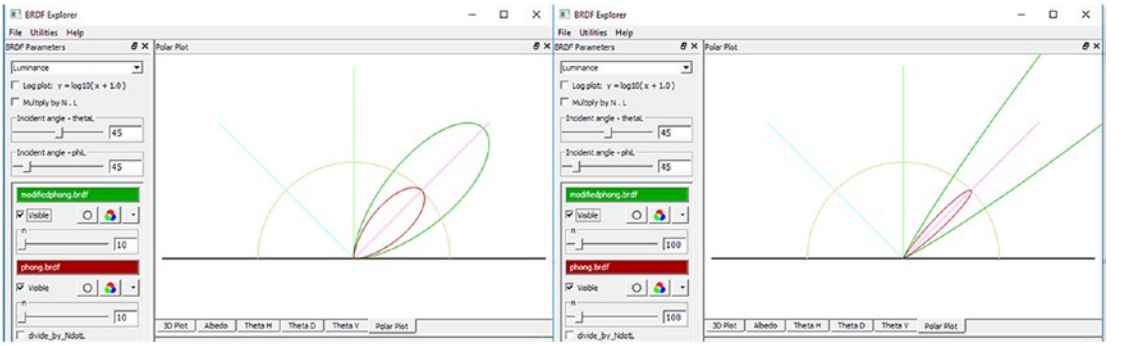

Figure 9-2. *Polar plot comparison of original Phong and modified Phong*

Figure 9-2 shows two versions of Phong, plotted on a graph in comparison to each other. On the left side of the image, both have n = 10. On the right side of the image, which corresponds to n = 100, you're seeing the modified Phong brightness increasing drastically, while the original Phong's brightness stays the same.

Summary

As you saw, it's possible to make even an old-fashioned BRDF more physically based. The same result can be achieved with BlinnPhong, by looking up normalization factors or by deriving them yourself. Deriving them yourself will require you to be handy with Calculus.

While this modified Phong is energy conserving, it still doesn't take Fresnel into account, so there will still be quite a bit of difference between this Phong and a BRDF derived from microfacet theory.

Next

Now that you have a somewhat physically based BRDF in your arsenal, you're ready to learn about post processing effects. They are going to help the shader seem more realistic. You'll need to use HDR cameras, ensure you're shading in linear space, and add tone mapping.

CHAPTER 10

Post-Processing Effects

As mentioned in Chapter 8, post-processing effects are an essential part of obtaining physically based shading. The need to use HDR makes it a must to have tone mapping, which is best implemented as a post-processing effect. When you have one post effect, you might as well add a few others in, such depth of field or bloom.

While the new official Post Effects system available on the Unity Asset Store is very powerful, you might still need to implement your own tone mapping, or some other post effect that Unity does not include, so it makes sense to get at least an overview of how to develop an image effect. Also keep in mind that post-processing effects before Unity 2017 used to be called image effects, if you encounter that term.

The new post-processing stack is evolving fast, and since any detail I provide about that is going to be obsolete fast, I'm going to stick to showing you how to make a post-process effect from scratch, without relying on that stack. That said, the new stack looks great, with movie-level color grading capabilities.

How Post-Processing Effects Work

A *post-processing effect* is a basically an Image Effect shader, which is applied to every pixel in the current screen. Imagine you have rendered your scene, not to the screen but to a separate buffer, which in Unity is called a `RenderTexture`.

Now you can either send that to be visualized by the screen or manipulate it. To manipulate it, you can access it as a texture within the post-processing shader. This shader is executed separately from the other ones in the scene, because it needs to be executed when the scene has just been rendered. To trigger that, there is a function signature you can use in a script attached to the camera you want to apply the post effect to.

Why Post-Processing Effects Are Useful

The most important use of post effects for physically based shading is for *tone mapping*. You should always use an HDR camera for better realism, and then it would need to be tone-mapped back to LDR before being shown on an LDR screen. Another effect that does a lot for realism, especially for skin and other subsurface-based materials, is depth of field.

For platforms where Unity doesn't support a linear color space, post effects are a great way to implement that with a minimum of fuss. Unity recently added support for a linear color space for mobile, but before, if you wanted to execute your calculations in linear space, you would need to first convert any color input to your shader (color picker, textures, etc.) to the linear space, do the shading calculations, and then convert back the entire rendered image to gamma as the last action in your post effects.

It might still be useful for you to know how to do that, so we'll cover it.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_10

Setting Up a Post Effect

You need three things to set up a post effect:

- A camera in the scene to apply it to
- A script to add to that camera as a component
- A shader that the component will execute

HDR and Linear Setup

First, check that your camera is using HDR. As shown in Figure 10-1, the Allow HDR option must be on. Notice also our rendering path is Forward. In Chapter 1, we talked about the different types of renderers available in Unity, and in general in the game industry. As of Unity 2017.2, the available choice of renderers is between Forward and Deferred, and two legacy options, VertexLit and an older Deferred.

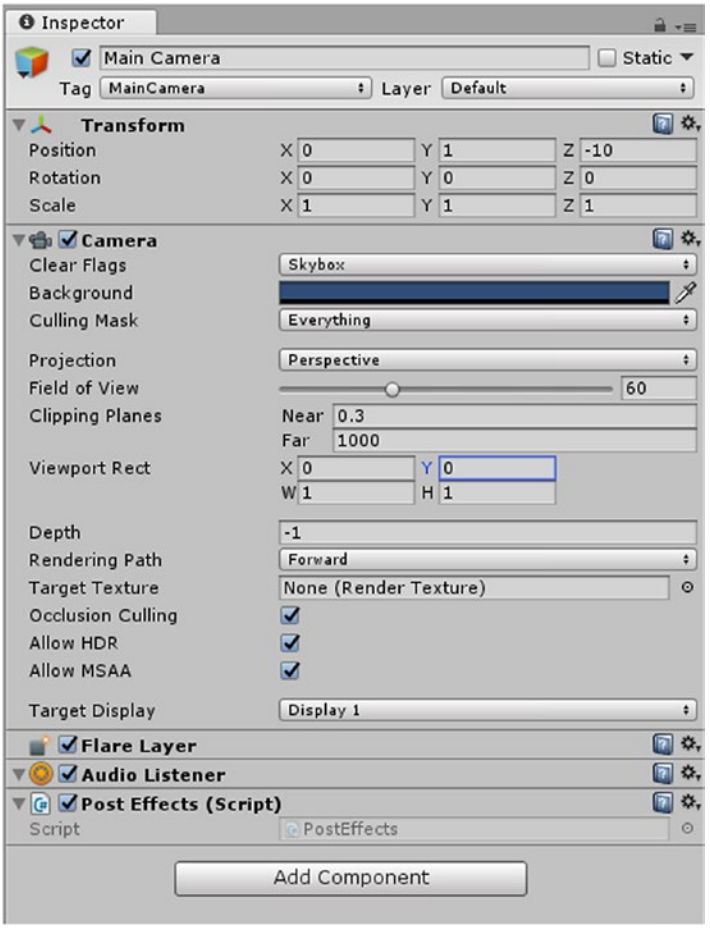

Figure 10-1. *HDR camera settings*

In the future, they are probably going to be joined by the new `ScriptableRenderLoop`, a super flexible renderer that's currently in development.

Our shaders work with the Unity Forward renderer, so we're going to stick to that. Deferred renderers can be limiting when developing complex BRDFs, as the decision of what data is made available to shaders has been set in stone when the renderer was being developed.

Make sure that in your Player Settings, the Color Space is set to Linear. To check that, open Build Settings, then Player Settings, then Other Settings and there you'll find the Color Space, as shown in Figure 10-2.

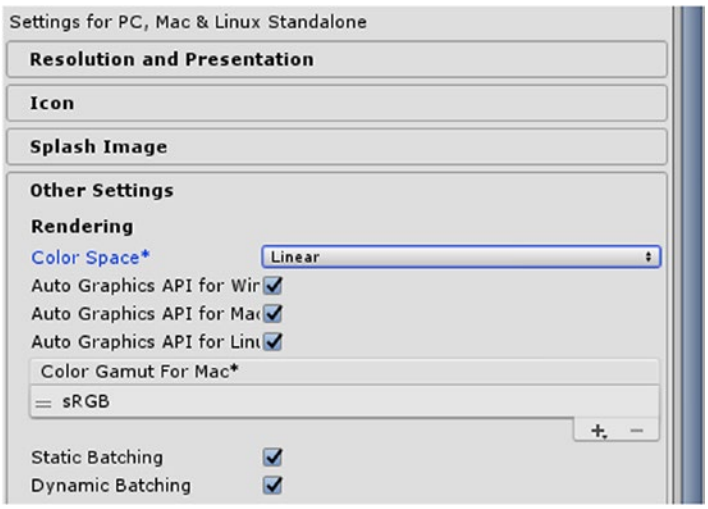

Figure 10-2. *Color Space settings*

Of these two, Color Space is the more important one, so if you need to choose only one, choose Linear. That said, HDR also helps achieving realism, so seriously consider rendering your game using HDR cameras.

Script Setup

Let's create a new post effect script by selecting the camera and clicking on the Add Component button in the inspector. Scroll to New Script, select C# as the language, and name the new script `PostEffects`. The script that's created should be the same as in Listing 10-1.

Listing 10-1. Default C# Script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PostEffects : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }
```

```
    // Update is called once per frame
    void Update () {

    }
}
```

It's the classic empty MonoBehaviour based script. Let's start customizing it to work as an applier of post effects. First we'll add a line that prevents us from adding this script as a component to a GameObject that's not a camera:

```
[RequireComponent (typeof (Camera))]
[ExecuteInEditMode]
```

The second line makes the script execute in edit mode as well; otherwise it'd be tricky to preview our changes without playing. Then we should add a few members to the class, plus a getter method. We need a reference to the Image Effect shader that the script is going to use, plus we need to create a material for it dynamically (see Listing 10-2).

Listing 10-2. Members and Getter to Add to the Script

```
public Shader curShader;
private Material curMaterial;

Material material
{
    get
    {
        if (curMaterial == null){
            curMaterial = new Material(curShader);
            curMaterial.hideFlags = HideFlags.HideAndDontSave;
        }
        return curMaterial;
    }
}
```

To make the script more convenient, we can fill the appropriate shader in automatically (finding it by name), making sure that the script doesn't error out if image effects are not supported or if the shader is not supported. We can do that by adding lines to the Start function, as shown in Listing 10-3.

Listing 10-3. Complete Start Function

```
void Start () {
    curShader = Shader.Find("Hidden/PostEffects");
    GetComponent<Camera>().allowHDR = true;
    if(!SystemInfo.supportsImageEffects){
        enabled = false;
        Debug.Log("not supported");
        return;
    }
```

```
    if (!curShader && !curShader.isSupported){
        enabled = false;
        Debug.Log("not supported");
    }
    GetComponent<Camera>().depthTextureMode = DepthTextureMode.Depth;
}
```

Note that we're also forcing the camera to compute a depth texture. A depth texture is useful for many effects, such as depth of field, but in this case, I only want to show you what it looks like and how to get to it.

Other administrative tasks include destroying the material when the `Camera` GameObject is disabled, and returning early from the `Update` function if the camera is not enabled (see Listing 10-4). The latter is necessary when you want to change the values for your Image Effects shader within the `Update` function.

Listing 10-4. Update and OnDisable

```
void Update () {
    if (!GetComponent<Camera>().enabled)
    return;
}

void OnDisable(){
    if(curMaterial){
        DestroyImmediate(curMaterial);
    }
}
```

Finally, we get to where the magic happens. `OnDisable`, `Start`, and `Awake` are methods triggered by the engine at the appropriate times.

To apply our effects, we need another of those functions, `OnRenderImage`:

```
void OnRenderImage(RenderTexture source, RenderTexture destination).
```

As you can see, it takes two arguments, a source RenderTexture and a destination one. You're supposed to put the code that actually applies the effect in this method. There is also another way, using two similar methods—`void OnPreRender()` and `void OnPostRender()`. In this case, you need to create the source render texture yourself, which depending on your platform, could be more efficient. We'll cover both methods. Let's start with `OnRenderImage`. There are a series of steps needed:

- Get the current rendered scene in a RenderTexture (`RenderTexture source` does that for us).
- Use `Graphics.Blit` to apply the image effect shader to the source texture. `Blit` means copying all pixels from an origin surface to a destination surface, while optionally applying some transformation.
- That also includes a destination RenderTexture. If that is null, Blit will send the result directly to the screen.

To put this into practice, let's make an Image Effect shader the usual way (right-click in the Project pane and then choose Create ➤ Shader ➤ Image Effect Shader). Call this shader `PostEffects`. The result is shown in Listing 10-5.

Listing 10-5. Default Image Effect Shader

```
Shader "Hidden/PostEffects"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        // No culling or depth
        Cull Off ZWrite Off ZTest Always

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
            };

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = v.uv;
                return o;
            }

            sampler2D _MainTex;

            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 col = tex2D(_MainTex, i.uv);
                // just invert the colors
                col = 1 - col;
                return col;
            }
```

```
            ENDCG
        }
    }
}
```

This looks mostly like an Unlit shader, except that by default the path is under `Hidden`, and it declared `Cull` and `Zwrite` off, and `ZTest Always`. We haven't previously messed with these values, because we haven't needed to. They are needed because we're not shading a model, we're processing a 2D image that's going to be shown on-screen as two triangles, forming a quad. We mustn't cull the back face, and Zwrite doesn't serve any purpose, because there is no depth.

The main texture is actually our rendered scene. Apply this shader in `OnRenderImage` and take a look at the result. We already have the source texture and a destination texture (which is the screen), so we only need a few lines of code to apply the effect (see Listing 10-6).

Listing 10-6. OnRenderImage First Version

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture) {
    if (curShader != null)
    {
        Graphics.Blit(sourceTexture, destTexture, material, 0);
    }
}
```

`Graphics.Blit` takes the source texture, the destination texture, the material that contains the shader to apply, and which pass to use, starting from 0. The result is shown in Figure 10-3.

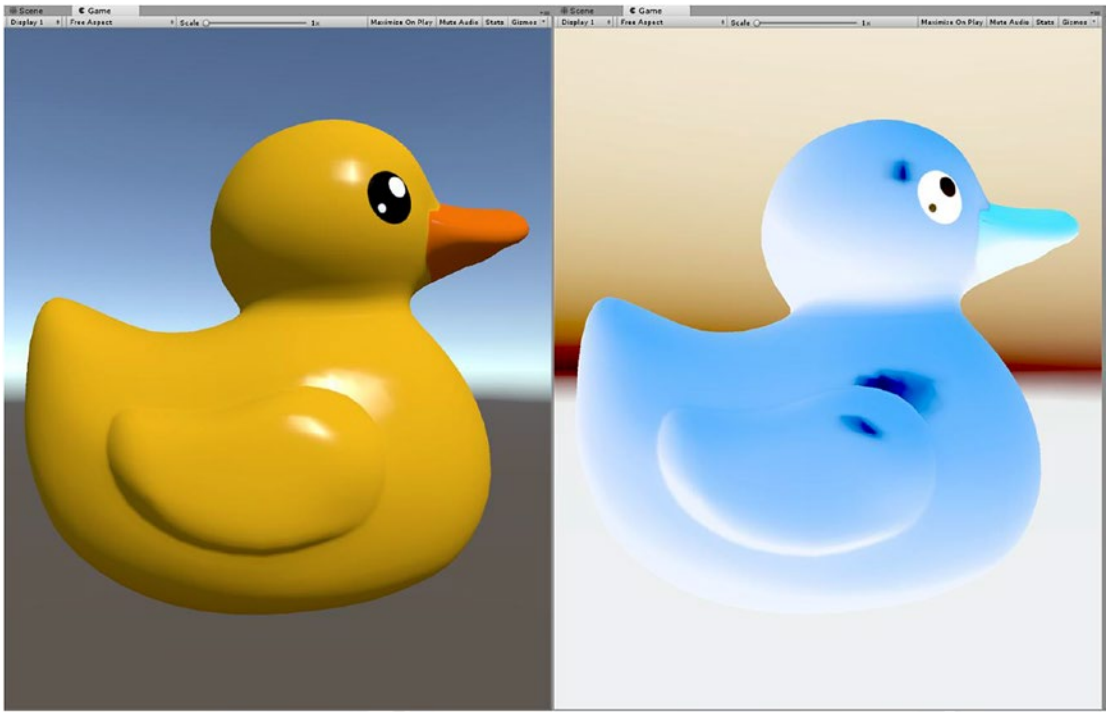

Figure 10-3. *The scene without the post effect (left), and the scene with post effect applied (right)*

As you can see, the default post effect inverts the scene colors. Let's copy and paste this pass and change the content of the frag function to show the depth texture of the camera. We can add a name within the pass, and we need to move the Cull Off line to the top of each pass. To access the depth texture for the camera we need to declare the variable with the right name (it's a convention) and then convert it to a grayscale color using a macro (Listing 10-7).

Listing 10-7. Decoding the Camera's Depth Texture

```
sampler2D _CameraDepthTexture;

fixed4 frag (v2f i) : SV_Target
{
    fixed depth = UNITY_SAMPLE_DEPTH( tex2D(_CameraDepthTexture, i.uv) );
    fixed4 col = fixed4(depth,depth,depth, 1.0);
    return col;
}
```

The complete pass will look like Listing 10-8.

Listing 10-8. An Image Effect Pass Showing the Camera Depth

```
Pass
    {
        name "DebugDepth"
        Cull Off ZWrite Off ZTest Always Lighting Off
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"

        struct appdata {
            float4 vertex : POSITION;
            float2 uv : TEXCOORD0;
        };

        struct v2f {
            float2 uv : TEXCOORD0;
            float4 vertex : SV_POSITION;
        };

        v2f vert (appdata v) {
            v2f o;
            o.vertex = UnityObjectToClipPos(v.vertex);
            o.uv = v.uv;
            return o;
        }

        sampler2D _CameraDepthTexture;
```

```
        fixed4 frag (v2f i) : SV_Target {
            fixed depth = UNITY_SAMPLE_DEPTH( tex2D(_CameraDepthTexture, i.uv) );
            fixed4 col = fixed4(depth,depth,depth, 1.0);
            return col;
        }
        ENDCG
    }
```

The pass you want to apply has changed, so you need to change the `Blit` line accordingly:

```
Graphics.Blit(sourceTexture, destTexture, material, 1);
```

You're applying the second pass in the shader. To show off the result, I reduced the camera far plane to 12 (the bigger the depth, the less precise it is) and added a few spheres farther back (see Figure 10-4).

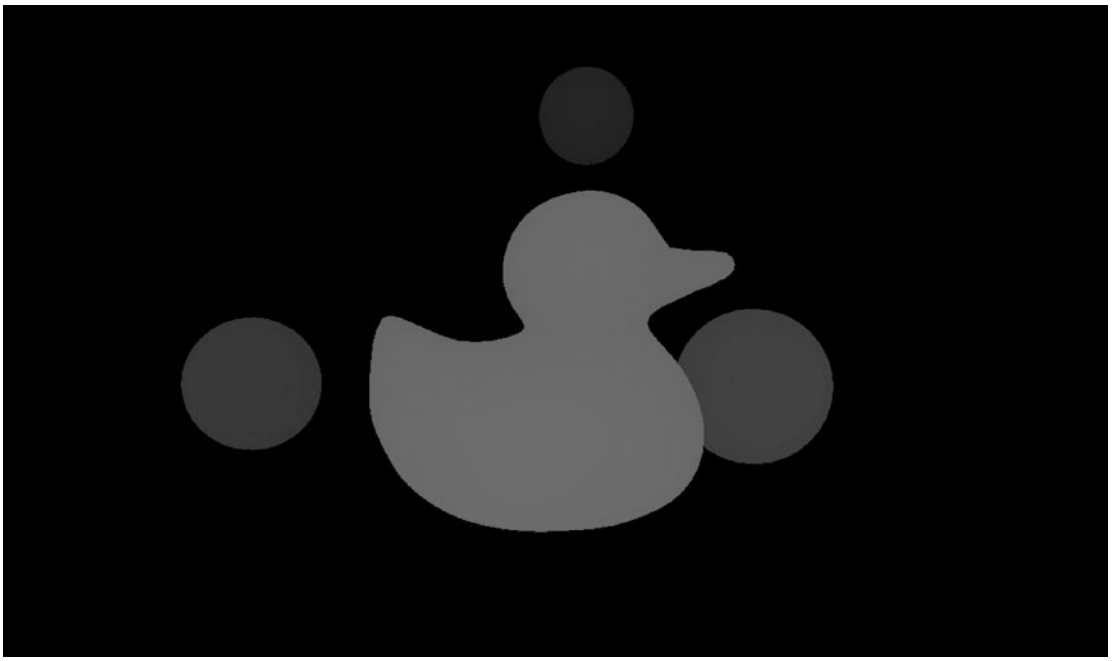

Figure 10-4. *The camera depth image effect*

Let's add a few options to the script, so you can switch which image effect pass is being applied. We need to add two booleans, one for the invert effect, and one for the depth effect. Inside OnRenderImage, we'll check which one is one and act accordingly. If both are on, invert will win. If none are on, the scene won't have any post effect applied (see Listing 10-9).

Listing 10-9. Switching Between Different Passes, According to bool Properties Visible in the Inspector

```
public bool InvertEffect;
public bool DepthEffect;

void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture) {
    if (curShader != null)
    {
        if (InvertEffect) {
            Graphics.Blit(sourceTexture, destTexture, material, 0);
        } else if (DepthEffect) {
            Graphics.Blit(sourceTexture, destTexture, material, 1);
        } else {
            Graphics.Blit(sourceTexture, destTexture);
        }
    }
}
```

Now you can switch between effects, from the inspector.

Conversion to Linear

To try this out, you'll need to switch the color space back to gamma in the player settings. For this to be completely correct, we'd need to make a shader for the duck as well, which converts any color within it to linear space. But that would be too long; we can still see the difference between an image effect calculated in linear space and one calculated in Gamma Space even without doing that.

Add another boolean to the members and call it `LinearInvertEffect`. Add another to the chain, and add another pass to the image effect shader. Within that pass, the `_MainTex` should be sampled within a power to 2.2. Then the invert is applied, and then the color should be elevated to the power of $\frac{1}{2.2}$ (Listing 10-10).

Listing 10-10. An Effect Calculated in Linear Space Within a Gamma Space Project

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = pow(tex2D( _MainTex, i.uv ), 2.2);
    col = 1 - col;
    return pow(col, 1/2.2);
}
```

Even though we're not doing much at all, you can still see a different result (within a project in gamma space, keep in mind) when the calculation of the effect is done in linear space (see Figure 10-5).

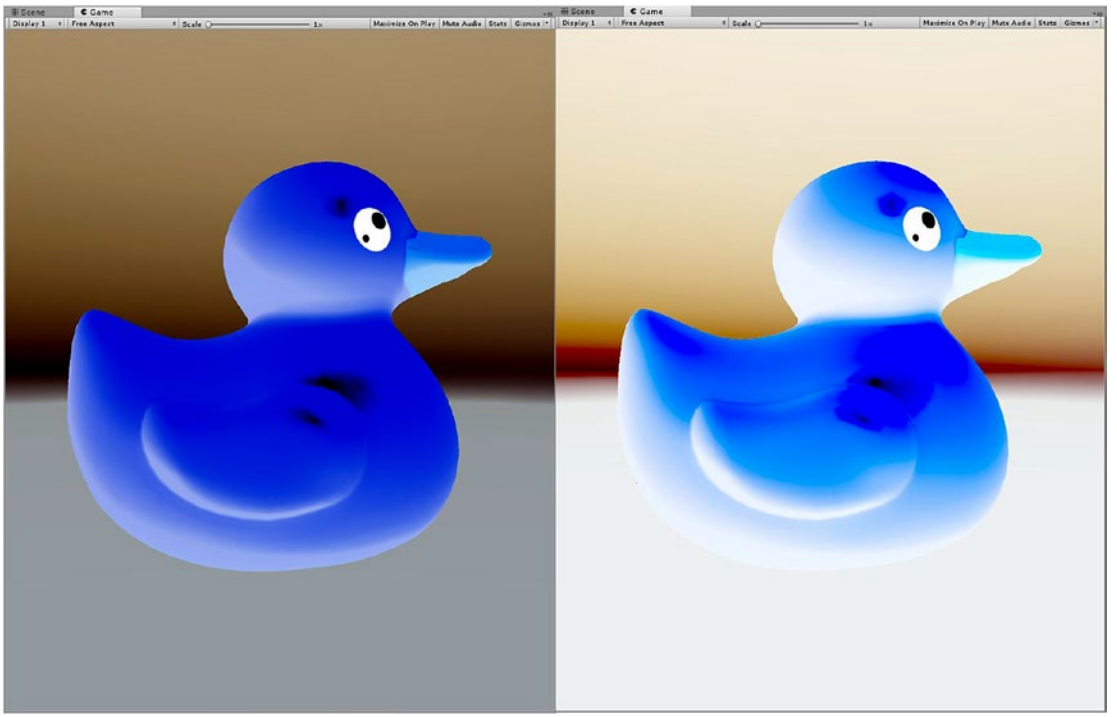

Figure 10-5. *Left: gamma space invert effect, right: linear space invert effect*

RenderTextures Brief Overview

RenderTextures are textures you can write to. You can set one as the target of a camera. You can create them from the UI, as assets, or you can create them programmatically.

When you do create them programmatically, it's important to remember to release them. One way you can do that is by getting a temporary `RenderTexture`:

```
RenderTexture.GetTemporary(512,512,24,RenderTextureFormat.DefaultHDR);
```

After you use it, call

```
RenderTexture.ReleaseTemporary(someRenderTex);
```

Up to now, we have only considered post effects that are not consecutive, as they don't use the result of the previous pass. Some effects that use it are blur and depth of field. In that case, you need to use temporary textures to pass to the next shader pass as arguments. This is outside of our scope in this book, because you'll most likely use the Unity post-processing stacks, anyway, to do this kind of effect.

As mentioned at the beginning of the chapter, there is an alternative way without using `OnRenderImage`, and it works as shown in Listing 10-11.

Listing 10-11. Alternative Way of Applying a Post Effect

```
RenderTexture aRenderTex;
void OnPreRender()
{
    aRenderTex = RenderTexture.GetTemporary(width,height,bitDepth, textureFormat);
    camera.targetTexture = myRenderTexture;
}
void OnPostRender()
{
    camera.targetTexture = null;
    Graphics.Blit(aRenderTex,null as RenderTexture, material, passNumber);
    RenderTexture.ReleaseTemporary(aRenderTex);
}
```

This technique may be quicker, depending on your platform. You can see within it a use of `GetTemporary` and `ReleaseTemporary`.

A Simple Tone Mapper

Tone mapping, as mentioned in Chapter 8, is a way to gracefully convert an HDR buffer to an LDR buffer. The idea is to map the HDR values to LDR values in aesthetically pleasing ways, instead of just clipping them.

There are many tone-mapping operators that you can use, but we'll stick to a relatively simple one, the one that John Hable invented for Uncharted 2, which he published on his blog for everyone to use.

First, we need to add a new boolean, a new `if` statement, and a new shader pass. We also need to add another property, an exposure for the camera. To obtain a slider in the inspector, you should use this code:

```
[Range(1.0f, 10.0f)]
public float ToneMapperExposure = 2.0f;
```

After declaring it, we want to send the value to the shader when the effect is active. For that we should use `material.setFloat`. We're going to put that within the `if` chain:

```
[...]
} else if (ToneMappingEffect) {
   material.SetFloat("_ToneMapperExposure", ToneMapperExposure);
   Graphics.Blit(sourceTexture, destTexture, material, 3);
} else {
[...]
```

Then we need to declare the property within the shader and declare the variable only within the pass where we're going to use it. Then, we implement Hable's operator in the fragment shader (see Listing 10-12).

Listing 10-12. The Hable Tone Mapper Operator

```
float _ToneMapperExposure;

float3 hableOperator(float3 col)
{
    float A = 0.15;
    float B = 0.50;
```

```
    float C = 0.10;
    float D = 0.20;
    float E = 0.02;
    float F = 0.30;
    return ((col * (col * A + B * C) + D * E) / (col * (col * A + B) + D * F)) - E / F;
}

fixed4 frag (v2f i) : SV_Target
{
    float4 col = tex2D(_MainTex, i.uv);
    float3 toneMapped = col * _ToneMapperExposure * 4;
    toneMapped = hableOperator(toneMapped) / hableOperator(11.2);
    return float4(toneMapped, 1.0);
}
```

The code for this shader has become very long, so I can't just paste it here for you to peruse. You should check that your final result is correct by downloading the book source code and looking at the scene called `chapter10-posteffects.unity`.

At this point, if you check the boolean that activates the tone mapper pass, you should be able to change the exposure slider in the inspector and see it change the result. Turning on and off the tone mapper should show you how it influences the final result (see Figure 10-6).

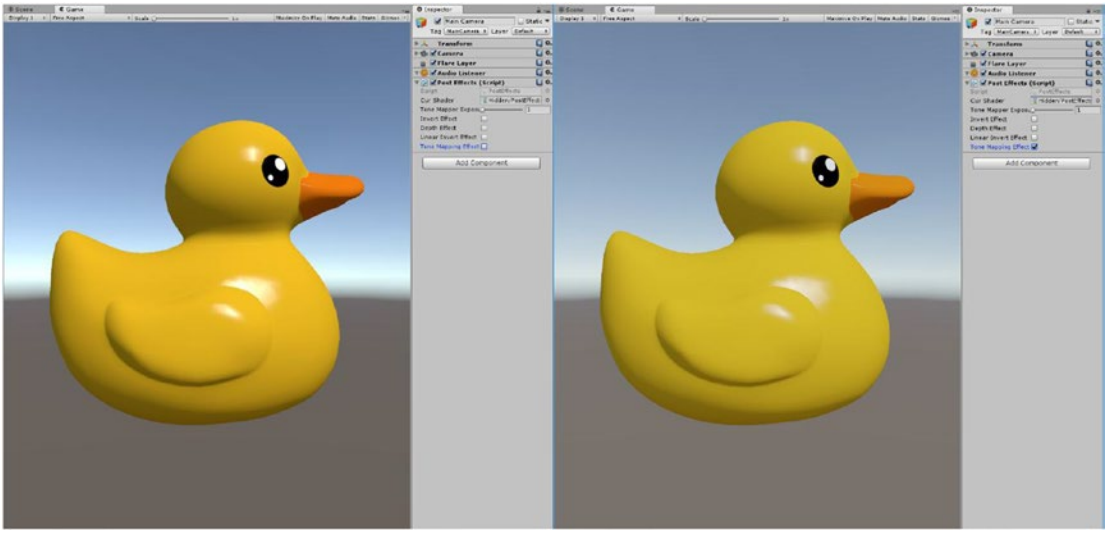

Figure 10-6. *Left, no tonemapping; right, hable tonemapping with exposure 1.0*

Now you have some idea about how tone mapping is implemented. There are various tone-mapping operators documented online, although you'll most likely going to want to use the powerful post-processing stack for this, which I'm going to briefly introduce next.

Post-Processing Stack v1

This processing stack is available on the Asset Store as of August 2017. It's the first version of the new post-processing stack and it gives you a number of pre-made effects that you can turn on and off and customize.

To obtain it, download it from the asset store and attach it to a camera as a component. You'll have to create a post-processing settings file (see Figure 10-7) where you can turn on and off different effects. If you're using HDR (as you should, if performance allows) then definitely turn on the Color Grading effect, which includes the tone mapper. Also consider the depth of field, as it helps with the perception of realism.

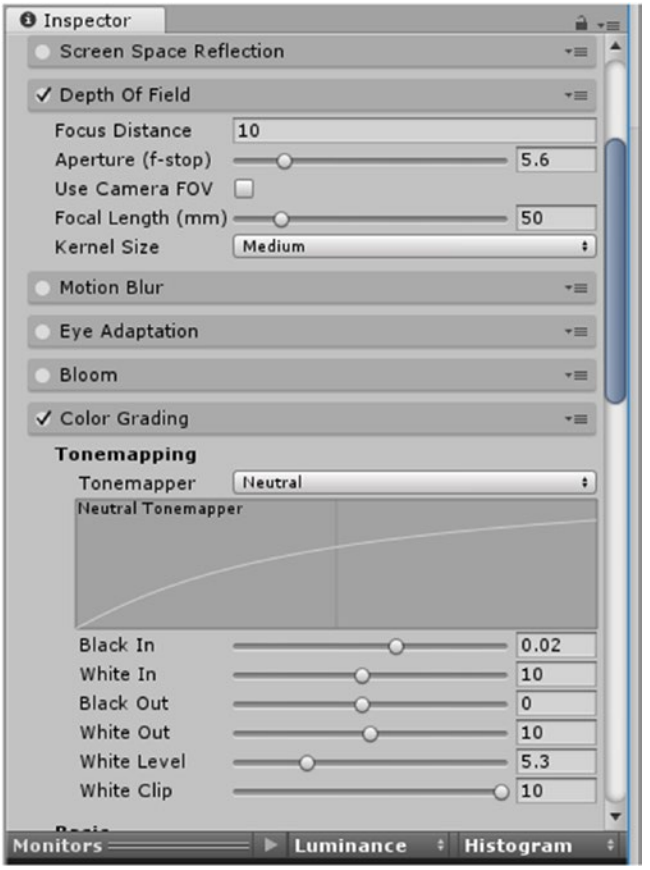

Figure 10-7. *Available options with the v1 of the post-processing stack*

Post-Processing Stack v2

This processing stack is available on GitHub as of August 2017. It's the second version of the new processing stack, and it adds more flexibility. Some ways to switch post-effects setup depending on various triggers, scriptability, ability to add your own effects, and compatibility with the upcoming rendering scriptable loop.

To get it, you might need to clone the v2 branch of the GitHub project into your Assets folder. The setup for this stack is more complex and might well change, so you should read the GitHub wiki, or the instructions within Unity's manual. You'll need to create a profile for this one as well and again consider using the included tone mapper and depth of field (see Figure 10-8).

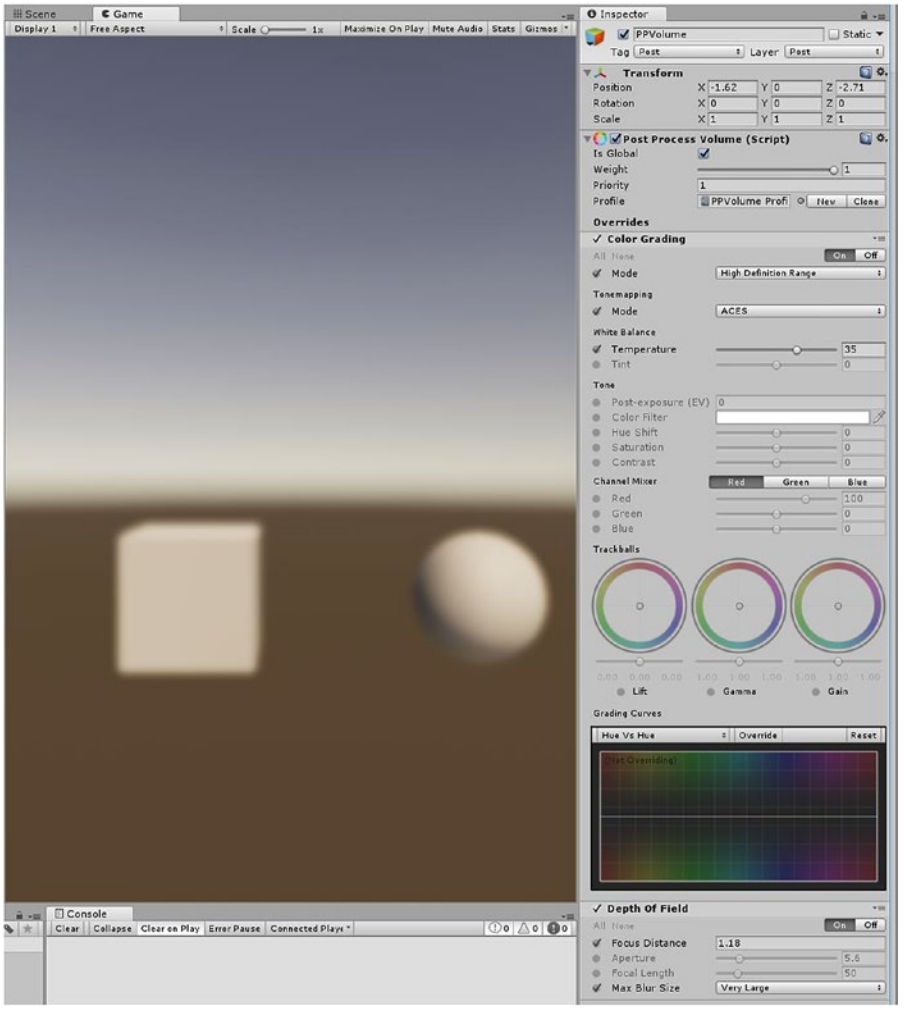

Figure 10-8. *The v2 stack, applied to a simple scene*

Summary

This chapter implemented various post effects from scratch, using C# scripts and Image Effect shaders. We introduced RenderTextures and we implemented a simple tone mapper. We also introduced the two post effects stack that Unity has developed, which you should likely use for your tone mapping and color grading needs.

Next

The next chapter presents a number of popular BRDFs that you may want to consider implementing. We'll introduce a tool to explore them and visualize their differences in many ways.

CHAPTER 11

BRDFs Who's Who

Now that you're familiar with the principles of physically based shading and how to ensure your BRDF implementation respects them, we can explore some of the BRDFs described in past papers.

You can invent your own from scratch, for example by deriving them from the microfacet theory model, or you can choose one out of the many BRDFs that have been described in graphics papers in the past.

BRDF Explorer

Since implementing a BRDF in Unity is a fair bit of work, to explore the many available BRDFs we're going to use a different tool. This tool has been developed by the researchers from Disney, and it's called BRDF explorer (see Figure 11-1). It's open source on GitHub, you can find the source code at https://github.com/wdas/brdf, and the precompiled version is at https://www.disneyanimation.com/technology/brdf.html.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_11

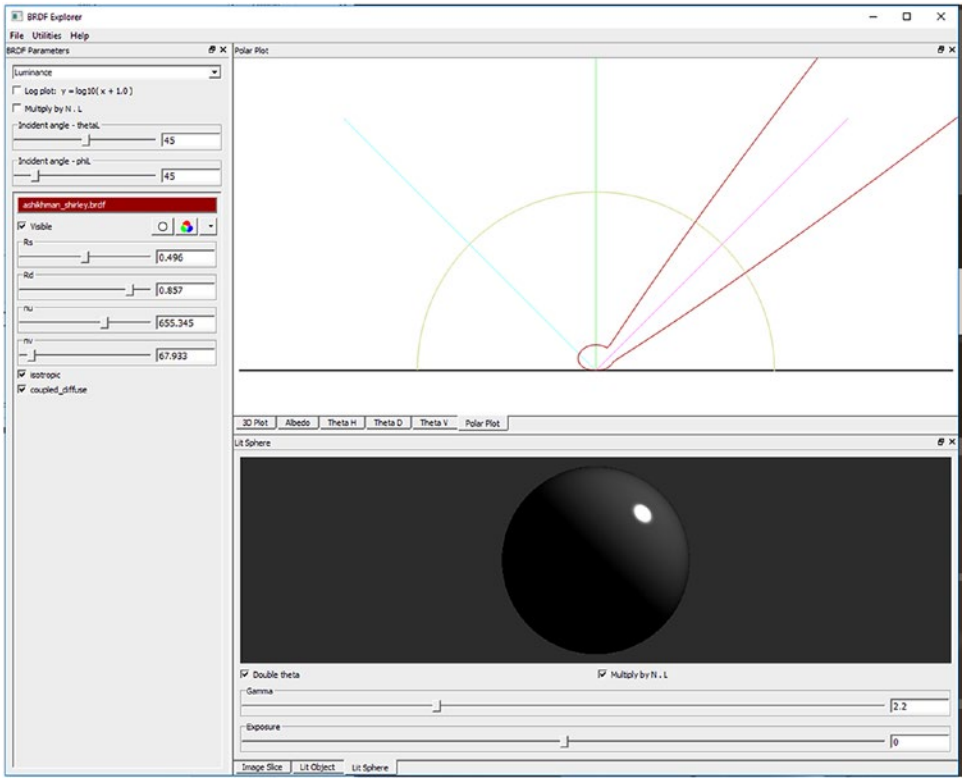

Figure 11-1. *BRDF Explorer analyzing a BRDF*

BRDF Explorer comes with many different BRDF implementations and many ways to analyze them. Those include graphs that extract relevant information out of them, both real-time and offline rendered BRDF preview. You can also compare many different BRDFs at the same time. Sometimes graphics research papers include implementations for BRDF Explorer of the technique they describe. All in all, a very useful tool, and well worth learning.

As you know from the chapter on the theory of physically based shading, there are common parameters between BRDFs (such as incoming light direction) that allow you to analyze the behavior of a BRDF in respect to those parameters and to compare them as well. BRDF Explorer is explicitly developed with the objective of making their access and comparison convenient. The program comes with a number of `.brdf` files that are included in the `/brdfs` subfolder. Each of them implements a BRDF using a custom dialect of the GLSL shader language.

BRDF Parameterizations

You might remember Figure 11-2 from the chapter about physically based shading. These angles, and others, are going to be important to read BRDF Explorer's output. This is one of the most common parameterizations of a BRDF: it uses four angles to describe two vectors, the incoming and outgoing light vectors.

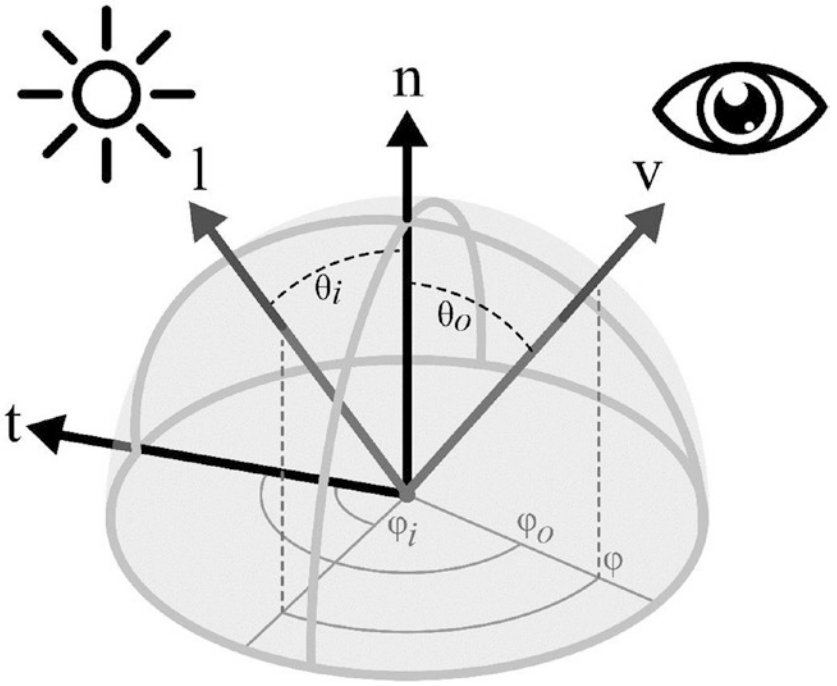

Figure 11-2. *Original BRDF parameterization in spherical polar coordinates*

BRDF Explorer was created to compare BRDFs acquired from real-life materials. There is a small, freely available database of such BRDFs, called MERL. This database uses a slightly different coordinate system compared to the one in Figure 11-2. It's still a spherical polar coordinate system, but instead of being based on the incoming and outgoing light vectors, it's based on two vectors derived from them. They are the *halfway vector* (or half-angle) and the *difference vector* (see Figure 11-3). This is often referred to as an alternative BRDF parameterization or a reparameterization.

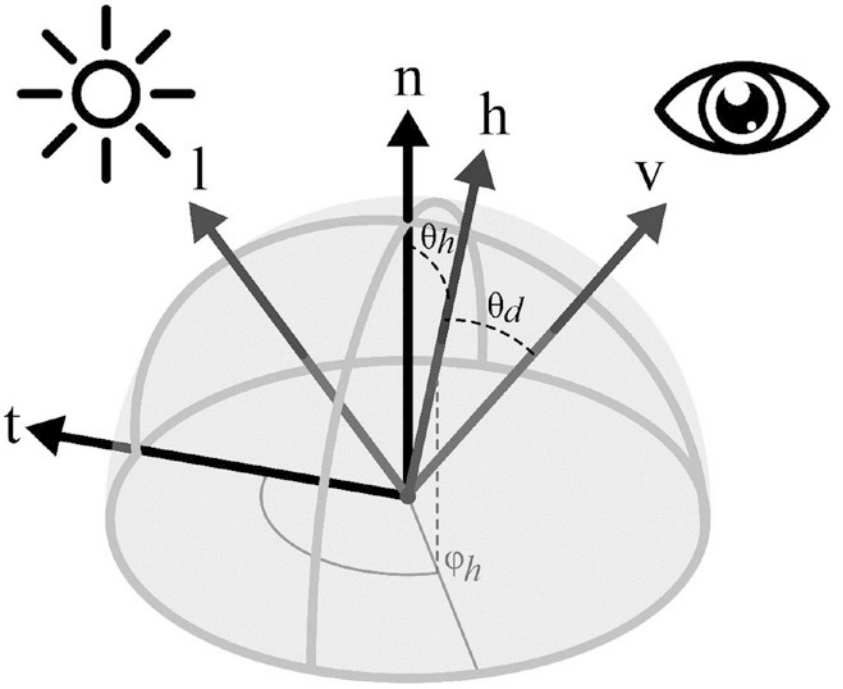

Figure 11-3. *BRDF reparameterization using the half vector and the difference vector*

The reason of this difference is because a BRDF parameterized on these angles can store the acquired data more efficiently. Still, isotropic BRDFs occupy about 33MB. *Isotropic* means that, when you rotate around the normal, the reflections won't change. In the case of this database, only including isotropic BRDFs allows them to store one less angle from the captured BRDF, which would have been angle ϕ_h.

The half-angle is obtained by adding the incoming and outgoing light vectors, while the difference angle is found in more complex ways. One way is to find the rotation that will align the halfway vector to the direction (0,0,1) and apply it to the incoming light vector.

One consequence of this change in the coordinate system is that the hemisphere is not centered around the normal vector anymore, so part of the acquired data goes under the hemisphere and will show up as black. It's something that will show up when you load a MERL BRDF and play with some of the various levers available in BRDF Explorer.

Now let's start digging into BRDF Explorer's functionality to see how useful it is.

Reading BRDF Explorer's Output

Let's get started by examining a BRDF we're quite familiar with: Phong. There are multiple versions of Phong included in the BRDF Explorer repository; we're going to look at the plain one for now.

Phong

First, we need to open the `phong.brdf` file, which is in the `brdf/` folder under the root of the program. It will add itself to the BRDF Parameters column on the left. From there, you can control which BRDF is currently shown, manipulate the BRDF's inputs, and make the BRDF invisible, all the while keeping it in the column (see Figure 11-4).

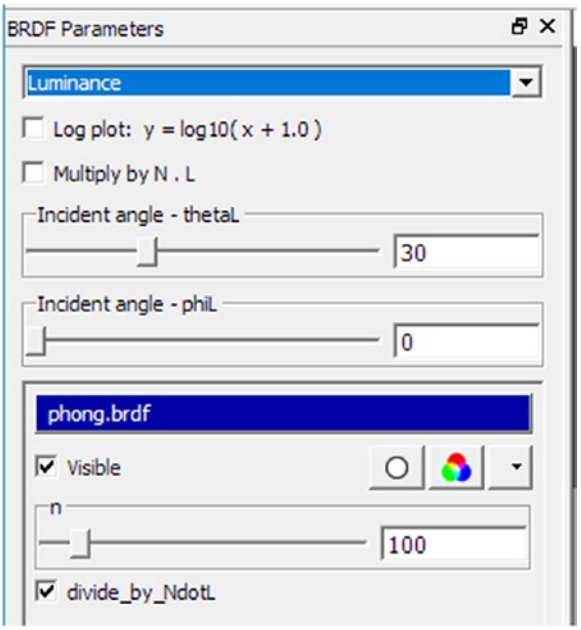

Figure 11-4. *BRDF Parameters pane*

At the top of the column, you can change the `thetaL` and the `phiL` angles, which you may better recognize as θ_i and ϕ_i. These are the two spherical polar coordinates angles that represent the incoming light direction.

The plane has quite a few options, so let's go through the most important options, one by one:

- Incoming light above the plane, the angle θ_l slider, from 0 to 90 degrees
- Incoming light on the plane, the angle ϕ_l slider, from 0 to 360 degrees

These two sliders specify the position of the incoming light. Changes will show up in real time in the 3D Plot tab (see Figure 11-5).

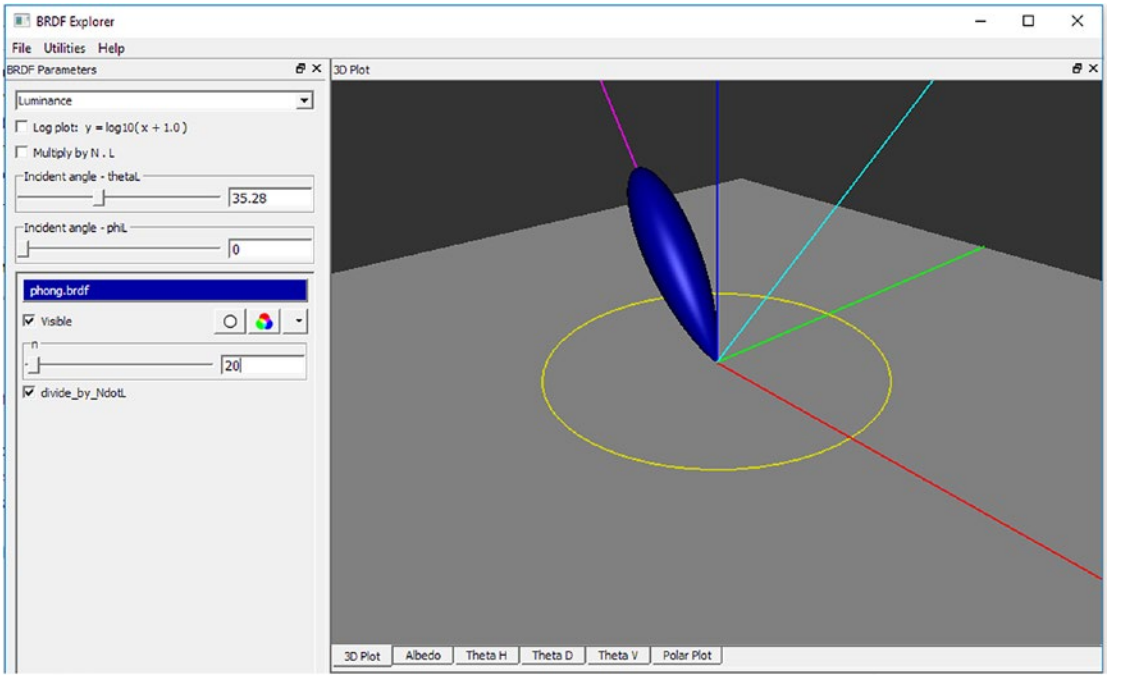

Figure 11-5. *The 3D Plot panel*

In the `phong.brdf` box, you'll find:

- The Visible check box, which turns the BRDF on and off
- The n value, which is the specular exponent, from 0 to 1000
- The divide by *N*·*L* checkbox

If you look at the 3D Plot panel while changing all these options, you'll get a good idea of the effect they have on the behavior of the BRDF. The 3D Plot panel shows the hemisphere with the incoming and outgoing light vectors, and the shapes of the BRDF. For example, Phong has one Specular Lobe, of which you might notice the resemblance with the specular approximation figure in Chapter 5. In some BRDFs, there is also a Diffuse Lobe, which also looks similar to the diffuse approximation figure from Chapter 5.

Looking at the code for `phong.brdf`, notice the parameters declarations that automatically produce sliders and other inputs within the interface. While the code is not in Cg, GLSL has a lot in common with it. You can create your own `.brdf` files or modify the existing ones, in order to try your theories out before delving into implementation in Unity.

Another useful panel is the Lit Sphere panel (see Figure 11-6). It shows you a real-time preview of the BRDF rendered on a sphere. Again, any changes to the parameters are shown in real time.

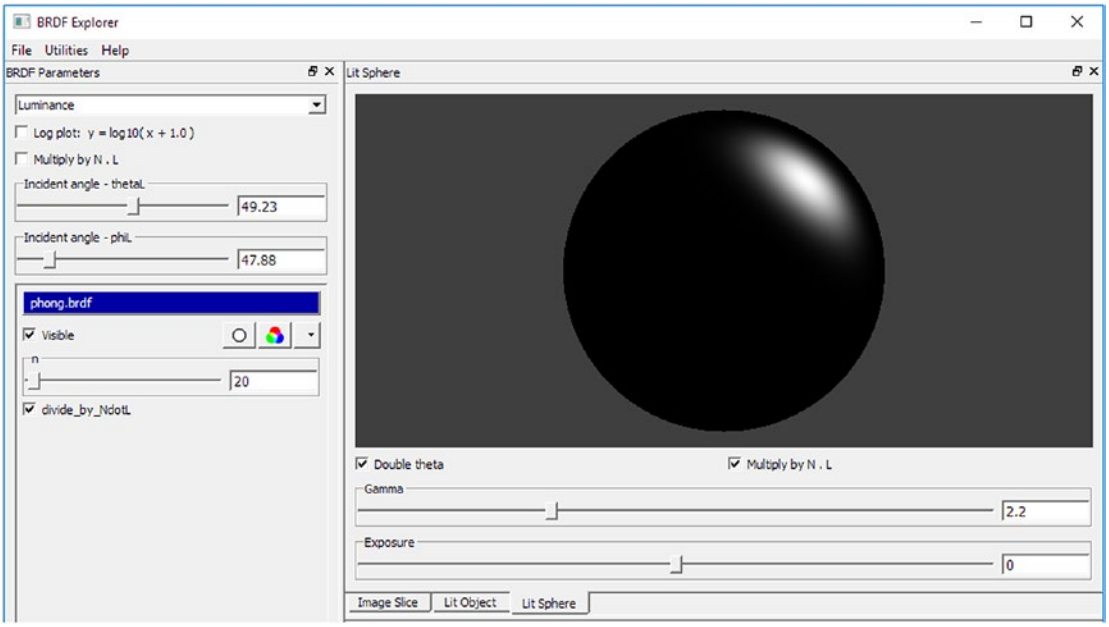

Figure 11-6. *The Lit Sphere panel*

If you want to test the BRDF in a more interesting setting, the Lit Object panel is what you want to use (see Figure 11-7). It renders with an offline renderer and supports image based lighting. You can import your own models (it supports the Obj format) and IBS cubemaps. Check the Keep Sampling checkbox to obtain a clear render, eventually.

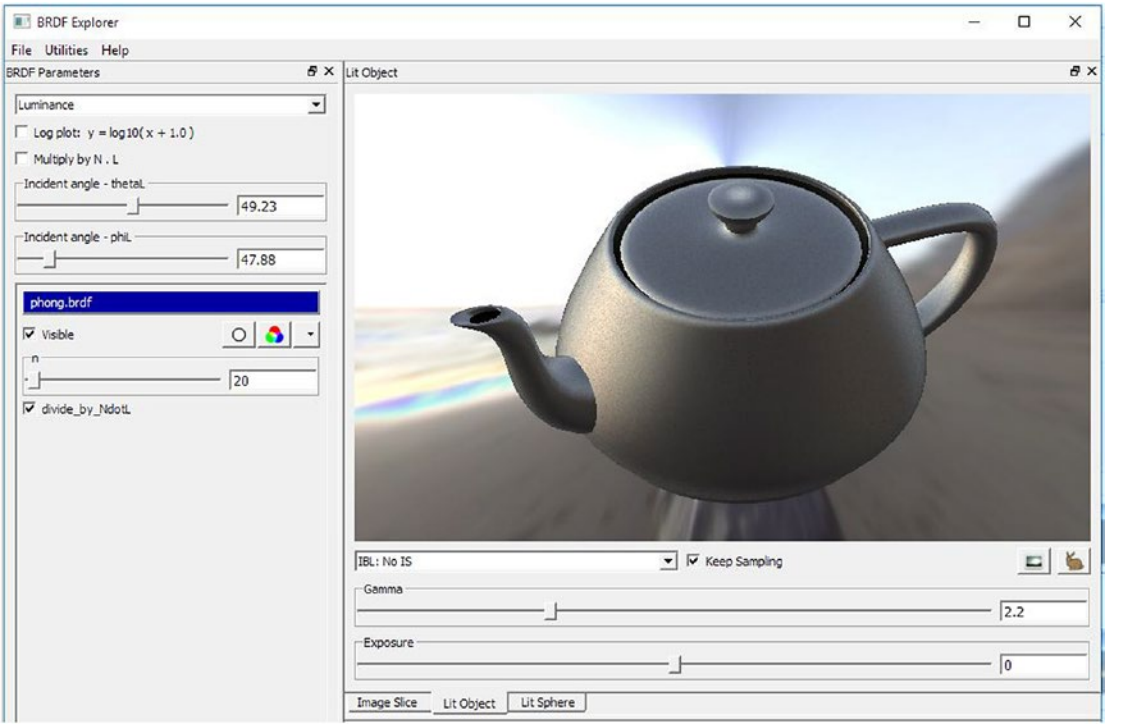

Figure 11-7. *The Lit Object panel*

Since we're using a different parameterization of BRDFs, there are a few extra angles involved. BRDF Explorer allows you to look at graphs of their behavior:

- Theta H (θ_h)has θ_d as a parameter
- Theta D (θ_d)has θ_h as a parameter
- Theta V (θ_v)has ϕ_v as a parameter

A general overview of all those values can be gleaned from the Image Slice panel (see Figure 11-8). The Image Slice panel is especially useful for comparing at a glance the behavior of BRDFs, included the ones acquired from a BRDF capture. The most informative slice corresponds to $\phi_d = 90$, with ϕ_d being the azimuthal rotation of the incoming light angle around the half vector.

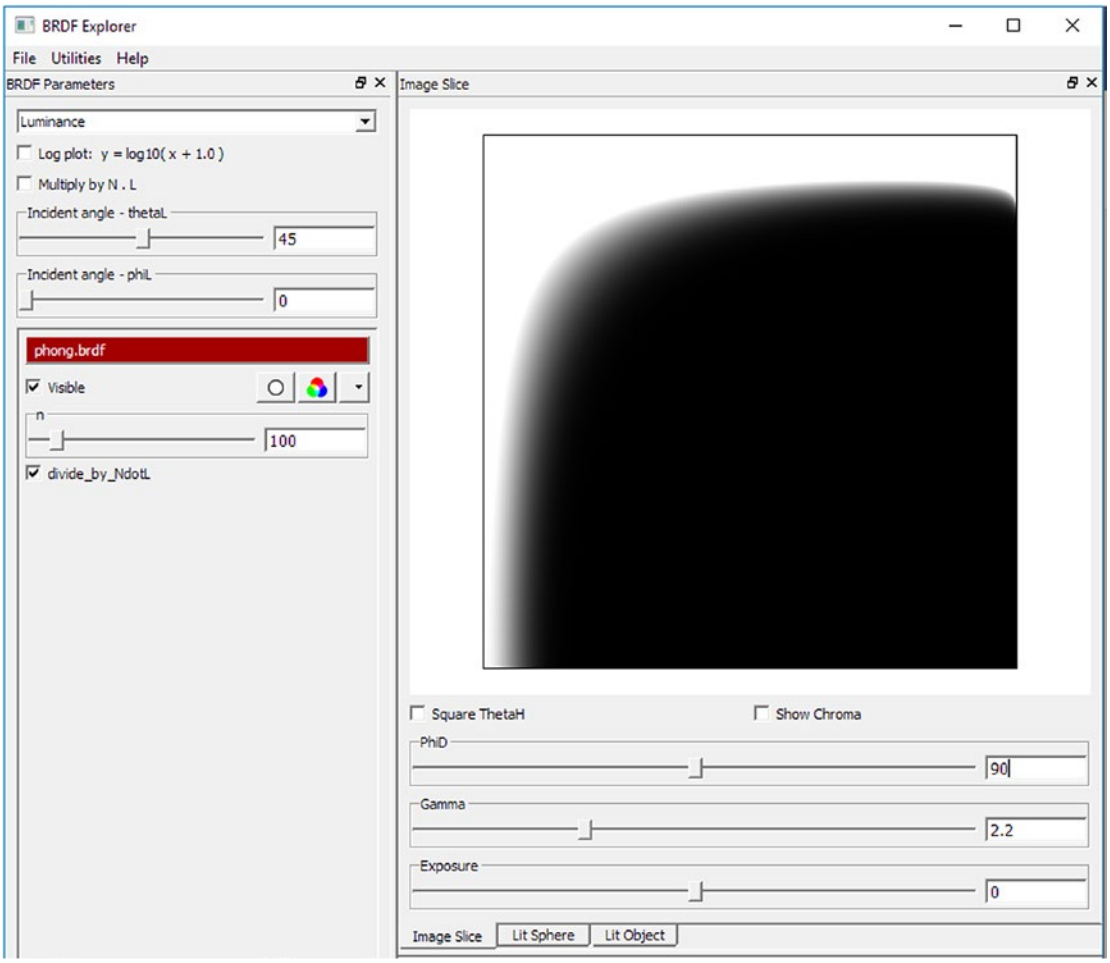

Figure 11-8. *The Image Slice panel*

Different parts of the image correspond to different characteristics of the BRDF, such as Fresnel reflection, specular reflection, grazing reflection, and retro-reflection (see Figure 11-9). The Image Slice panel is particularly useful for comparing the binary BRDFs acquired from real life with other BRDFs, either binary or implemented in code. As explained in Chapter 8, BRDFs cannot express all light phenomena, so these acquired BRDFs might not show some aspects that the original material had, such as iridescence.

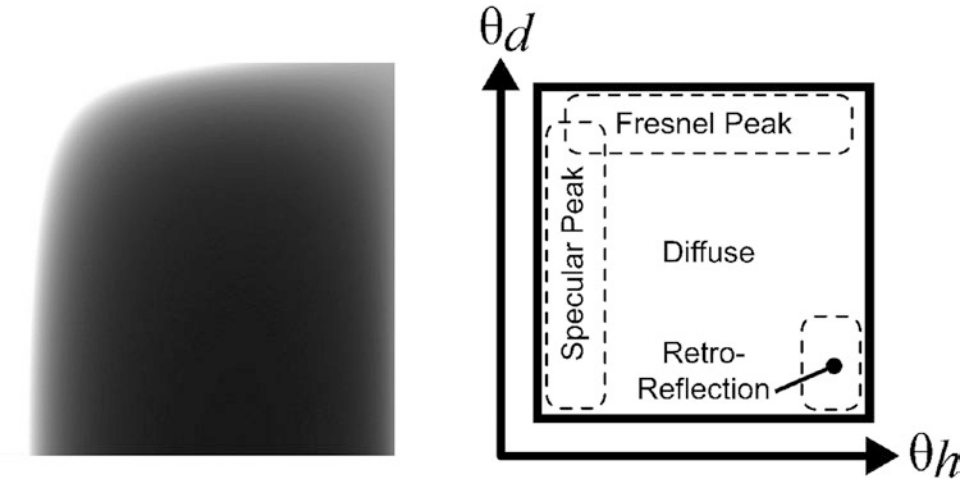

Figure 11-9. *A BRDF image slice (left), and how to read it (right)*

That's it for the Phong analysis.

MERL Database

The MERL database contains 100 captured BRDFs. You can download all of them from `https://www.merl.com/brdf/` as `.binary` files. BRDF Explorer can read `.binary` files, so you can explore these BRDFs as you can ones implemented in code.

Comparing BRDFs

As you might have noticed, BRDF Explorer allows you to add many different BRDFs and toggle them. One of the most useful panels for this kind of comparison is the Polar Plot panel (see Figure 11-10). Different BRDFs are traced with different colors, so you can compare them easily. It's very similar to the 3D Plot panel, but easier to peruse when there is more than one BRDFs involved.

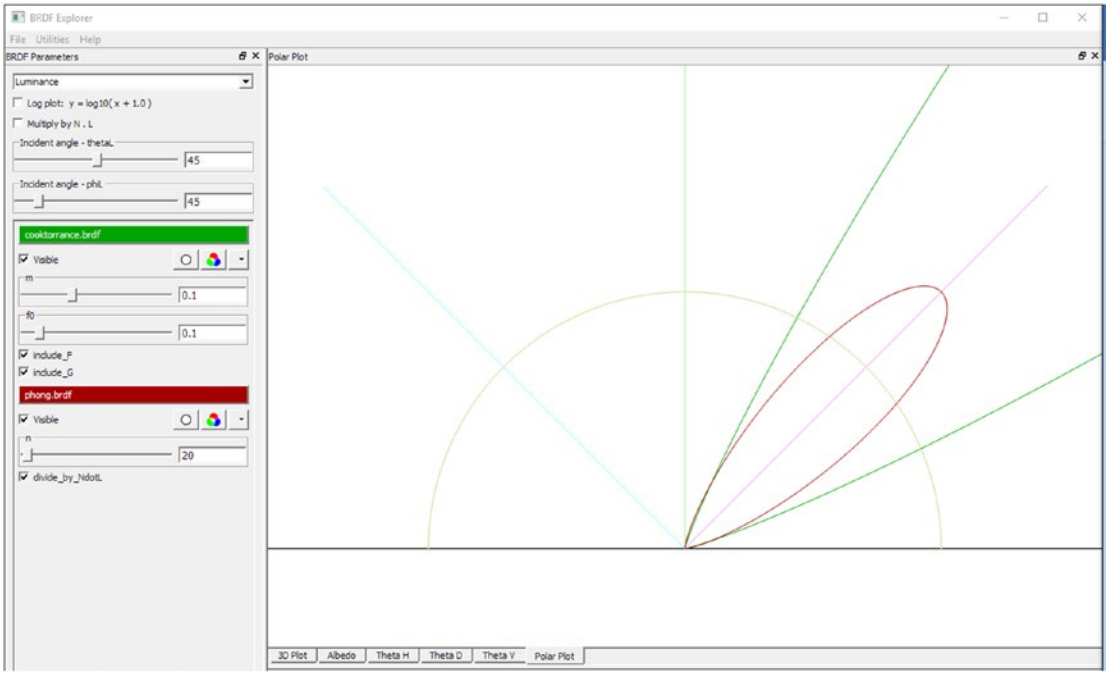

Figure 11-10. *Polar Plot panel, showing the specular lobe for Phong and Cook Torrance*

To compare BRDFs, you only need to load more than one `.brdf` or `.binary` file, and you'll be able to toggle them with the Visible checkbox and with the Solo This BRDF button. You can add as many BRDFs to the comparison as you'd like. Watch out in the 3D Plot panel, because the different BRDF lobes might overlap and obscure each other.

To try the multiple BRDF comparison, let's load the `cooktorrance.brdf` file. Cook Torrance is a very popular BRDF at the moment. It's used in Unity >= 5.x, in Unreal 4, and in other engines as well. It's very flexible and allows you to simulate many types of materials with only one BRDF. Since implementing indirect light for a BRDF within an engine is still cumbersome, due to the BRDF-related processing needed to create appropriate cubemaps and global illumination, engines tend to choose one, rather than having many available. You'll also rarely find them called by their actual names in the engine documentation.

For example, Unity 5.x uses calls the default shader "Standard" and doesn't specify which BRDFs lie underneath that name. Depending on platform, the actual BRDF used can be a variant of BlinnPhong, CookTorrance, or the Disney BRDF. The only way to find out is to look at the Standard shader code.

An Incomplete List of BRDFs Used in Real-Time Rendering

From a quick perusal of the files under the `brdf/` folder, you'll see many BRDF names, which you might not have heard of. Keep in mind that any file starting with `d_` is the distribution term of a microfacet BRDF, and any file starting with `g_` is the geometric term. This section describes the most relevant ones.

Ashikhmin Shirley

This is an anisotropic BRDF. It's a particularly good fit for metals, and oil and water layers (see Figure 11-11). You can see it applied to our usual duck in Figure 11-12.

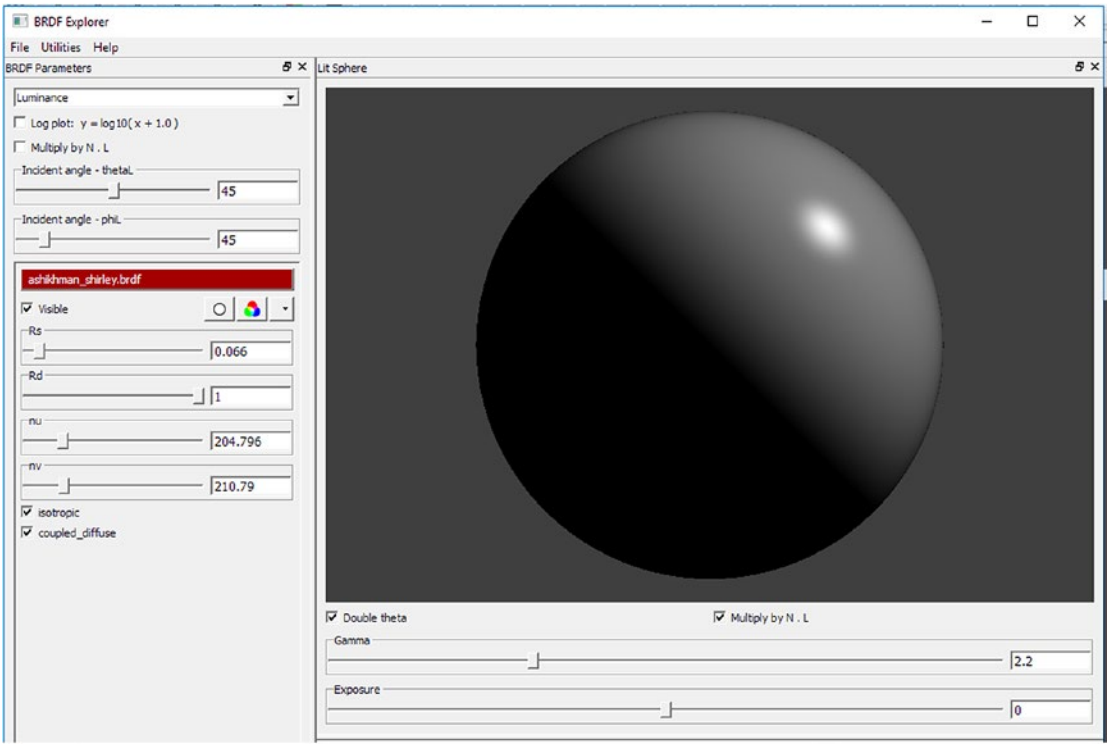

Figure 11-11. *Ashikhmin Shirley BRDF in BRDF Explorer*

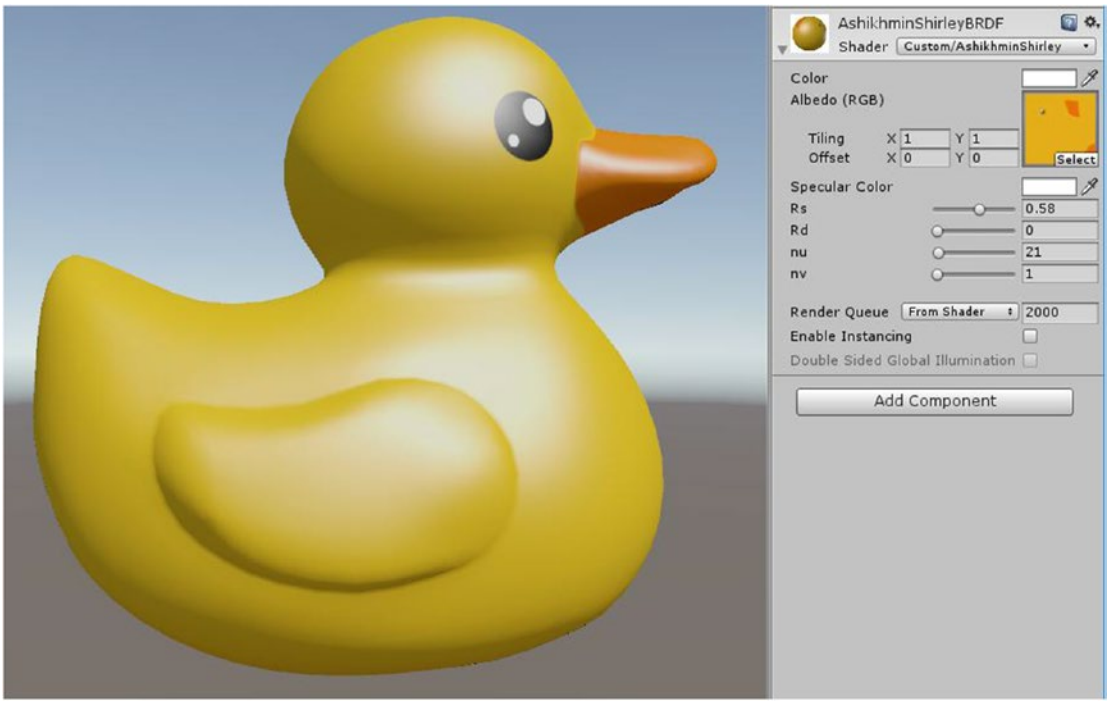

Figure 11-12. *Ashikhmin Shirley Specular BRDF in Unity*

This BRDF was introduced in the 2000 paper entitled *An Anisotropic Phong BRDF Model* by Michael Ashikhmin and Peter Shirley.

Cook Torrance

As mentioned, this is a very flexible BRDF. It can convincingly simulate metals, wood, plastic, and many other materials. It's based on microfacet theory. It can simulate materials with different roughness as well. See Figure 11-13.

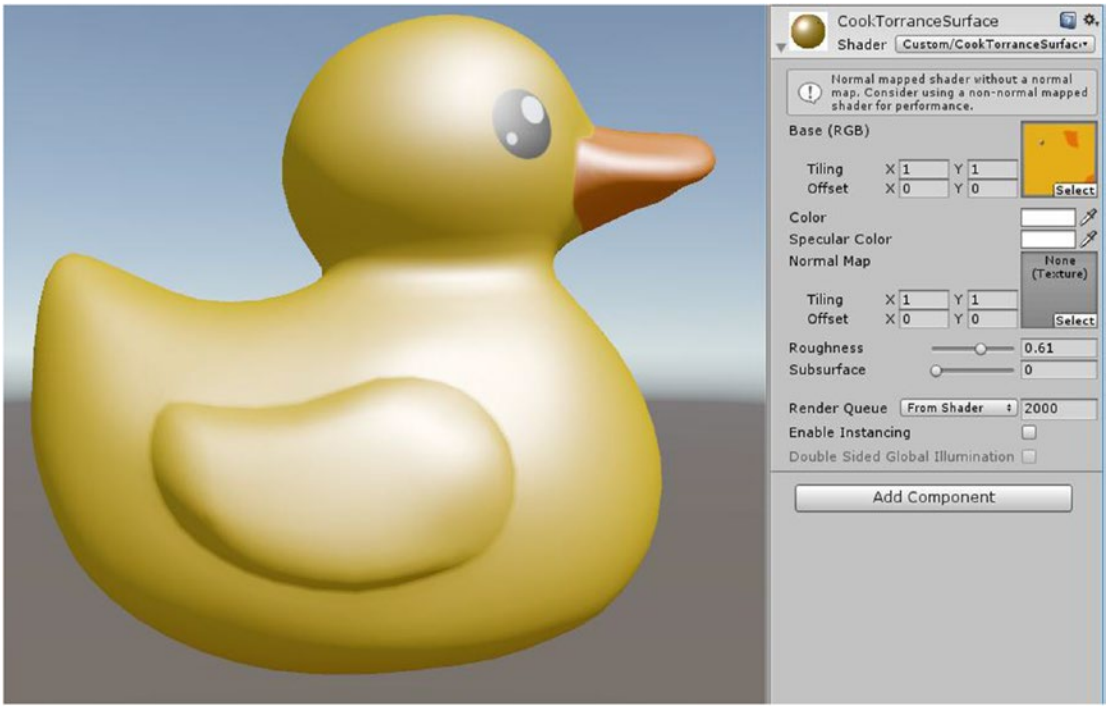

Figure 11-13. *CookTorrance Specular BRDF in Unity*

It was introduced in the 1982 paper entitled *A Reflectance Model for Computer Graphics* by Robert L. Cook and Kenneth E. Torrance.

Oren Nayar

This is a diffuse-only BRDF. It's more flexible than Lambert, and it can simulate roughness in diffuse materials. It was originally developed in order to model the non-Lambertian behavior of the moon. It doesn't include a specular component, so it needs to be paired with one to simulate materials that need a specular term. See Figure 11-14.

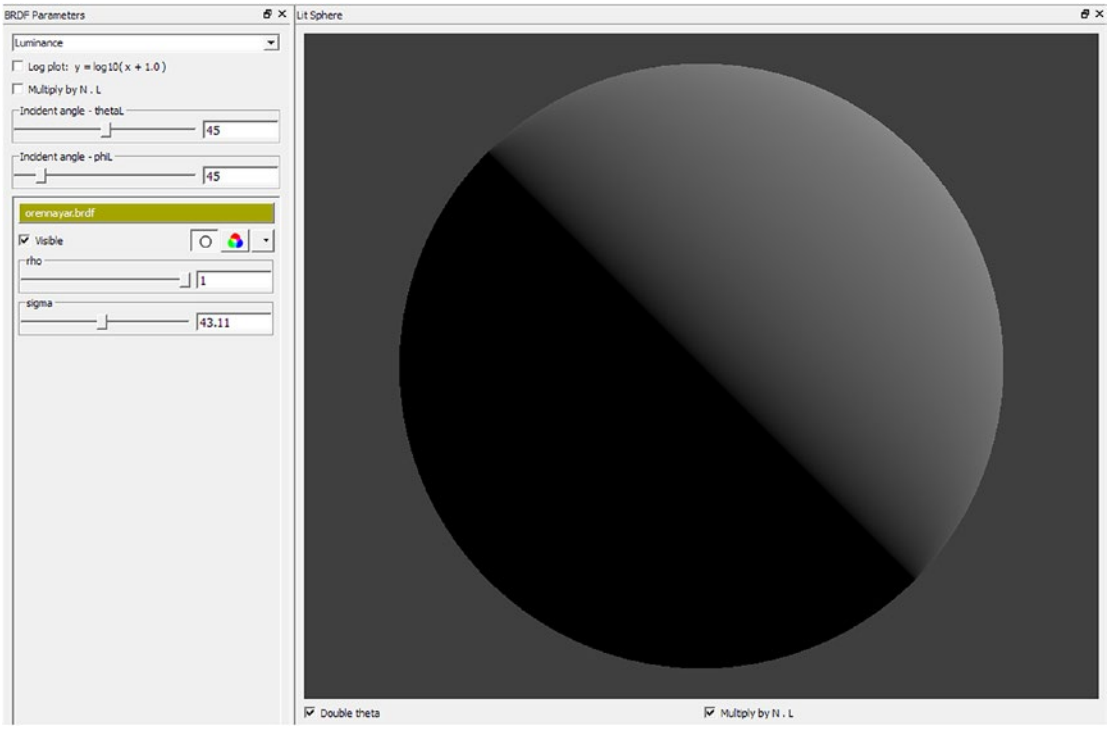

Figure 11-14. *Oren Nayar Diffuse BRDF in BRDF Explorer*

This BRDF was introduced in the 1994 paper entitled *Generalization of Lambert's Reflectance Model* by Michael Oren and Shree K. Nayar.

Ward

This is a rarely used BRDF. It's based on empirical data, and is made to fit the measured behavior of anisotropic reflection in materials. As a result, it can simulate metals well. See Figure 11-15.

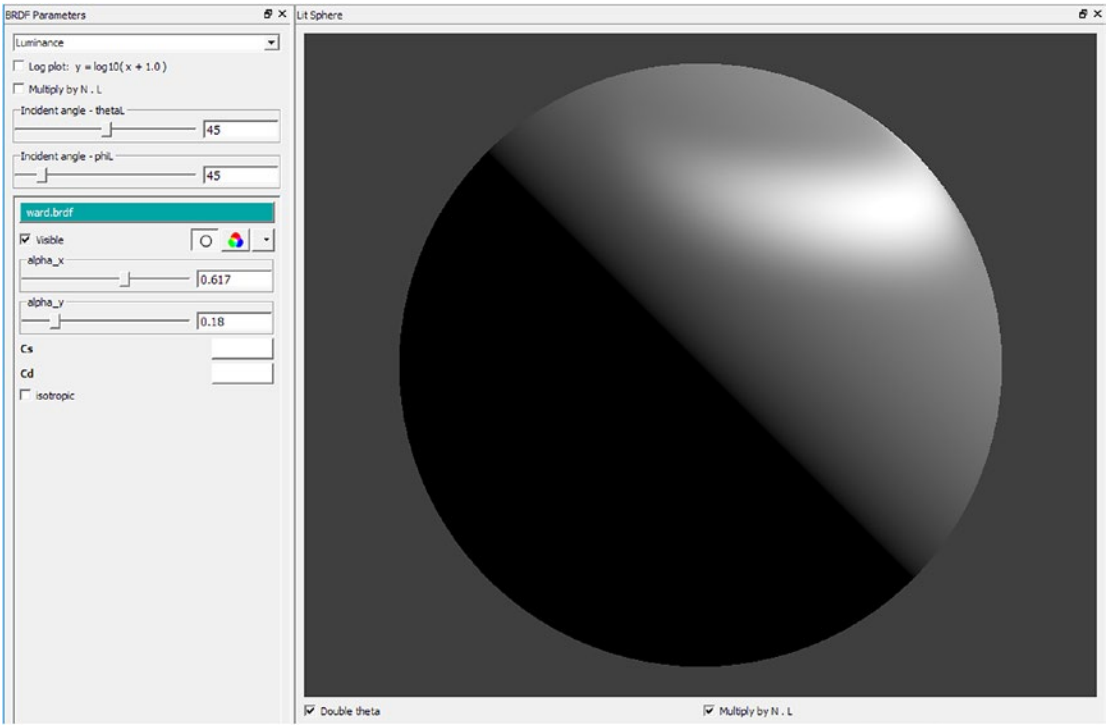

Figure 11-15. *Anisotropic Ward specular BRDF in BRDF Explorer*

This BRDF was introduced in the 1992 paper *Measuring and modeling anisotropic reflection* by Gregory J. Ward.

Disney

A BRDF invented by Brent Burley, for use in Disney's 3D rendered animated movies. It's an interesting BRDF to analyze, as it's built to be extremely flexible, yet very easy for artists to use. You can download an implementation for it from BRDF Explorer's GitHub, as it's not included in the binary release.

Looking at the parameters, you'll likely notice how much more intuitive the naming is. It uses actual descriptive names instead of opaque acronyms, or even single letters. You should try different settings while looking at the Polar or 3D Plot panels, to keep an eye on how dramatic changes in lobe shape can be with this BRDF (see Figure 11-16).

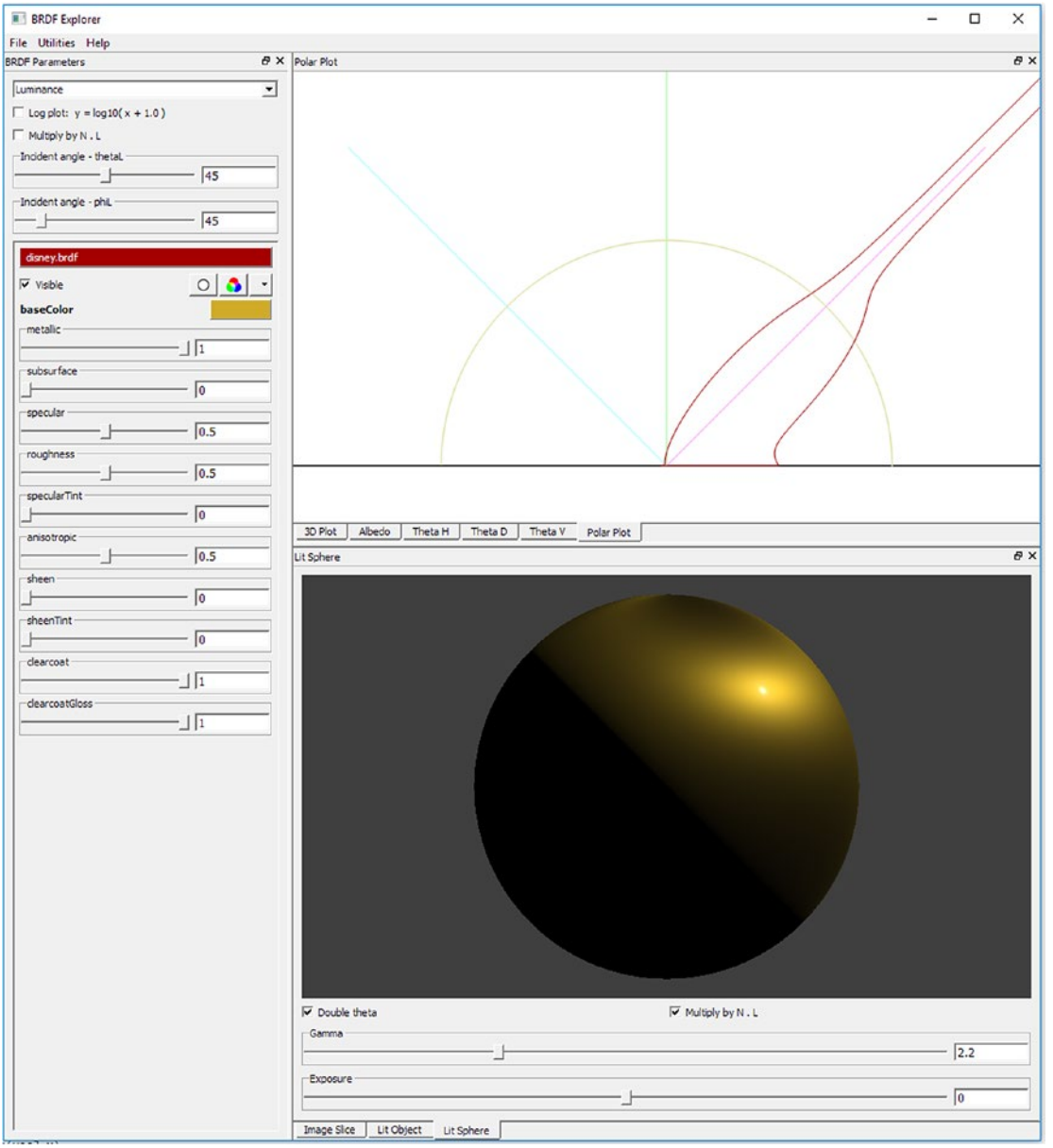

Figure 11-16. *Disney BRDF in BRDF Explorer*

One simple experiment is to switch from 0 metallic to 1 metallic, which will singlehandedly make the rendered material look like metal without changing any other parameters. Note how much lobe shape can change when you change the settings. That's because they are interpolating between very different lobes, instead of having just one type of lobe.

This BRDF was introduced in the talk *Physically Based Shading at Disney*, presented at the *Practical Physically Based Shading in Film and Game Production* course at SIGGRAPH 2012 by Brent Burley.

This is it for our presentation of BRDF Explorer and a few popular BRDFs. It's a very handy program and I encourage you to keep exploring what it can do.

Summary

This chapter introduced the BRDF Explorer program and showed how it's useful to check out and analyze BRDFs. It also presented many famous BRDFs and explained the materials they can simulate well.

Next

The next chapter shows you how to implement one of the BRDFs mentioned in this chapter in a Unity Surface shader.

CHAPTER 12

Implementing a BRDF

In Chapter 9, we modified Phong to make it energy conserving. In the last chapter, we introduced many different BRDFs that can be implemented.

In this chapter, we're going to implement one of those BRDFs as a surface shader in Unity.

Which BRDF to Implement?

The most popular BRDFs used in games at the moment are CookTorrance and the Disney BRDF. They're both physically based BRDFs to different degrees. The Disney BRDF is referred to as "principled" rather than entirely physically based.

Their main objective, when developing the Disney BRDF, was to make a BRDF capable of simulating most materials, but also one that's very intuitive for artists to deal with. When physics and intuitiveness contradicted each other, intuitiveness won. When we analyzed both of them in BRDF Explorer, you might have noticed how flexible Disney's lobes are. They can contort into some complex shapes, while the CookTorrance lobes can vary a lot in size and thickness, but they tend to keep their characteristic shape throughout.

CookTorrance is a good example of a microfacet-based BRDF, but Disney's is a more interesting BRDF overall, and it's a great example of designing a BRDF to be used by artists. In the interest of learning as much as is practical from both, we'll go through the first steps of implementation for both. Those could be called the "gathering references" stage of the process. The complete Disney BRDF is quite complex to implement, so it's not appropriate for your first BRDF implementation.

We are going to implement CookTorrance for the specular term and implement only the diffuse part of the Disney BRDF for the diffuse term. Microfacet BRDFs are specular-only, so they always need to be complemented with a diffuse BRDF. Normally, that would be Lambert. However, OrenNayar and the diffuse portion of the Disney BRDF are also good options.

Finding References

The first issue to address when implementing a BRDF is to find good sources. You should find the original BRDF formula, later variations on it, and a reference implementation (if you're lucky), or at least someone else's implementation attempt. For older BRDFs, there are plenty of implementations lying around, but they're liable to have been adapted to some specific platform or requirements. You should check how much they follow the original formula, rather than trusting them.

Most of the time, the code you find won't be targeted at the platform you want to implement the BRDF for anyway, so it pays to learn about the most common shading languages to be able to read those implementations. Shading languages are all fairly similar, so it shouldn't be a huge effort. HLSL and GLSL are among the most commonly used shading languages.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_12

In your search for reference materials, you're likely to find semi-abandoned repositories, implementers who ended up changing half the BRDF anyway, and other materials that can occasionally be useful, but can also confuse and mislead you. Thread with caution. Always check out the original paper, if it's available.

Warnings made—now let's gather our references.

CookTorrance

The CookTorrance BRDF has been around since 1981, when it was introduced in a research paper. It's very popular, so a lot of people have further researched it and refined it. It is derived from microfacet theory, which itself has been written about extensively.

Our main references:

- The slides and course notes for *Physics and Math of Shading* by Naty Hoffman, which was presented at SIGGRAPH every year from 2012 to 2015
- The slides and course notes for *Real Shading in Unreal Engine 4,* presented by Brian Karis at SIGGRAPH, 2013
- The implementation of the CookTorrance BRDF included in BRDF Explorer (GLSL)
- The article entitled *Specular BRDF Reference* on Brian Karis' blog [1]
- The paper entitled *An Introduction to BRDF Models* by Daniël Jimenez Kwast

Other, somewhat deeper, references include:

- The original paper entitled *A Reflectance Model for Computer Graphics* from 1981
- The slides, and course notes, from *Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs* by Eric Heitz at SIGGRAPH, 2014 (great for understanding microfacet theory)
- The paper entitled *Microfacet Models for Refraction through Rough Surfaces* from EGSR, 2017

As you can see, we're drawing from more recent reference materials, which include the latest research for AAA game rendering techniques. But if you have time, it's good to go through the history of a BRDF and track which changes were made by whom, and where. For this, a SIGGRAPH membership can be invaluable, but you should be able to just Google these references and find free versions of them, should you want to.

Many graphics programmers and researchers also keep a blog, where they often publish useful information and explain papers and techniques; see Chapter 18 for an incomplete list of them.

Disney

The Disney BRDF has been published by Brent Burley in 2012. It's a very recent BRDF, and that means there isn't a lot of material about it yet, apart from the original sources.

Our references for the Disney BRDF:

- The original slides, and course notes, from the talk *Physically Based Shading at Disney,* presented at the Practical Physically Based Shading in Film and Game Production course at SIGGRAPH, 2012, by Brent Burley
- The implementation of the Disney BRDF included in BRDF Explorer (GLSL)

[1] http://graphicrants.blogspot.co.uk/2013/08/specular-brdf-reference.html

Other, somewhat useful, references include:

- The original slides, and course notes, from the talk *Extending the Disney BRDF to a BSDF with Integrated Subsurface Scattering* presented at SIGGRAPH, 2015, by Brent Burley
- The original slides, and course notes, from the talk *Moving Frostbite to Physically Based Rendering* presented at SIGGRAPH, 2015, by Sébastien Lagarde and Charles de Rousiers (only for the Diffuse)
- Implementation of Disney BSDF for Blender (C++)

Starting from the Paper

This section goes through the main characteristics of the BRDFs and their formulas, where available. We show quite a few of the formulas included in the papers, because after a certain point, implementing a BRDF boils down to translating the math into code.

So beware, math is coming, although there's nothing particularly fancy about these formulas mathematically. They just use addition, multiplication, subtraction, division, exponentiation, fractions, and cosines, which in the code implementation will become dot products. Possibly the worst part is that they are sprinkled with one-letter variable names that need some getting used to. For some added ease of mental parsing, I've included code examples for most formulas.

CookTorrance (or Microfacet) BRDF

Everything you learned about microfacet theory in Chapter 8 applies to CookTorrance. CookTorrance is almost a synonym for microfacet theory at this point, because the skeleton of it ends up being reused while being refitted with newer "parts". You might remember that BRDFs derived from microfacet theory have different terms: **D** (Distribution), **F** (Fresnel), and **G** (Geometry, which mainly deals with masking and shadowing).

You can plug in different approximations in place of those terms. For example, there are various approximations for Fresnel, with Schlick being the most popular. Likewise, for the Geometry term, a popular option is Schlick's approximation to the Smith shadowing function. There is also the Walter approximation of the same function, and other functions altogether.

And last, but actually most important, there are many distribution functions (NDF, Normal Distribution Functions) you can choose from. The distribution function makes a big difference on the BRDF result. Some suitable NDFs are Beckmann, Phong, and GGX.

In order to build our CookTorrance implementation, we have to choose for each term, which of the options to go for. Normally it would be a matter of keeping the shader fast, while making it look as good as possible. The choices would vary according to the power of the target platform. In our case we'll go with the most common options, meaning GGX, Schlick for the Fresnel term, and Schlick again for the Geometry term.

This is the formulation of CookTorrance you saw in Chapter 8, which you will find in papers from recent years:

$$f(l,v)=\frac{D(h)F(l,h)G(l,v,h)}{4(n\cdot l)(n\cdot v)}$$

D is the Distribution function, **G** is the Geometry term, and **F** is the Fresnel term. From this formula, we can see that the NDF depends on the half-vector, the Fresnel depends on light direction and the half-vector, and the Geometry term depends on light direction, view direction, and the half-vector.

We already gave an overview of the terms in this formula, now let's provide formulas for them too. First, the Schlick approximation for the Fresnel:

$$F_{schlick}(F_0, l, h) = F_0 + (1 - F_0)(1 - (l \cdot h))^5$$

As you may have noticed, it depends on the half-vector, rather than on v or ω_0 or any other variable that stands in for the outgoing light direction. This is because this is the version to be used in BRDFs. F_0 stands for the specular color.

In a naïve implementation, this Fresnel term could look like this:

```
float SchlickFresnel(float4 SpecColor, float lightDir, float3 halfVector)
{
    return SpecColor + ( 1 - SpecColor ) * pow( 1 - ( dot( lightDir, halfVector )), 5);
}
```

Next is the Schlick approximation for Smith's shadowing function. We're going to use the modified version from the Unreal paper, because it's harmonized to work with the NDF of our choice, GGX.

First we define K:

$$k = \frac{(Roughness + 1)^2}{8}$$

In code:

```
float modifiedRoughness = _Roughness + 1;
float k = sqr(modifiedRoughness) / 8;
```

Which is then used by the G_1 function:

$$G_1(v) = \frac{n \cdot v}{(n \cdot v)(1 - k) + k}$$

In code:

```
float G1 (float k, float NdotV)
{
    return NdotV / (NdotV * (1 - k) + k);
}
```

This needs to be applied once for the light direction and once for the view direction, and then multiplied:

$$G(l, v, h) = G_1(l) G_1(v)$$

In code:

```
float g1L = G1(k, NdotL);
float g1V = G1(k, NdotV);
G = g1L * g1V;
```

Lastly, we need a formulation for the NDF, the distribution function. We're going to use GGX, and again we're going with a slightly modified version from the Unreal paper:

$$D(h)=\frac{\alpha^2}{\pi\left((n\cdot h)^2\left(\alpha^2-1\right)+1\right)^2}$$

In code:

```
float alphaSqr = sqr(alpha);
float denominator = sqr(NdotH) * (alphaSqr - 1.0) + 1.0f;
D = alphaSqr / (PI * sqr( denominator));
```

You might have noticed that we did not mention a diffuse. This is because CookTorrance doesn't include one. You'll have to plug one in from a different source. Commonly Lambert is used, but there are other options.

With this, we should have all the pieces to implement this BRDF. The next step is to start the implementation.

Disney BRDF

Similarly to CookTorrance, this section overviews the Disney BRDF.

An important point from the paper, which can be applied to any BRDF implementation, is the fact that all parameters are supposed to share one range, from 0 to 1. This is a very nice touch, as many BRDF have counterintuitive, wildly varying ranges.

In this BRDF, many parameters are basically just lerping between different lobe shapes, which is an interesting approach. It allows the BRDF to match the variability of real-life materials better.

Looking at the formulas, this BRDF is inspired from microfacet theory, even if not outright derived from it, so the different terms that compose the formula will be familiar, as we just looked at CookTorrance.

The diffuse part of the model is meant to improve on Lambert. You might remember the Fresnel Schlick approximation; this is a formulation to use for diffuse:

$$\left(1-F(\theta_l)\right)\left(1-F(\theta_d)\right)$$

Differently from Lambert, the Disney diffuse formula uses Fresnel:

$$f_d=\frac{baseColor}{\pi}\left(1+\left(F_{D90}-1\right)\left(1-\cos\theta_l\right)^5\right)\left(1+\left(F_{D90}-1\right)\left(1-\cos\theta_v\right)^5\right)$$

where

$$F_{D90}=0.5+2\times roughness\times\cos^2\theta_d$$

These formulas are defined in the original paper, which differs from the actual implementation code more than anything we've seen with CookTorrance, because they don't include the lerping between different lobes.

In a naïve implementation, they would look like this:

```
float fresnelDiffuse = 0.5 + 2 * sqr(LdotH) * roughness;
float fresnelL = 1 + (fresnelDiffuse - 1) * pow(1 - NdotL, 5);
float fresnelV = 1 + (fresnelDiffuse - 1) * pow(1 - NdotV, 5);
float3 Fd = (BaseColor / PI) * fresnelL * fresnelV
```

This is an empirical model; the aim is to behave differently according to the roughness of the material. Smooth materials need to be slightly darker, which is achieved by a Fresnel shadow, and rough materials need to be made slightly lighter. The Subsurface parameter blends between this diffuse shape and a different BRDF, inspired by the Hanrahan-Krueger subsurface BRDF [2].

Next, we're going to summarize the Specular part of the BRDF for completeness, but we're not going to implement it.

We start from **D**, the distribution:

$$D_{GTR} = c / \left(\alpha^2 \cos^2 \theta_h + sin^2 \theta_h \right)^{\gamma}$$

Called *Generalized-Trowbridge-Reitz,* as you may surmise from the name, it is inspired by the Trowbridge-Reitz distribution, with the objective of obtaining a longer tail, which is the softer falloff of a specular outside of the brightest highlight. α is a roughness parameter, just as with the modified CookTorrance from the previous section, and c is a scaling constant. They prefer remapping α as $\alpha = roughness^2$ because it produces a perceptual result that feels more linear. Achieving the illusion of linear increases in some visual quality is not necessarily easy or intuitive. For example, our eyes perceive colors in a non-linear fashion. We see more shades of dark colors than of light ones.

They chose to have three specular lobes, one for the main specular, one for the clear coat (both using this GTR model), and one for the sheen. The sheen specular lobe uses the Fresnel Schlick shape. The Specular parameter determines the incident specular amount. It remaps to a range that covers most common materials. The clear coat layer has a fixed value for the specular that corresponds to the index of refraction of polyurethane.

Next, the Fresnel term. Again, it's the Schick approximation; nothing new to see here. Next, the G term uses a modified version of one derived for GGX. The G for the clear coat layer is fixed.

Implementation

Let's start implementing our CookTorrance specular, and later we'll add our Disney diffuse to it.

Properties

Let's get started with the `Properties` block and the beginning of the `SubShader` block (see Listing 12-1). The only extra property we need is `_Roughness`. Note that if we were implementing the entire Disney BRDF, there would be many more properties declared here. We're also getting the `surf` function out of the way, since there is nothing fancy in there either.

[2]Hanrahan, Pat and Wolfgang Krueger. *Reflection from layered surfaces due to subsurface scattering*. In proceedings of the 20th annual conference on computer graphics and interactive techniques, SIGGRAPH, 1993, pages 165–174, New York, NY, USA, 1993. ACM.

Listing 12-1. Properties, Variables, Structs, and Surf Setup

```
Shader "Custom/CookTorranceSurface" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _ColorTint ("Color", Color) = (1,1,1,1)
        _SpecColor ("Specular Color", Color) = (1,1,1,1)
        _BumpMap ("Normal Map", 2D) = "bump" {}
        _Roughness ("Roughness", Range(0,1)) = 0.5
        _Subsurface ("Subsurface", Range(0,1)) = 0.5

    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf CookTorrance fullforwardshadows
        #pragma target 3.0

        struct Input {
            float2 uv_MainTex;
        };

        #define PI 3.14159265358979323846f

        sampler2D _MainTex;
        sampler2D _BumpMap;
        half _Roughness;
        float _Subsurface;
        fixed4 _ColorTint;

        UNITY_INSTANCING_CBUFFER_START(Props)
        UNITY_INSTANCING_CBUFFER_END

        struct SurfaceOutputCustom {
            fixed3 Albedo;
            fixed3 Normal;
            fixed3 Emission;
            fixed Alpha;
        };

        void surf (Input IN, inout SurfaceOutputCustom o) {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _ColorTint;
            o.Albedo = c.rgb;
            o.Normal = UnpackNormal( tex2D ( _BumpMap, IN.uv_MainTex ) );
            o.Alpha = c.a;
        }
```

Note the `SurfaceOutputCustom` struct, which has fewer members than the `Standard` one. We are already passing our custom lighting function to `surf`, but we haven't started writing it yet, so the code won't compile yet. Among the variables, `_Roughness` is used by the CookTorrance specular and the Disney diffuse. `_Subsurface` is only used by the Disney diffuse. We could optionally skip the subsurface approximation within the Disney diffuse, but it looks nice, so we're keeping it.

Make sure you declare the variable `PI` with a `#define` exactly how it looks in Listing 12-1; otherwise, you might incur horrible shading errors, depending on the platform you're using.

Custom Light Function Implementation

Now we'll start the process of condensing this information in implementations of CookTorrance and the Disney diffuse. First, we need to create the two functions necessary to implement a custom light in a surface shader. In this case they'll be called:

- `inline void LightingCookTorrance_GI`
- `inline fixed4 LightingCookTorrance`

For now, we'll just copy one of custom GI functions we already wrote in the past chapters. We'll dig some more into how they work in the next chapter. We're also going to need some extra utility functions later, to abstract away some repeated operations.

I prefer to keep the custom lighting function as the place we calculate most necessary values (`NdotL`, etc.), and where we stitch together diffuse and specular (see Listing 12-2). The actual specular and diffuse calculations are kept in their own functions, so it's all nice and modular.

This way you can easily switch a diffuse or a specular for a different one, while only changing the bare minimum lines of code in the custom lighting function.

Listing 12-2. The Custom Lighting Function Setup

```
inline float4 LightingCookTorrance (SurfaceOutputCustom s, float3 viewDir, UnityGI gi){
    //Needed values

    UnityLight light = gi.light;

    viewDir = normalize ( viewDir );
    float3 lightDir = normalize ( light.dir );
    s.Normal = normalize( s.Normal );

    float3 halfV = normalize(lightDir+viewDir);
    float NdotL = saturate( dot( s.Normal, lightDir ));
    float NdotH = saturate( dot( s.Normal, halfV ));
    float NdotV = saturate( dot( s.Normal, viewDir ));
    float VdotH = saturate( dot( viewDir, halfV ));
    float LdotH = saturate( dot( lightDir, halfV ));

    // BRDFs
    float3 diff = DisneyDiff(s.Albedo, NdotL,  NdotV, LdotH, _Roughness);
    float3 spec = CookTorranceSpec(NdotL, LdotH, NdotH, NdotV, _Roughness, _SpecColor);

    //Adding diffuse, specular and tints (light, specular)
    float3 firstLayer = ( diff + spec * _SpecColor) * _LightColor0.rgb;
    float4 c = float4(firstLayer, s.Alpha);
```

```
    #ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
        c.rgb += s.Albedo * gi.indirect.diffuse;
    #endif

    return c;
}
```

Straight away, we're calculating most variables we need and then passing them to the diffuse and specular functions. This has the advantage of calculating the values only once, and also reminding us where they are actually used. It's pretty easy to forget unused variables, which can make the code plenty more confusing.

Utility Functions

The next priority is to get the utility functions out of the way (see Listing 12-3).

Listing 12-3. Various Utility Functions

```
float sqr(float value)
{
    return value * value;
}

float SchlickFresnel(float value)
{
    float m = clamp(1 - value, 0, 1);
    return pow(m, 5);
}

float G1 (float k, float x)
{
    return x / (x * (1 - k) + k);
}
```

`sqr` is a simple power of two function; we're using it to make the code more readable. Then we have the `SchlickFresnel` approximation, used both by the diffuse and by the specular. Then we have the `G1` function, used by the CookTorrance Geometry function.

CookTorrance Implementation

Now that the infrastructure of the shader is set up, let's get started with the CookTorrance implementation. We're going to create a separate function, which takes $n \cdot l$, $l \cdot h$, $n \cdot h$, $n \cdot v$, *roughness,* and F_0 he specular color, which stands in for the index of refraction). Then within it, we're going to calculate the **F**, **D**, and **G** terms, according to the formulas you saw earlier in the chapter.

Modified GGX Distribution Term

We call it modified, because the GGX is using the $\alpha = roughness^2$ reparameterization. This formulation of GGX is relatively simple to implement (see Listing 12-4).

$$D(h) = \frac{\alpha^2}{\pi\left((n \cdot h)^2\left(\alpha^2 - 1\right) + 1\right)^2}$$

Listing 12-4. The Code Implementation of the Modified GGX NDF

```
// D
float alphaSqr = sqr(alpha);
float denominator = sqr(NdotH) * (alphaSqr - 1.0) + 1.0f;
D = alphaSqr / (PI * sqr(denominator));
```

If you look at the formula and the code closely, you can see how they match each other quite precisely.

Schlick Fresnel Term

This is the Schlick Fresnel as used in CookTorrance. As you can see, the part of it that is in common with the Disney diffuse is implemented in a utility function to avoid repeated code. By using `F0` as a variable, instead of the more intuitive `_SpecColor`, you can really see the parallels between the function and code (see Listing 12-5).

$$F_{schlick}(F_0, l, h) = F_0 + (1 - F_0)\left(1 - (l \cdot h)\right)^5$$

Listing 12-5. Implementation of the Fresnel Term

```
// F
float LdotH5 = SchlickFresnel(LdotH);
F = F0 + (1.0 - F0) * LdotH5;
```

Modified Schlick Geometry Term

This is a slightly more complex term. To better follow the Smith Geometry term, the variable k has been changed to $k = \frac{\alpha}{2}$. Further changes coming from the Disney BRDF have been applied to *Roughness*, which has been changed to $\frac{Roughness + 1}{2}$ before squaring.

The result of these changes can be seen here:

$$k = \frac{(Roughness + 1)^2}{8}$$

This last change comes from the Disney BRDF, and it should help harmonize our Disney diffuse with this CookTorrance implementation.

$$G_1(v) = \frac{n \cdot v}{(n \cdot v)(1-k)+k}$$

$$G(l,,v,,h) = G_1(l)G_1(v)$$

Again, the implementation code (see Listing 12-6) is relatively straightforward and pretty much the same as our example implementation a few sections ago. Keep in mind that the function G_1 is in the utility functions.

Listing 12-6. Implementation of the Geometry term, Schlick's Approximation of Smith

```
// G
float r = _Roughness + 1;
float k = sqr(r) / 8;
float g1L = G1(k, NdotL);
float g1V = G1(k, NdotV);
G = g1L * g1V;
```

Putting the CookTorrance Together

Now that we have implemented all the terms, let's put them together into one function (see Listing 12-7).

Listing 12-7. The Complete CookTorrance Function

```
inline float3 CookTorranceSpec( float NdotL, float LdotH, float NdotH,
                                float NdotV, float roughness, float F0)
{
    float alpha = sqr(roughness);
    float F, D, G;

    // D
    float alphaSqr = sqr(alpha);
    float denom = sqr(NdotH) * (alphaSqr - 1.0) + 1.0f;
    D = alphaSqr / (PI * sqr(denom));

    // F
    float LdotH5 = SchlickFresnel(LdotH);
    F = F0 + (1.0 - F0) * LdotH5;

    // G
    float r = _Roughness + 1;
    float k = sqr(r) / 8;
    float g1L = G1(k, NdotL);
    float g1V = G1(k, NdotV);
    G = g1L * g1V;

    float specular = NdotL * D * F * G;
    return specular;
}
```

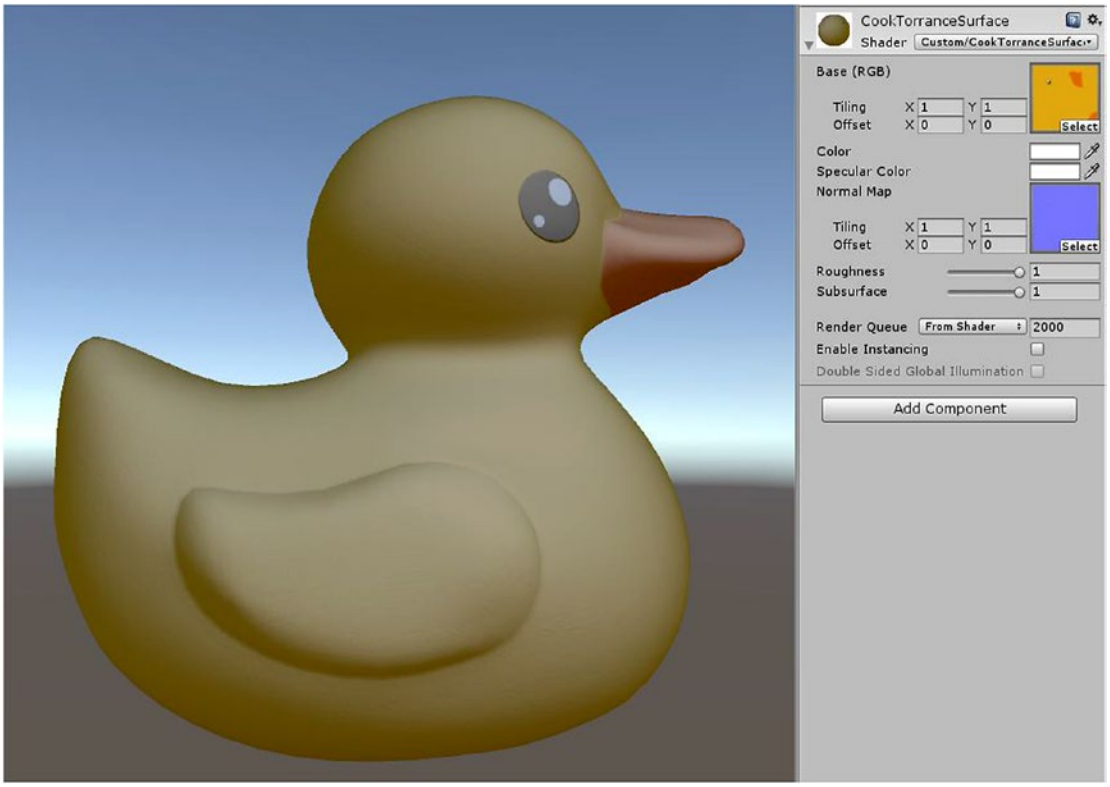

Figure 12-1. *CookTorrance at maximum roughness*

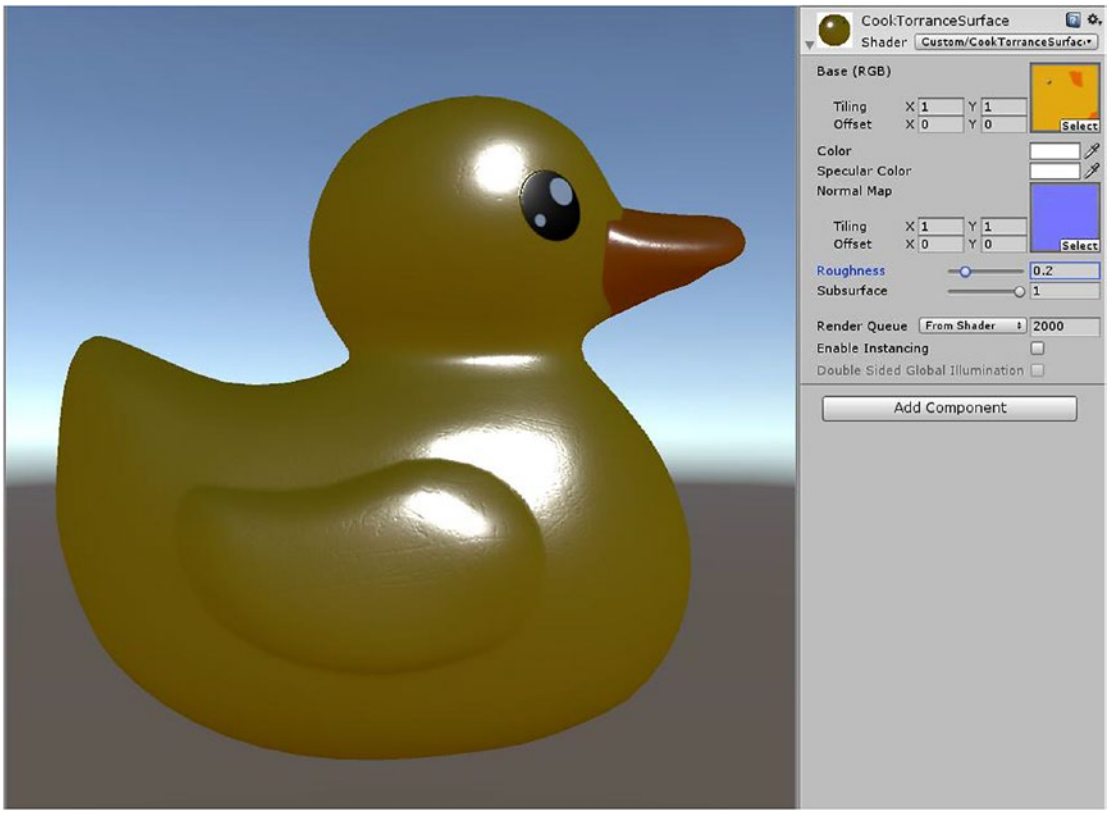

Figure 12-2. *CookTorrance at 0.2f roughness*

Let's take a look at our temporary result. In Figure 12-1, you can see how the specular looks at maximum roughness. Notice that it looks a bit dark, which is normal since we don't have a diffuse.
Figure 12-2 shows how the specular looks with 0.2f roughness. You can see it does look physically based, as the small highlight is much brighter.

With the CookTorrance specular completely implemented, let's focus on the next step: the Disney diffuse.

Disney Diffuse

We've chosen to implement the Disney diffuse. We could have gone with a simpler Lambert, but you've done Lambert already, and this is more interesting.

The Disney BRDF is quite extensive, but its diffuse is small enough for our purposes. From our description of the BRDF, here are the formulas that concern the diffuse:

$$F_{D90} = 0.5 + 2 \times roughness \times \cos^2 \theta_d$$

$$f_d = \frac{baseColor}{\pi}\left(1+\left(F_{D90}-1\right)\left(1-\cos\theta_l\right)^5\right)\left(1+\left(F_{D90}-1\right)\left(1-\cos\theta_v\right)^5\right)$$

We'll use the Schlick Fresnel function from the utility functions for this. Also, we should consider code that's not included in the paper, but is in the reference implementation in the BRDF Explorer program. Looking at the implementation code in Listing 12-8, you may notice that there are quite a few lerps used. A lot of this could be simplified away, but I elected to keep it, because we have no performance worries and the result looks nice. In a real-life commercial game, you'd have to be much more ruthless about what makes enough of a visual difference, and approximate or remove the rest.

Listing 12-8. The Disney Diffuse Implementation

```
//Disney Diffuse
inline float3 DisneyDiff(float3 albedo, float NdotL,
                         float NdotV,  float LdotH, float roughness){

    // luminance approximation
    float albedoLuminosity = 0.3 * albedo.r
                           + 0.6 * albedo.g
                           + 0.1 * albedo.b;

    // normalize luminosity to isolate hue and saturation
    float3 albedoTint = albedoLuminosity > 0 ?
                        albedo/albedoLuminosity:
                        float3(1,1,1);

    float fresnelL = SchlickFresnel(NdotL);
    float fresnelV = SchlickFresnel(NdotV);

    float fresnelDiffuse = 0.5 + 2 * sqr(LdotH) * roughness;

    float diffuse = albedoTint
                  * lerp(1.0, fresnelDiffuse, fresnelL)
                  * lerp(1.0, fresnelDiffuse, fresnelV);

    float fresnelSubsurface90 = sqr(LdotH) * roughness;

    float fresnelSubsurface = lerp(1.0, fresnelSubsurface90, fresnelL)
                            * lerp(1.0, fresnelSubsurface90, fresnelV);

    float ss = 1.25 * (fresnelSubsurface * (1 / (NdotL + NdotV) - 0.5) + 0.5);

    return saturate(lerp(diffuse, ss, _Subsurface) * (1/PI) * albedo);
}
```

This is the complete Disney diffuse function. Matching it with the formula is somewhat harder, compared to the CookTorrance we just implemented, which was almost a perfect match. Although the key bits of functionality can be traced back to the formulas, the rest is adapted from the reference implementation.

Now the shader is complete. Figure 12-3 shows the result at maximum roughness and maximum subsurface. Figure 12-4 shows maximum roughness and minimum subsurface. As you can see, this diffuse is much more varied and more interesting than Lambert. The subsurface approximation is also quite good.

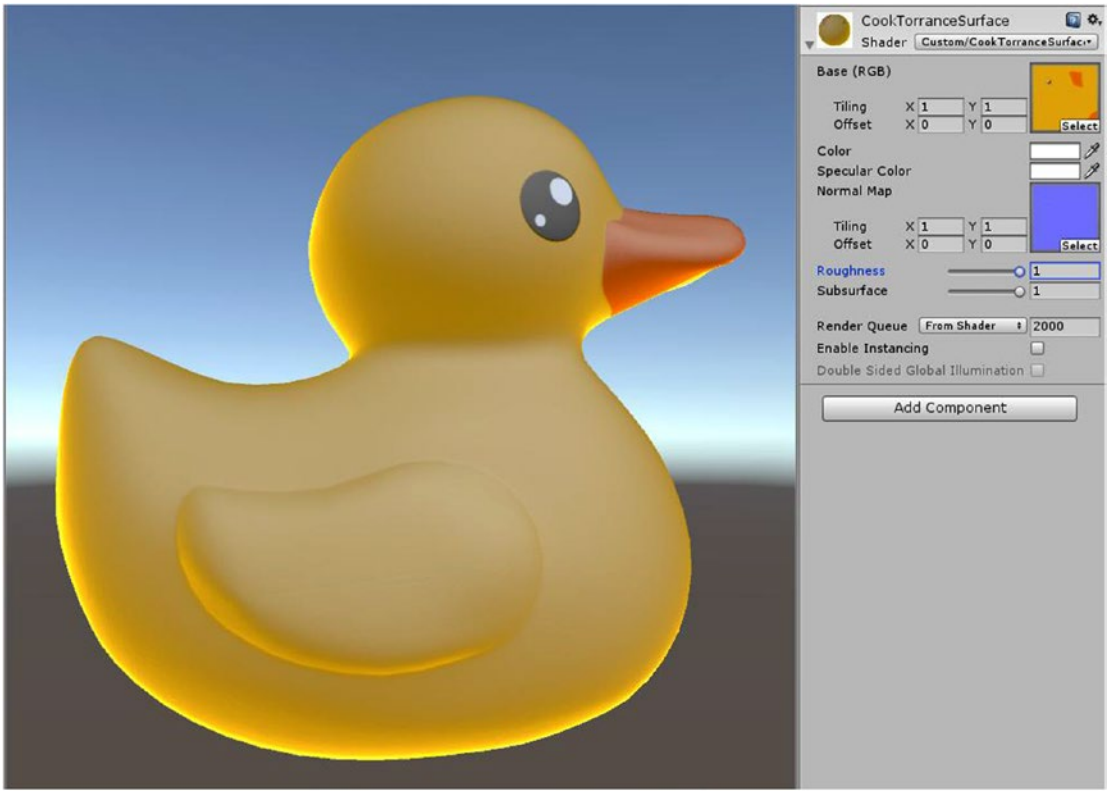

Figure 12-3. *CookTorrance and Disney diffuse at maximum roughness and maximum subsurface*

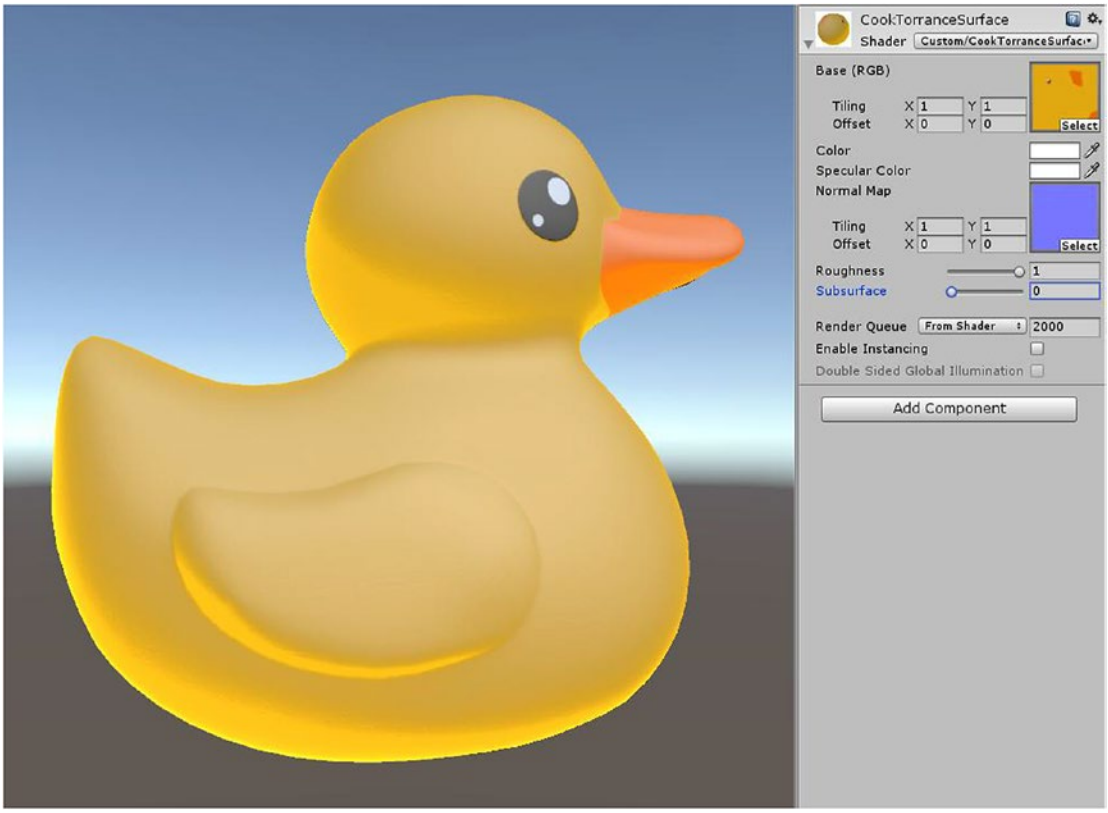

Figure 12-4. *CookTorrance and Disney diffuse at maximum roughness and minimum subsurface*

For completeness, let's also look at the shader at lower roughness, as shown in Figure 12-5. It does behave well, but at very low roughness, the highlight all but disappears. You'll probably want to reduce the range of the `_Roughness` to start at `0.1f` or thereabouts.

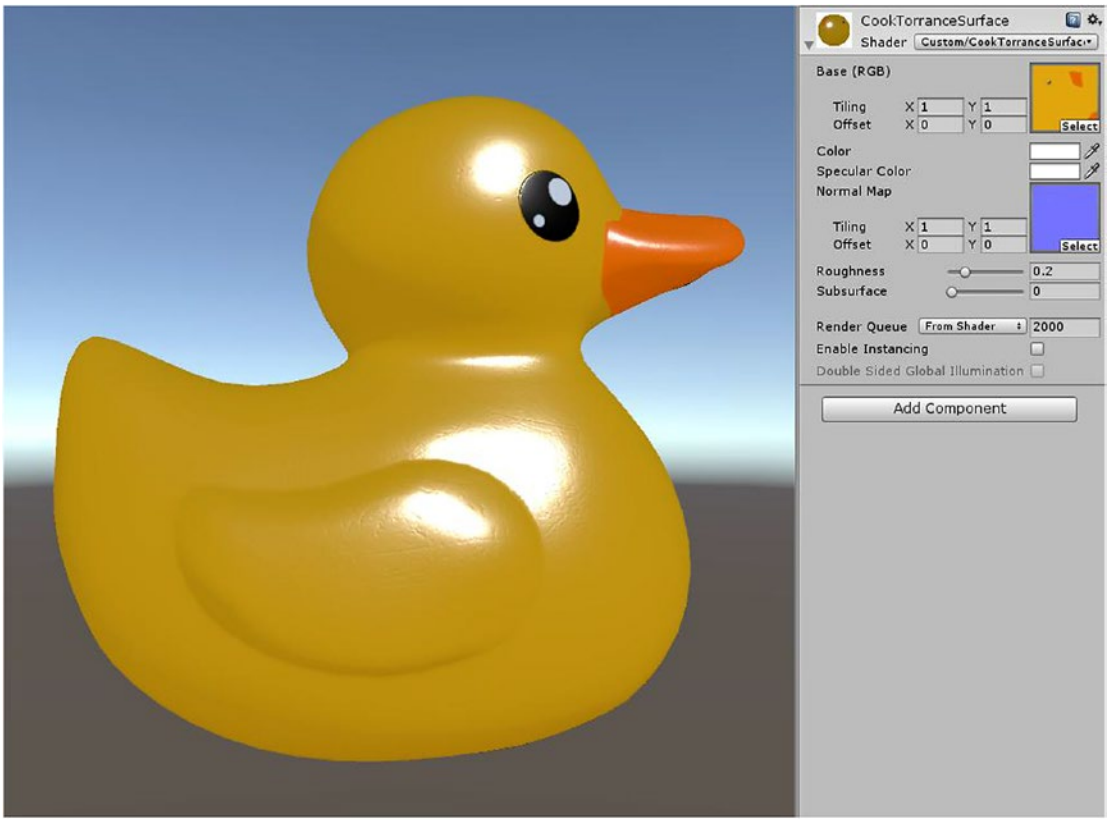

Figure 12-5. *The CookTorrance specular and Disney diffuse at low roughness*

The Disney BRDF diffuse has been developed to work with the Disney BRDF, which can have three lobes—a clear coat, a sheen, and a specular. As a result, it tends to get too dark with lower roughness values. Some tweaking might help it work better with just a CookTorrance specular lobe, but that's outside the scope of this book, as we'd need to speak a lot more about math and how to derive BRDFs.

But we can do something about it, even without delving into more complex math. We can try to implement the modified Disney diffuse from the Frostbite paper, which was one of the references. They are also using it with GGX, and without many the extra lobes from the Disney BRDF, so they didn't want it to get too dark at lower roughness either. Avoiding that problem basically boils down to renormalizing it and therefore getting energy conservation back. You should take a look at the Frostbite paper for more information.

Another Implementation of the Disney Diffuse

Without spending a lot of time on it, we can adapt the example code from the Frostbite paper and make it work with our infrastructure (see Listing 12-9).

The Frostbite infrastructure is very different from Unity's. Frostbite is focused on area lights; Unity does not include these for real-time as of 2017.1. This difference makes their code not directly portable to Unity without great effort. So tweaking it to work exactly as it was intended, but in Unity, would require some serious perusal of the Frostbite paper.

The objective here is to try another implementation of the Disney diffuse and see whether it works better for our use case.

Listing 12-9. Disney Diffuse Implementation Adapted from Frostbite

```
    float3 FresnelSchlickFrostbite (float3 F0, float F90, float u)
    {
        return F0 + (F90 - F0) * pow (1 - u, 5);
    }

    inline float DisneyFrostbiteDiff( float NdotL, float NdotV,
                                      float LdotH, float roughness)
    {
        float energyBias = lerp (0, 0.5, roughness) ;
        float energyFactor = lerp (1.0, 1.0/1.51, roughness );
        float Fd90 = energyBias + 2.0 * sqr(LdotH) * roughness;
        float3 F0 = float3 (1, 1, 1);
        float lightScatter = FresnelSchlickFrostbite (F0, Fd90, NdotL).r;
        float viewScatter  = FresnelSchlickFrostbite (F0, Fd90, NdotV).r;
        return lightScatter * viewScatter * energyFactor;
    }
[...]
    //within the custom lighting function
    float3 diff2 = (DisneyFrostbiteDiff(NdotL, NdotV, LdotH, _Roughness) * s.Albedo)/PI;
```

Using this diffuse implementation, we've lost the subsurface, but we get more even lighting. It still ends up being darker overall compared to the previous implementation (see Figures 12-6 and 12-7).

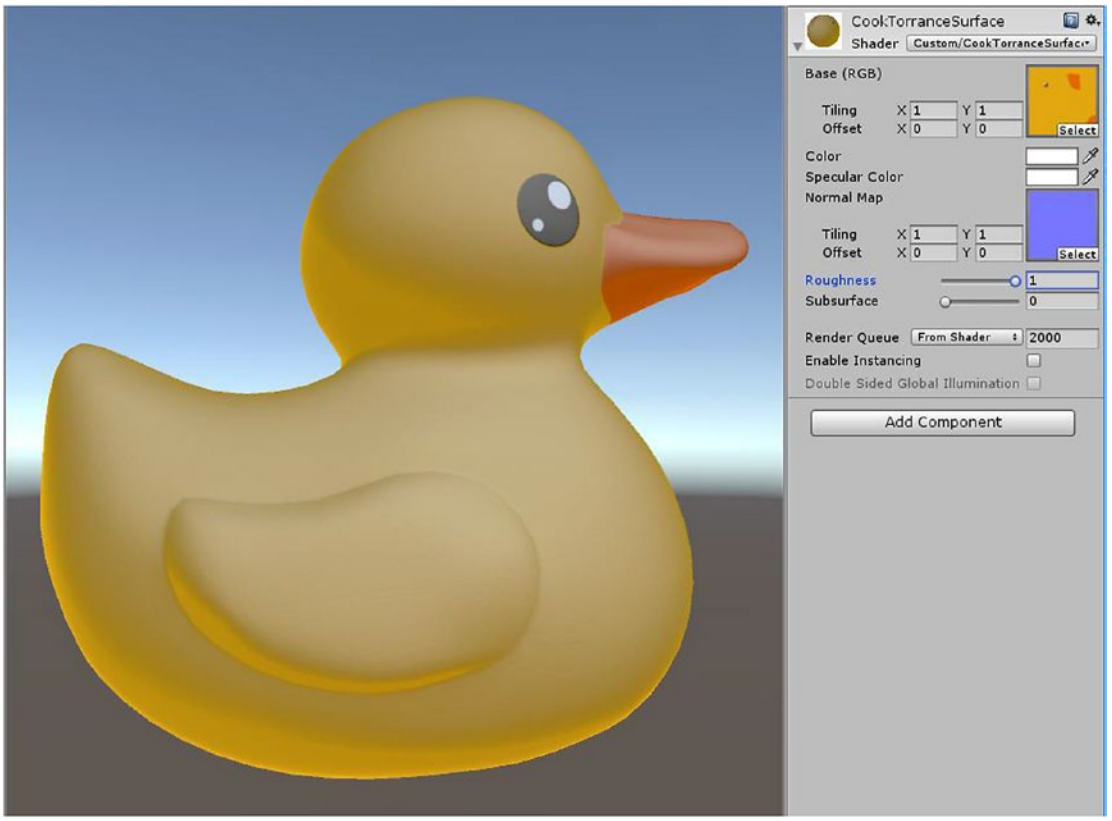

Figure 12-6. *The Frostbite version of the Disney diffuse at maximum roughness*

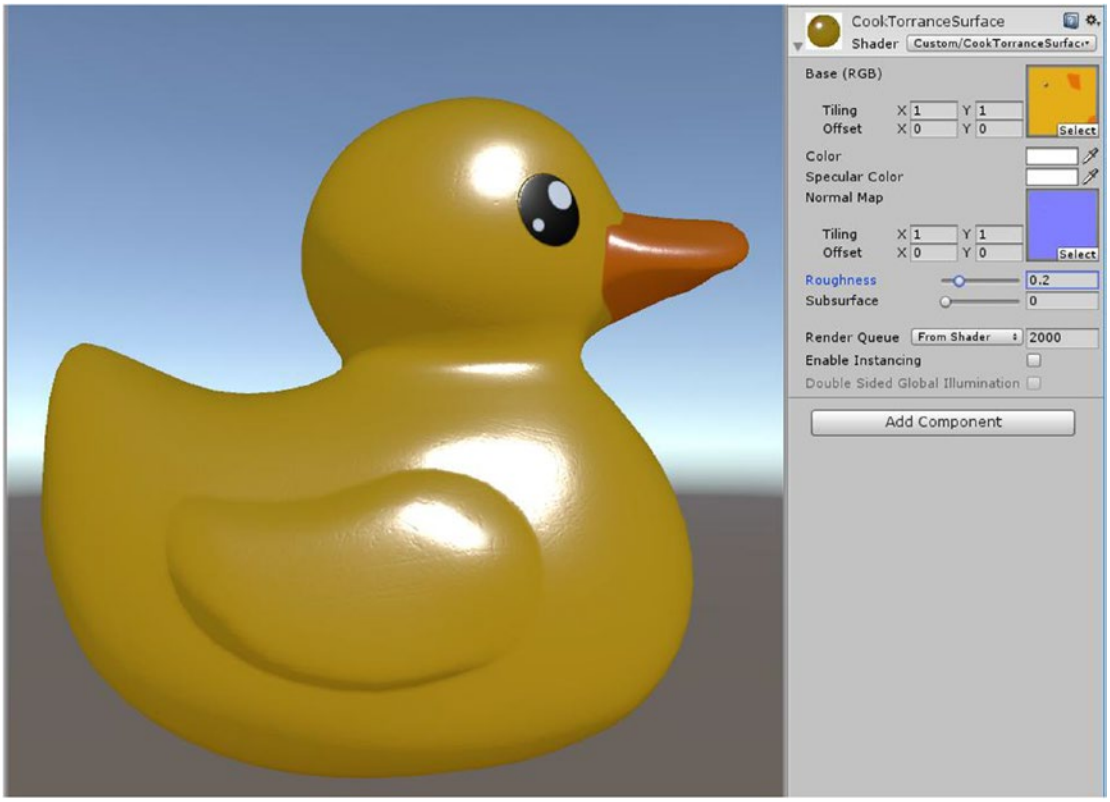

Figure 12-7. *The Frostbite version of the Disney diffuse at 0.2 roughness*

You may prefer this formulation. It's more even and cheaper than the original one. It will depend on your objectives. For our part, we have a working BRDF, including a diffuse and a specular, and we can now work on getting it integrated into Unity, more than a plain surface shader would allow. To do that, we delve into some "shader standard library" code analysis in the next chapter.

Putting It All Together

I included most parts of this shader in the chapter already. Since it's quite a beast, it's impractical to print in the book, but you can find it in the book's source code.

It is a big shader, but as machines get more powerful, shaders get more intricate and more beautiful too. Thus we get to implement better looking, if more complex, shaders.

Summary

In this chapter, we presented the CookTorrance and Disney BRDFs in detail, implementing CookTorrance specular and Disney diffuse. We also showed you how to go about choosing, gathering references, and implementing any BRDF.

Next

The next chapter shows you how to integrate this custom BRDF with the rest of the Unity shader subsystems. This includes doing some reverse-engineering of the Unity Standard shader code, which may be subject to change without warning in the future.

CHAPTER 13

Hooking Into the Standard Shader

When you create a surface or Unlit shader, it doesn't automatically take advantage of all the features of the Unity rendering system. You could reimplement them, but that would take a lot of work. Fortunately, it's possible to obtain that functionality by hacking your BRDF implementation into the Standard shader infrastructure. You can write your shader so it automatically uses Unity's Standard shader functionality. That's what we're going to do in this chapter, but keep in mind that there are no guarantees that the way to hook into these subsystems will stay the same in new versions of Unity.

This chapter explains how to hook our shader from the previous chapter to global illumination, reflection probes, etc. Some of them are going to be easy; some of them are going to be quite complex. The real objective is giving you the tools to reverse-engineer the shader standard library, and be able to redo this work on your own, should the methods change. Which they most likely will at some point.

Reverse-Engineering the Standard Shader

What we want to do is analyze the Standard shader and understand how it hooks into the functionality. It's not sufficient to look at just one file, so beware, this analysis might get long. Let's first look at one of the Standard shaders. There are various versions of the Standard shader—the plain one, the one for the Specular setup, and the one for the Roughness setup.

Let's look at the beginning part of the plain Standard shader, shown in Listing 13-1.

Listing 13-1. The Beginning of the Default Standard Shader

```
Shader "Standard"
{
    Properties
    {
        _Color("Color", Color) = (1,1,1,1)
        _MainTex("Albedo", 2D) = "white" {}

        _Cutoff("Alpha Cutoff", Range(0.0, 1.0)) = 0.5

        [...]
    }

    CGINCLUDE
        #define UNITY_SETUP_BRDF_INPUT MetallicSetup
    ENDCG
```


C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_13

```
    SubShader
    {
        Tags { "RenderType"="Opaque" "PerformanceChecks"="False" }
        LOD 300

        // ------------------------------------------------------------------
        //  Base forward pass (directional light, emission, lightmaps, ...)
        Pass
        {
            Name "FORWARD"
            Tags { "LightMode" = "ForwardBase" }

            Blend [_SrcBlend] [_DstBlend]
            ZWrite [_ZWrite]

            CGPROGRAM
            #pragma target 3.0

            // -------------------------------------

            #pragma shader_feature _NORMALMAP
            #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON _ALPHAPREMULTIPLY_ON
            #pragma shader_feature _EMISSION
            #pragma shader_feature _METALLICGLOSSMAP
            #pragma shader_feature ___ _DETAIL_MULX2
            #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
            #pragma shader_feature _ _GLOSSYREFLECTIONS_OFF
            #pragma shader_feature _PARALLAXMAP

            #pragma multi_compile_fwdbase
            #pragma multi_compile_fog
            #pragma multi_compile_instancing

            #pragma vertex vertBase
            #pragma fragment fragBase
            #include "UnityStandardCoreForward.cginc"

            ENDCG
        }
        // ------------------------------------------------------------------
        //  Additive forward pass (one light per pass)
        Pass
        {
            Name "FORWARD_DELTA"
            Tags { "LightMode" = "ForwardAdd" }
            Blend [_SrcBlend] One
            Fog { Color (0,0,0,0) } // in additive pass fog should be black
            ZWrite Off
            ZTest LEqual
             [...]
```

It has a different structure compared to the shaders we've seen in the last few chapters. To start with, it's not even a Surface shader! It's an Unlit shader, which you encountered in the first half of this book. Then you should notice the large number of passes divided into different sub-shaders.

The actual functionality of this shader is contained in the `.cginc` files, which are included in each pass. The passes in this file contain a list of on/off "switches" that enable or disable the shader code contained in the include files.

There are two types of these switches—`#pragma shader_feature` and `#pragma multi_compile_*`. To understand how this works, you need to know that shaders are generally compiled to many different versions. For example, you might not have included a Normal Map in the material, if you haven't there's no point in including that functionality in the shader used in the game. Hence, both a version without Normal Map and a version without a Normal Map are going to be compiled, and the appropriate one is going to be used automatically.

The compiled versions also depend on other variables, such as which platform you're compiling for and what capabilities your GPU has. These pragmas are a way to compile multiple versions of your shader, according to variables you choose, in addition to the built-in Unity ones that are already being used.

Shader Keywords

Our *switches* are technically called *keywords*. You can turn them on and off with `Material.EnableKeyword` and `Material.DisableKeyword` by material, or globally with `Shader.EnableKeyword` and `Shader.DisableKeyword`. Most commonly, these methods are used within the custom material Inspector GUI, which is generally written in C#.

`#pragma multi_compile VARIANT_OFF VARIANT_ON` will compile to two shaders, one with `VARIANT_OFF` defined and one with `VARIANT_ON` defined. At runtime, according to the keywords enabled, one of them will be activated. If neither keyword is activated, the first one will be used by default. You can specify more than two keywords for each `multi_compile`, and you can use underscores to mean no keyword is defined.

`#pragma shader_feature` is the same, except that unused variants of the compiled shader will not be included in the build.

`#pragma shader_feature SOME_VARIANT` is a shortcut for `#pragma shader_feature __ SOME_VARIANT`, meaning by default nothing is defined.

Some predefined `multi_compile_*` are included within the standard library, and many of them are used in the Standard shader, such as `multi_compile_fwdbase`.

Within the actual implementation code, there are blocks of code that depend on some keyword being enabled. They are be contained between `#ifdef KEYWORD` and `#else` or `#endif`. Think of it like the compiler preprocessor cutting some parts of the total shader code that we don't need, or including them if we do need them, but both versions are compiled and are available to use.

Standard Shader Structure

As stated the Standard shader is composed of two sub-shaders. The first subshader includes five passes: `FORWARD`, `FORWARD_DELTA`, `ShadowCaster`, `DEFERRED`, and `META`. The second subshader includes only the `FORWARD`, `FORWARD_DELTA`, `ShadowCaster`, and `META` passes.

The first subshader targets the Shader Model 3.0, while the second targets a previous one, 2.0. The second subshader will be able to run on older GPUs, but it lacks some features of the first. We won't analyze the second subshader.

Now let's look at each pass separately, starting from the FORWARD pass, shown in Listing 13-2.

Listing 13-2. The Complete First Sub-Shader's FORWARD Pass

```
Pass
{
    Name "FORWARD"
    Tags { "LightMode" = "ForwardBase" }

    Blend [_SrcBlend] [_DstBlend]
    ZWrite [_ZWrite]

    CGPROGRAM
    #pragma target 3.0

    #pragma shader_feature _NORMALMAP
    #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON _ALPHAPREMULTIPLY_ON
    #pragma shader_feature _EMISSION
    #pragma shader_feature _METALLICGLOSSMAP
    #pragma shader_feature ___ _DETAIL_MULX2
    #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
    #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
    #pragma shader_feature _ _GLOSSYREFLECTIONS_OFF
    #pragma shader_feature _PARALLAXMAP

    #pragma multi_compile_fwdbase
    #pragma multi_compile_fog
    #pragma multi_compile_instancing

    #pragma vertex vertBase
    #pragma fragment fragBase
    #include "UnityStandardCoreForward.cginc"

    ENDCG
}
```

We start with the pass name, then specify the Tags, which are the right tags for a ForwardBase pass, as you might remember from the chapters on Unlit shaders. Then we specify how this pass will blend together with the subsequent passes _SrcBlend and _DstBlend. These properties will be hidden in the inspector. They default respectively to 1.0 and 0.0. Using variables for this, instead of hardcoded numbers, makes it possible to make the shaded mesh fade in and fade out.

_ZWrite is also a hidden variable. After that the actual code for the pass starts, these were only the settings. We have the list of shader_feature pragmas, then the multi_compile_* pragmas, and finally we set the vertex and fragment functions, taking them from the include file UnityStandardCoreForward.cginc. There is no other code, which might make you wonder where exactly is everything implemented. The answer is within the many include files. As mentioned, in this file we only keep the switches.

The FORWARD_DELTA pass (see Listing 13-3) largely resembles the FORWARD pass, only tweaked to be appropriate to a ForwardAdd pass. The vertex and fragment functions are different, but they come from the same file.

Listing 13-3. The Complete First Sub-Shader's FORWARD_DELTA Pass

```
Pass
{
    Name "FORWARD_DELTA"
    Tags { "LightMode" = "ForwardAdd" }
    Blend [_SrcBlend] One
    Fog { Color (0,0,0,0) } // in additive pass fog should be black
    ZWrite Off
    ZTest LEqual

    CGPROGRAM
    #pragma target 3.0

    #pragma shader_feature _NORMALMAP
    #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON _ALPHAPREMULTIPLY_ON
    #pragma shader_feature _METALLICGLOSSMAP
    #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
    #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
    #pragma shader_feature ___ _DETAIL_MULX2
    #pragma shader_feature _PARALLAXMAP

    #pragma multi_compile_fwdadd_fullshadows
    #pragma multi_compile_fog

    #pragma vertex vertAdd
    #pragma fragment fragAdd
    #include "UnityStandardCoreForward.cginc"

    ENDCG
}
```

Now for the ShadowCaster pass (see Listing 13-4). Since the objective is different, this pass is also quite different. It still includes the switches that make a difference for rendering shadows. Vertex and fragment functions come from a different file from the previous two passes.

Listing 13-4. The Complete First Sub-Shader's ShadowCaster Pass

```
Pass {
    Name "ShadowCaster"
    Tags { "LightMode" = "ShadowCaster" }

    ZWrite On ZTest LEqual

    CGPROGRAM
    #pragma target 3.0

    #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON _ALPHAPREMULTIPLY_ON
    #pragma shader_feature _METALLICGLOSSMAP
    #pragma shader_feature _PARALLAXMAP
    #pragma multi_compile_shadowcaster
    #pragma multi_compile_instancing
```

```
    #pragma vertex vertShadowCaster
    #pragma fragment fragShadowCaster

    #include "UnityStandardShadow.cginc"

    ENDCG
}
```

We're going to skip the DEFERRED pass, because we're not going to include one in our shader. Next, we're looking at the META pass (see Listing 13-5), which is supposed to gather information for lightmapping and global Illumination. It follows the same patterns as the previous ones.

Listing 13-5. The Complete First Sub-Shader's META Pass

```
Pass
{
    Name "META"
    Tags { "LightMode" = "Meta" }

    Cull Off

    CGPROGRAM
    #pragma vertex vert_meta
    #pragma fragment frag_meta

    #pragma shader_feature _EMISSION
    #pragma shader_feature _METALLICGLOSSMAP
    #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
    #pragma shader_feature ___ _DETAIL_MULX2
    #pragma shader_feature EDITOR_VISUALIZATION

    #include "UnityStandardMeta.cginc"
    ENDCG
}
```

We're going to skip the second subshader, because it's more of the same, but with fewer features. Finally, the last lines of the shader define a fallback and a custom inspector editor (see Listing 13-6).

Listing 13-6. End of the Standard Shader

```
    FallBack "VertexLit"
    CustomEditor "StandardShaderGUI"
}
```

Later on, we're going to make our own custom GUI, so keep in mind where to set it up. Thus, we have finished our look at the different passes. The second subshader passes are pretty much the same, so we're going to move on and skip them.

Chasing Down Shader Keywords

Now that we have skimmed the whole shader, let's try digging into one of those shader keywords. We're going to chase down every occurrence of `_NORMALMAP`, to see in practice how the keywords work within the shader code. By searching for it within the entire Standard shader codebase, we can find where the Normal Map functionality is implemented. We find it in many files:

- `UnityStandardConfig.cginc`
- `UnityStandardCore.cginc`
- `UnityStandardCoreForwardSimple.cginc`
- `UnityStandardInput.cginc`

Let's pick a simple example from `UnityStandardCoreForwardSimple.cginc` (see Listing 13-7). It's one of the `structs` used in the `vertex` function. When we compile with the `_NORMALMAP` keyword defined, the members declared within the `#ifdef _NORMALMAP` are included. Otherwise, they are left out.

Listing 13-7. Use of _NORMALMAP Keyword Within the Standard Shader Includes

```
struct VertexOutputBaseSimple
{
    UNITY_POSITION(pos);
    float4 tex                          : TEXCOORD0;
    half4 eyeVec                        : TEXCOORD1; // w: grazingTerm
    half4 ambientOrLightmapUV           : TEXCOORD2; // SH or Lightmap UV
    SHADOW_COORDS(3)
    UNITY_FOG_COORDS_PACKED(4, half4) // x: fogCoord, yzw: reflectVec
    half4 normalWorld                   : TEXCOORD5; // w: fresnelTerm

#ifdef _NORMALMAP
    half3 tangentSpaceLightDir          : TEXCOORD6;
    #if SPECULAR_HIGHLIGHTS
        half3 tangentSpaceEyeVec        : TEXCOORD7;
    #endif
#endif
#if UNITY_REQUIRE_FRAG_WORLDPOS
    float3 posWorld                     : TEXCOORD8;
#endif

    UNITY_VERTEX_OUTPUT_STEREO
};
```

For any keyword that you want to find out more about, this is the procedure to follow.

As you might imagine, it is a tricky business to handle all of this complexity, only armed with conditional compiling. And yet, there is no way out of it until a better system or shader language comes along.

Implementing the Standard Shader Substitute

Now we're going to start the boilerplate-filled, and thankless, process of copying/pasting our way to our BRDF hooking into the Standard shader functionality. What we need to do is copy and paste the entire Standard shader, only changing the part that does the shading. If it sounds primitive, it's because it is. Nevertheless, it works.

There are a bunch of functions that handle the lighting calculations within the Standard shader. We basically want to overwrite those, to get our custom lighting function called by the Standard shader machinery when it triggers calculation of a final shaded color.

Within `UnityPBSLighting.cginc`, there is a choice of lighting function depending on the platform capabilities (see Listing 13-8). We want to make it so our lighting function is always chosen.

Listing 13-8. BRDF Function Choice

```
// Default BRDF to use:
#if !defined (UNITY_BRDF_PBS) // allow to explicitly override BRDF in custom shader
    // still add safe net for low shader models, otherwise we might end up with shaders
failing to compile
    #if SHADER_TARGET < 30
        #define UNITY_BRDF_PBS BRDF3_Unity_PBS
    #elif defined(UNITY_PBS_USE_BRDF3)
        #define UNITY_BRDF_PBS BRDF3_Unity_PBS
    #elif defined(UNITY_PBS_USE_BRDF2)
        #define UNITY_BRDF_PBS BRDF2_Unity_PBS
    #elif defined(UNITY_PBS_USE_BRDF1)
        #define UNITY_BRDF_PBS BRDF1_Unity_PBS
    #elif defined(SHADER_TARGET_SURFACE_ANALYSIS)
        // we do preprocess pass during shader analysis and we dont actually care about brdf
        as we need only inputs/outputs
        #define UNITY_BRDF_PBS BRDF1_Unity_PBS
    #else
        #error something broke in auto-choosing BRDF
    #endif
#endif
```

As you can see, there are three different lighting function choices within the Standard shader:

- `BRDF3_Unity_PBS` is a BlinnPhong used on platform that doesn't support the Shader Model 3.0.
- `BRDF2_Unity_PBS` is a simplified CookTorrance.
- `BRDF1_Unity_PBS` is a modified version of the Disney BRDF.

They are contained within `UnityStandardBRDF.cginc`, and we'll need to fit our BRDF in one or more of these functions.

Let's look at the type signatures they have in common:

```
half4 BRDF1_Unity_PBS ( half3 diffColor, half3 specColor, half oneMinusReflectivity,
                        half smoothness, half3 normal, half3 viewDir,
                        UnityLight light, UnityIndirect gi)
```

They all take the same eight arguments. We'll need to package our BRDF in a function that takes these same arguments and returns a color.

There are various built-in Unity defined keywords that control various aspects of the lighting function used, such as UNITY_BRDF_GGX, UNITY_COLORSPACE_GAMMA, SHADER_API_MOBILE, etc. We're only going to worry about a couple of them as an example. But keep in mind that, depending on the platforms you want to target with the shader, you may want to also support other keywords needed by different platforms in order to get better performance.

So, in a different file, we need to implement this choice based on SHADER_TARGET and the functions it will call. We will reduce it to call just one function, for the sake of brevity. Then, we'll implement a function containing our BRDF that uses the above function signature, so it can be slotted in with the rest of the code.

Implementing the BRDF

Let's create a file called TestOverwriteInclude.cginc in the project. We're going to do the bare minimum to test our thesis about how to overwrite the Standard shader lighting function. In Listing 13-9, you can see that it boils down to including UnityStandardBRDF.cginc, defining UNITY_BRDF_PBS to correspond to our lighting function, and declaring the lighting function with the signature we used previously.

Listing 13-9. Minimum Code Needed to Overwrite the Lighting Function

```
#ifndef UNITY_OVERWRITE_LIGHTING_INCLUDED
#define UNITY_OVERWRITE_LIGHTING_INCLUDED

#include "UnityStandardBRDF.cginc"

#define UNITY_BRDF_PBS TestBRDF_PBS

half4 TestBRDF_PBS ( half3 diffColor, half3 specColor, half oneMinusReflectivity,
                     half smoothness, half3 normal, half3 viewDir,
                     UnityLight light, UnityIndirect gi)
{
    return half4(1, 0, 0, 1);
}

#endif
```

To have a shader to use this with, we need to create a new shader file and copy and paste the code from Standard.shader. Then we need to include the TestOverwriteInclude.cginc file before the line #include "UnityStandardCoreForward.cginc" in both the FORWARD and the FORWARD_DELTA passes.

And that's it. Select the shader for use with a material, and your result should look like Figure 13-1, as we are only returning a single color from the lighting function.

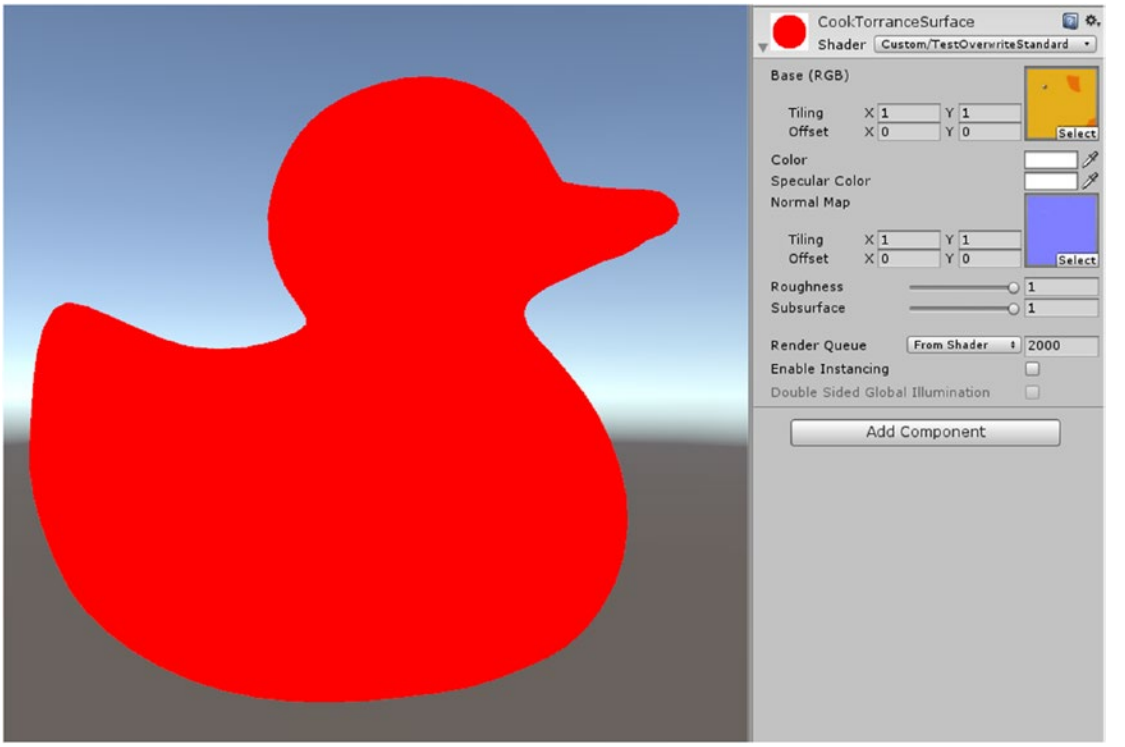

Figure 13-1. *The results of returning a single color from the lighting function*

What happens is that, by including our overwritten file first, we're overriding the `#define` of the function `UNITY_BRDF_PBS`, which is contained within the Standard shader `includes` and the lighting function choice. As we're not overriding anything else, the rest of our copied Standard shader file is still using the default Standard shader code. If we wanted to change more things, such as modifying how the vertex shader works, then we'd need to copy and overwrite more functions.

The modified `FORWARD` pass is shown in Listing 13-10.

Listing 13-10. FORWARD Pass Overwriting the Standard Lighting Function

```
Pass
{
    Name "FORWARD"
    Tags { "LightMode" = "ForwardBase" }

    Blend [_SrcBlend] [_DstBlend]
    ZWrite [_ZWrite]

    CGPROGRAM
    #pragma target 3.0

    #pragma shader_feature _NORMALMAP
    #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON _ALPHAPREMULTIPLY_ON
    #pragma shader_feature _EMISSION
```

```
    #pragma shader_feature _METALLICGLOSSMAP
    #pragma shader_feature ___ _DETAIL_MULX2
    #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
    #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
    #pragma shader_feature _ _GLOSSYREFLECTIONS_OFF
    #pragma shader_feature _PARALLAXMAP

    #pragma multi_compile_fwdbase
    #pragma multi_compile_fog
    #pragma multi_compile_instancing

    #pragma vertex vertBase
    #pragma fragment fragBase
    #include "TestOverrideInclude.cginc"
    #include "UnityStandardCoreForward.cginc"

    ENDCG
}
```

Now that we know this method works, we need to adapt our custom BRDF from the last chapter to this format.

Making the BRDF Shader

We start with copying and pasting the file `Standard.shader` again, to the file `CustomStandardShader.shader`. Then we need to customize it with our properties, variables, and `includes`. The properties can be copied from the Surface shaders from the previous chapters, but take care not to remove the many extra properties that the Standard shader infrastructure can use.

The result should look like Listing 13-11.

Listing 13-11. The new custom Standard shader's properties

```
Shader "Custom/CustomStandardShader" {
    Properties {
        _Roughness ("Roughness", Range(0,1)) = 0.5
        _Subsurface ("Subsurface", Range(0,1)) = 0.5

        _SpecColor("Specular", Color) = (1, 1, 1)
        _Color("Color", Color) = (1,1,1,1)
        _MainTex("Albedo", 2D) = "white" {}

        _Cutoff("Alpha Cutoff", Range(0.0, 1.0)) = 0.5

        _Glossiness("Smoothness", Range(0.0, 1.0)) = 0.5
        _GlossMapScale("Smoothness Scale", Range(0.0, 1.0)) = 1.0
        [Enum(Metallic Alpha,0,Albedo Alpha,1)] _SmoothnessTextureChannel ("Smoothness
        texture channel", Float) = 0

        [ToggleOff] _SpecularHighlights("Specular Highlights", Float) = 1.0
        [ToggleOff] _GlossyReflections("Glossy Reflections", Float) = 1.0
```

```
        _BumpScale("Scale", Float) = 1.0
        _BumpMap("Normal Map", 2D) = "bump" {}

        _Parallax ("Height Scale", Range (0.005, 0.08)) = 0.02
        _ParallaxMap ("Height Map", 2D) = "black" {}

        _OcclusionStrength("Strength", Range(0.0, 1.0)) = 1.0
        _OcclusionMap("Occlusion", 2D) = "white" {}

        _EmissionColor("Color", Color) = (0,0,0)
        _EmissionMap("Emission", 2D) = "white" {}

        _DetailMask("Detail Mask", 2D) = "white" {}

        _DetailAlbedoMap("Detail Albedo x2", 2D) = "grey" {}
        _DetailNormalMapScale("Scale", Float) = 1.0
        _DetailNormalMap("Normal Map", 2D) = "bump" {}

        [Enum(UV0,0,UV1,1)] _UVSec ("UV Set for secondary textures", Float) = 0

        // Blending state
        [HideInInspector] _Mode ("__mode", Float) = 0.0
        [HideInInspector] _SrcBlend ("__src", Float) = 1.0
        [HideInInspector] _DstBlend ("__dst", Float) = 0.0
        [HideInInspector] _ZWrite ("__zw", Float) = 1.0
    }
```

A particularly useful part of the shader is the CGINCLUDE block. Within it, we need to declare what setup (specular, metallic, etc.) we're going to use. But it can also serve as a place to put some of our variables from the CookTorrance shader, so that the passes and the custom lighting function will have access to them (see Listing 13-12).

Listing 13-12. CGINCLUDE Block

```
CGINCLUDE
    #define UNITY_SETUP_BRDF_INPUT MetallicSetup
    float _Roughness;
    float _Subsurface
ENDCG
```

Now we need to create another include file, called CustomBRDFOverrideInclude.cginc, and copy and paste the previous TestOverrideInclude.cginc include file to it. The lighting function will still need all the supporting functions from the CookTorranceSurface.shader file, which we should paste before the lighting function.

The lighting function from that shader needs to be modified to work with the current arguments, as we don't get direct access to the SurfaceOutput struct anymore. We need to look at the circumstances in which UNITY_BRDF_PBS is called to understand the parameters. For that, we need to read the UnityPBSLighting.cginc file. The function we're interested in is LightingStandardSpecular, which you can see in Listing 13-13.

Listing 13-13. The LightingStandardSpecular Function

```
inline half4 LightingStandardSpecular ( SurfaceOutputStandardSpecular s, half3 viewDir,
                                        UnityGI gi)
{
    s.Normal = normalize(s.Normal);

    // energy conservation
    half oneMinusReflectivity;
    s.Albedo = EnergyConservationBetweenDiffuseAndSpecular (s.Albedo, s.Specular, /*out*/
    oneMinusReflectivity);

    // shader relies on pre-multiply alpha-blend (_SrcBlend = One, _DstBlend =
    OneMinusSrcAlpha)
    // this is necessary to handle transparency in physically correct way - only diffuse
    component gets affected by alpha
    half outputAlpha;
    s.Albedo = PreMultiplyAlpha (s.Albedo, s.Alpha, oneMinusReflectivity, /*out*/
    outputAlpha);

    half4 c = UNITY_BRDF_PBS (s.Albedo, s.Specular, oneMinusReflectivity, s.Smoothness,
    s.Normal, viewDir, gi.light, gi.indirect);
    c.a = outputAlpha;
    return c;
}
```

Note that the albedo is not coming straight from the texture, but it's already getting some energy conservation calculation applied, before getting to our function. This may disturb our result, depending on how much it changes, but changing this would require copy pasting more files, so we're just going to live with it.

For the same reason, we don't want to customize the `SurfaceOutput` struct, so the shader will use the one already included, which you can see in Listing 13-14.

Listing 13-14. The SurfaceOutputStandardSpecular Struct

```
struct SurfaceOutputStandardSpecular
{
    fixed3 Albedo;      // diffuse color
    fixed3 Specular;    // specular color
    fixed3 Normal;      // tangent space normal, if written
    half3 Emission;
    half Smoothness;    // 0=rough, 1=smooth
    half Occlusion;     // occlusion (default 1)
    fixed Alpha;        // alpha for transparencies
};
```

Since our lighting function is going to be called from a completely different place, it might have been awkward to use our variables from it. We solved this problem earlier, by putting them in the `CGINCLUDE` block. If you want a more elegant solution, you may want to store the values in the `SurfaceOutputStandardSpecular struct` somehow.

One method would be changing the label of a property we're not going to use, such as `Glossiness`, and use it instead to store `_Roughness`. But be aware that anywhere in the Standard shader includes code, these values may be going through some processing and end up changed. This is why we're sticking to the simpler solution and just making the variables available to use directly.

Going back to our custom lighting function, we can copy and paste the custom lighting function from the CookTorrance shader. Then we need to correct it for our needs. The normal is normalized outside of this function, so we don't need it to normalize any more. Then we need to use `normal` and `diffColor` instead of the equivalent variables names from before (see Listing 13-15).

Listing 13-15. Temporary Custom Lighting Function

```
half4 CustomDisneyCookTorrance_PBS ( half3 diffColor, half3 specColor, half
                                      oneMinusReflectivity,
                                     half smoothness, half3 normal, half3 viewDir,
                                     UnityLight light, UnityIndirect gi)
{
    viewDir = normalize ( viewDir );
    float3 lightDir = normalize ( light.dir );

    float3 halfV = normalize(lightDir+viewDir);
    float NdotL = saturate( dot( normal, lightDir ));
    float NdotH = saturate( dot( normal, halfV ));
    float NdotV = saturate( dot( normal, viewDir ));
    float VdotH = saturate( dot( viewDir, halfV ));
    float LdotH = saturate( dot( lightDir, halfV ));

    float3 diff = DisneyDiff(diffColor, NdotL,  NdotV, LdotH, _Roughness);
    float3 spec = CookTorranceSpec(NdotL, LdotH, NdotH, NdotV, _Roughness, _SpecColor);
    float3 diff2 = (DisneyFrostbiteDiff(NdotL, NdotV, LdotH, _Roughness) * diffColor)/PI;
    float3 firstLayer = ( diff + spec * _SpecColor) * _LightColor0.rgb;
    float4 c = float4(firstLayer, 1);

    return c;
}
```

We are not using `specColor` as it is, because it's too dark for our BRDF. To keep things simple, we'll just keep using the `_SpecColor` variable. Remember to set the custom lighting function to this function, using `#define UNITY_BRDF_PBS CustomDisneyCookTorrance_PBS` at the top of the file. At this point, you should be able to set your shader material to `Custom/CustomStandardShader` and be able to see our BRDF being applied. But we're not done, because we're not using reflections and the other capabilities of the Standard shader.

Modifying our BRDF to do that will be a bit of a fight. The way to do it is to copy and paste the `BRDF1_Unity_PBS` function from the `UnityStandardBRDF.cginc` file. It includes a lot of functionality, which we don't really want to worry about, so we're going to simplify it a little bit. We're going to keep the `UNITY_HANDLE_CORRECTLY_NEGATIVE_NDOTV` and `_SPECULARHIGHLIGHTS_OFF` functionality.

As our variable naming convention and the Standard shader naming convention clashes, you're going to see both `VdotL` and `nl` as variable names. That should help you determine which lines come from the Standard shader, and which lines were copied straight from our BRDF. In Listing 13-16, you can see the final result. The most interesting part is the final color assembling, where the global illumination and the reflections are put together with the BRDF.

Listing 13-16. Final Custom Lighting Function

```
half4 CustomDisneyCookTorrance_PBS ( half3 diffColor, half3 specColor, half oneMinusReflectivity,
                                     half smoothness, half3 normal, half3 viewDir,
                                     UnityLight light, UnityIndirect gi)
{
    half3 halfDir = Unity_SafeNormalize (light.dir + viewDir);

#define UNITY_HANDLE_CORRECTLY_NEGATIVE_NDOTV 0

#if UNITY_HANDLE_CORRECTLY_NEGATIVE_NDOTV
    half shiftAmount = dot(normal, viewDir);
    normal = shiftAmount < 0.0f ? normal + viewDir * (-shiftAmount + 1e-5f) : normal;
    half nv = saturate(dot(normal, viewDir));
#else
    half nv = abs(dot(normal, viewDir));    // This abs allow to limit artifact
#endif

    float VdotH = saturate( dot( viewDir, halfDir ));
    float LdotH = saturate( dot( light.dir, halfDir ));

    half nl = saturate(dot(normal, light.dir));
    half nh = saturate(dot(normal, halfDir));

    half lv = saturate(dot(light.dir, viewDir));
    half lh = saturate(dot(light.dir, halfDir));

    float3 diffuseTerm = DisneyDiff(diffColor, nl,  nv, LdotH, _Roughness);
    float3 specularTerm = CookTorranceSpec(nl, LdotH, nh, nv, _Roughness, _SpecColor);

#if defined(_SPECULARHIGHLIGHTS_OFF)
    specularTerm = 0.0;
#endif
    specularTerm *= any(specColor) ? 1.0 : 0.0;

    half grazingTerm = saturate(smoothness + (1-oneMinusReflectivity));
    half3 color =   diffColor * (gi.diffuse + light.color * diffuseTerm)
                    + specularTerm * light.color
                    + gi.specular * FresnelLerp (specColor, grazingTerm, nv);

    return half4(color, 1);
}
```

One difficultly was that the Standard shader treats roughness differently from the way our custom lighting function does, so we fell back on keeping the smoothness and roughness separate, and only using the smoothness for the reflections part of the shader. This has the side effect that we're keeping around two values that should be describing the same things, and that the specular is not getting automatically reduced when roughness increases.

The final result is not perfect, as you can see in Figure 13-2, but it's similar enough to our target for our purposes.

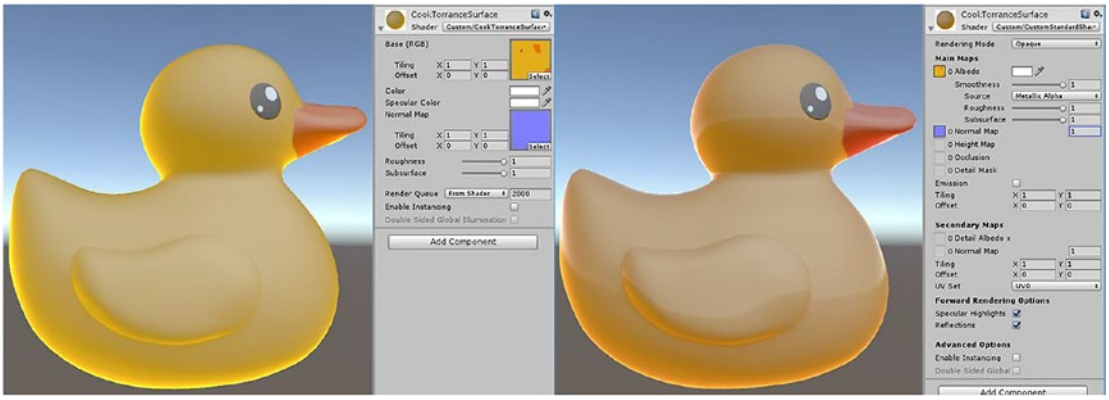

Figure 13-2. *Left, our custom CookTorrance Surface shader, right, our Custom standard shader, with reflections*

Making a GUI for the BRDF Shader

Now we have our shader, we should make a nice GUI for it. Again, we should copy, paste, and modify the file that implements this functionality for the Standard shader. The file is called `StandardShaderGUI.cs`, and it can be found with the shader includes in the `Editor/` folder.

We need two new value fields, one for `_Roughness` and one for `_Subsurface`. We're going to keep the Smoothness field and use it to turn the specular reflections on and off.

I'm assuming you named this new file `CustomShaderGUI.cs` and changed the class name to match. The first step is to add these two lines to the `Styles` section:

```
public static GUIContent roughnessText = new GUIContent("Roughness", "Roughness value");
public static GUIContent subsurfaceText = new GUIContent("Subsurface", "Subsurface value");
```

They are declaring the `GUIContent` (basically the labels) that we'll use when we're showing the variable field. Then, you need to declare two extra `MaterialProperty` for them:

```
MaterialProperty roughness = null;
MaterialProperty subsurface = null;
```

Next, you need to add these two lines to the `FindProperties(MaterialProperty[] props)` function:

```
roughness = FindProperty("_Roughness", props);
subsurface = FindProperty("_Subsurface", props);
```

They are connecting the `MaterialProperty` to the actual variables names used in the shader. Finally, add these two lines to the end of the `DoSpecularMetallicArea()` function:

```
m_MaterialEditor.ShaderProperty( roughness, Styles.roughnessText, indentation );
m_MaterialEditor.ShaderProperty( subsurface, Styles.subsurfaceText, indentation );
```

They finally draw the fields that we've been setting up so far. To use this custom inspector for your shader, change the final line of the shader to this:

```
CustomEditor "CustomShaderGUI"
```

That should be all. You should automatically see this inspector editor when you select a mesh that uses this shader (see Figure 13-3). The code is way too long to reproduce here, so remember you can check it in the book's source code.

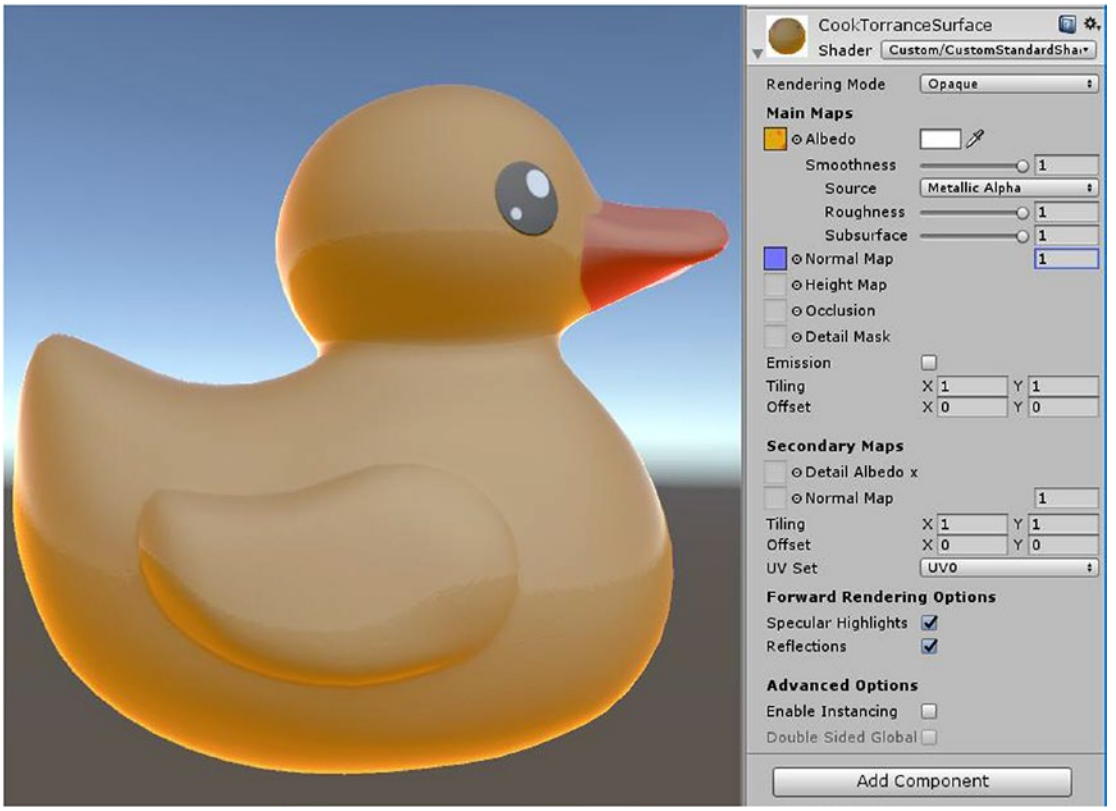

Figure 13-3. *The custom shader inspector*

Summary

This chapter covered how to hack the Standard shader code to use a BRDF of our choice. This give you access to the extensive functionality included in the Standard shader that a plain Surface shader would lack. You could possibly reimplement it, but this way is much quicker.

Next

The next chapter describes and implements some useful advanced techniques to better simulate complex light behaviors that our BRDF can't do in real time without big approximations, such as subsurface scattering and translucency.

CHAPTER 14

Implementing Advanced Techniques

As mentioned, GPUs are not currently fast enough to simulate the finer points of simulating the physics of light in real-time. But you still want to achieve realistic and impressive shading in your games. To work around this problem there is an entire industry worth of tricks, hacks, and approximations designed to give you almost the true result, but with a fraction of the computational expense.

This chapter explains the process of choosing and implementing such techniques in Unity. We're choosing a technique that can be implemented quickly enough to get into details. There are many other worthwhile techniques around, such as volumetric fog, area light, various kinds of subsurface scattering, etc., but they deserve much more space than we could give them here.

Where to Find Techniques

There are many sources from which you can get techniques descriptions, some of which are listed in Chapter 18. Some of them are books, some of them are conference talks, and some of them are just articles or code repositories on the Internet. Devising techniques generally requires mathematical and algorithmic insight, and time to spend on R&D, so they are most commonly invented by graphics programming researchers or graphics programmers working in well-funded studios.

For the technique we're going to implement in this chapter, we're going back to a Game Developer Conference talk from 2011. That year, DICE presented a technique to implement translucency cheaply, which they used in their then engine, Frostbite 2. They also published a chapter about it within the *GPU PRO 2* book.

Implementing Translucency

One classic real-world example of translucency in action is holding your hand between you and a strong light, such as the sun. You'll see the light penetrate the flesh and make the contours of your fingers look red. In general, any time a translucent material is between the point of view and a light, a part of the light will come through to the other side, where it will reach the camera.

Some support of translucency is necessary for skin shading, but it's not enough. Skin is composed of many layers which influence the outcome. This translucent shader is more appropriate for homogeneous materials such as wax, which don't have different interacting layers.

Translucency can't be modeled with a BRDF. The light hitting a translucent material is scattered through the surface, exiting at points that can be quite far from where the ray first hit the material. To describe this behavior accurately, we'd need a bidirectional scattering distribution function (BSDF), or a bidirectional surface scattering reflectance distribution functions (BSSRDF). Meaning that the approximations that optics

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_14

give us are not enough, and we need to go deeper. The medium through which light is scattering, below the surface, is influencing the light transport. To describe that, we'd need volumetric scattering. But this technique simplifies that a lot: it only concerns itself with calculating the light that will get through the mesh, by deviating the direction of the light according to a few properties. This extra light is then added to the diffuse term.

Properties

This translucency technique depends on these properties:

- The `Power` and the `Scale` factors scale the intensity of translucency, in different measure.
- The `Subsurface Distortion` factor influences how distorted the direction of the outgoing light will be. It works by shifting the surface normal.
- The `Thickness` map represents the thickness of the mesh at a certain point. It's supposed to be built by inverting the surface normals, calculating ambient occlusion, and then inverting the result of the ambient occlusion calculation.
- The `Subsurface Color` is not strictly needed, but it's going to make it easier for us to see the result of the translucency calculation, by choosing a color that contrasts with our test model. In the original paper, an ambient value is used in place of choosing a color.

You can see what these properties look like in code in Listing 14-1.

Listing 14-1. The Properties Necessary for Implementing this Technique

```
_Thickness ("Thickness (R)", 2D) = "white" {}
_Power ("Power Factor", Range(0.1, 10.0)) = 1.0
_Distortion ("Distortion", Range(0.0, 10.0)) = 0.0
_Scale ("Scale Factor", Range(0.0, 10.0)) = 0.5
_SubsurfaceColor ("Subsurface Color", Color) = (1, 1, 1, 1)
```

Implementation

You should make a new surface shader and copy/paste the `CookTorranceSurface` shader to it, changing the path to `Custom/TranslucencyShader`. Then, you'll need to add the new properties to the properties in the shader and declare the respective variables.

You'll need to add a `Thickness` member to the `SurfaceOutputCustom` struct, and then sample the thickness map, and apply it to this member in the surface function (see Listing 14-2).

Listing 14-2. The Necessary Changes to SurfaceOutputCustom and the Surface Function

```
struct SurfaceOutputCustom {
    float3 Albedo;
    float3 Normal;
    float3 Emission;
    float Thickness;
    float Alpha;
};

void surf (Input IN, inout SurfaceOutputCustom o) {
    float4 c = tex2D (_MainTex, IN.uv_MainTex) * _ColorTint;
    o.Albedo = c.rgb;
```

```
    o.Thickness = tex2D (_Thickness, IN.uv_MainTex).r;
    o.Normal = UnpackNormal( tex2D ( _BumpMap, IN.uv_MainTex ) );
    o.Alpha = c.a;
}
```

After that, the implementation of this technique boils down to just a few lines of code (see Listing 14-3). First we shift the light direction, then we calculate the dot product of the inverse of that direction and the view direction, and finally scale it to the properties. Then we use the thickness map and the extra color with the dot product, which will use the color of the translucency pixel by pixel. Finally, we add the result to the diffuse term.

Listing 14-3. Calculating the Translucency Color

```
float3 translucencyLightDir = lightDir + s.Normal * _Distortion;
float translucencyDot = pow(saturate(dot(viewDir, -translucencyLightDir)), _Power) * _Scale;
float3 translucency = translucencyDot * s.Thickness * _SubsurfaceColor;

float3 diff = DisneyDiff(s.Albedo, NdotL,  NdotV, LdotH, _Roughness) + translucency;
```

While this matches the technique example code quite close, we simplified it a bit. We skipped on the attenuation, as it's somewhat awkward to extract it from the custom lighting function infrastructure. We also skipped multiplying the translucency value for albedo and light color, as that happens anyway when we add the diffuse term to the specular term.

You can take a look at the final result in Figure 14-1. Note that the thickness map is not the actual inverted ambient occlusion map. It's more of an example to see that the shader does work.

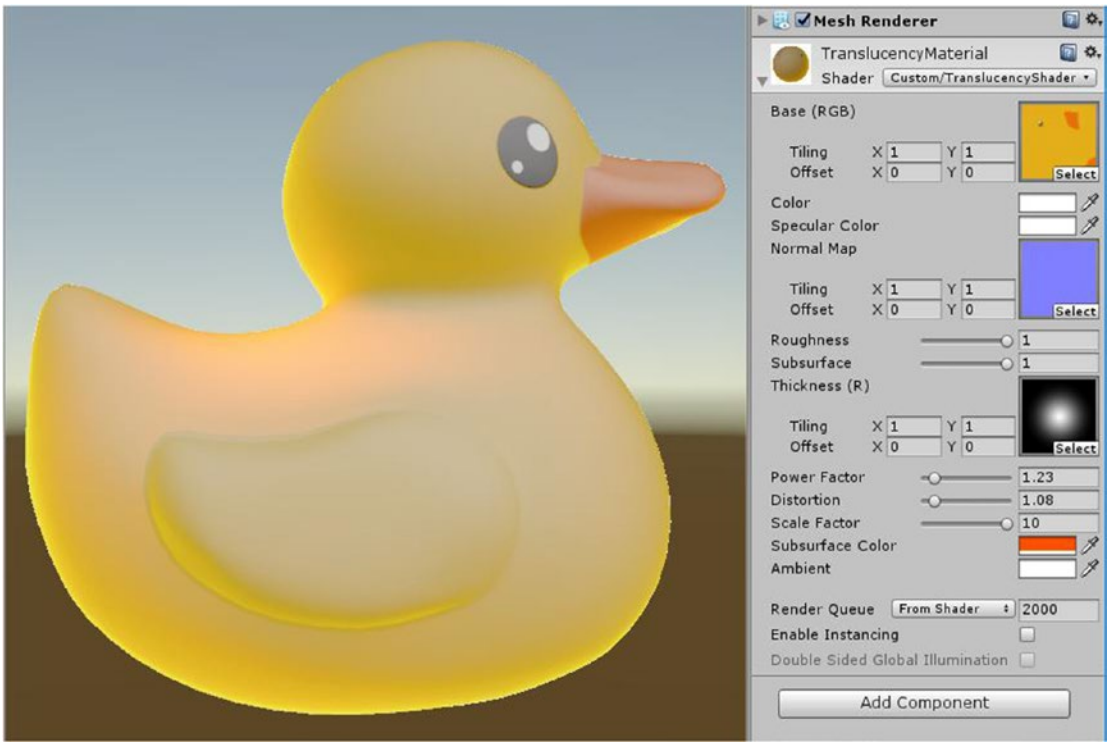

Figure 14-1. *The CookTorrance shader with added translucency*

This was our advanced technique. There are many more around for you to lay your hands on. Many will work outside of the BRDF, and then get piled on top it somehow, as this one does. That is not necessarily a correct way to add complex effects that can't be described by BRDFs, but it tends to work well enough, depending on your objectives.

Next, we'll cover briefly how the mechanisms behind image based lighting and reflection probes work in Unity.

Real-Time Reflections

As of Unity 2017.1, Unity includes an implementation of *real-time reflections,* but this hasn't always been the case. In Unity 4, there was no such built-in functionality, and if you wanted reflections, you needed to implement them from scratch. In this section, we'll summarize the necessary steps to implement reflections in a real-time renderer.

Although you may never have to implement them yourself, there are some advantages to knowing how it's done. The included implementation of image based lighting and reflections works well with the Unity Standard BRDF, but it won't necessary work well with any new BRDF you might want to implement. Since the BRDF it uses is most likely going to be different from the one you want to implement, the result of putting together your BRDF, and the reflections resulting from calculations meant for a different BRDF, will be subtly (or even noticeably) wrong.

That said, the Standard shader uses three different BRDFs depending on platform. While they might have implemented different cubemap processing for each BRDF, it could also mean that they haven't and they're fine with them being slightly off.

I think it is worthwhile to have an underlying knowledge of how reflection probes work underneath, because they are a necessary part of physically based shading. Should you ever want to build your own renderer, you'll need to implement this functionality from scratch, and Unity's reflection probes hide amazingly well how much work implementing them really is.

What Is a Cubemap

We haven't talked much about cubemaps up to now, because in Unity 5.0 and higher, we only need to deal with them to set a skybox, and the default one was fine for our purposes.

Cubemaps are fundamental to implementing reflections. A *cubemap* is basically a capture of all the incoming light into a certain point, which should sound familiar, as it is pretty much the definition of irradiance (see Figure 14-2).

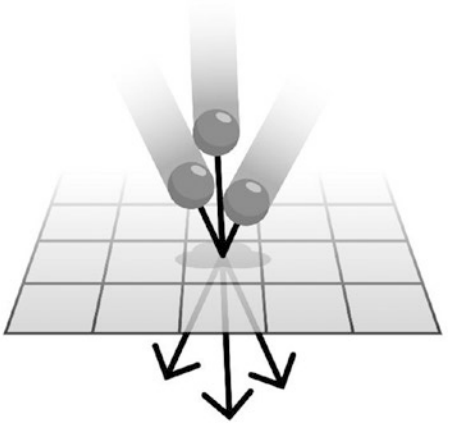

Figure 14-2. *Irradiance, incoming light from all directions into a point*

A key difference between cubemaps and irradiance, as we've used it up to now in BRDFs, is that irradiance is measured on a hemisphere, as it is on a surface. Cubemaps, on the other hand, are 360° captures of the surrounding scene. They are obtained by rendering the scene six times, in six different directions (+X, -X, +Y, -Y, +Z, and -Z), at field of view 90°, into a square. Then those renders need to be stitched together into an image, according to cubemap conventions (see Figure 14-3). There are many possible formats, which expect different faces to be in different places. Exporting the cubemap so the faces match can be a very confusing part of implementing a cubemap capture.

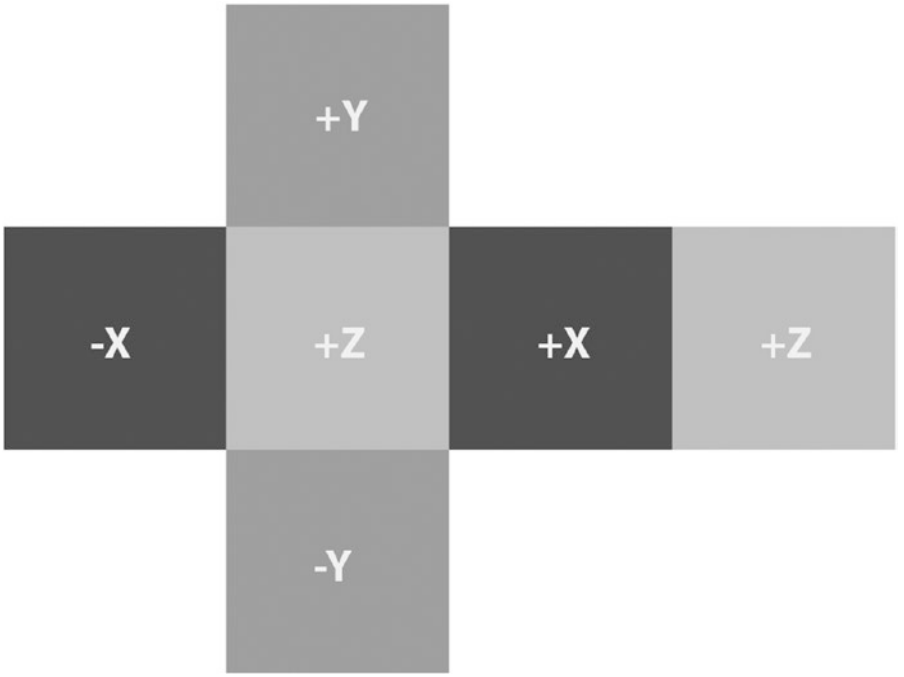

Figure 14-3. *Layout of a cubemap*

Cubemaps can be used to represent light coming from infinite distances, such as a skybox, or they can be used to represent local incoming light. In the latter case, additional information about the size of the area needs to be stored outside of the cubemap.

What Are Reflection Probes

Reflection probes are basically an automated way of capturing cubemaps in real-time and processing them, so they can be used for reflections.

In Chapter 13, we added reflections just by adding the content of the built-in variable `gi.specular`, multiplied by a Fresnel factor, to our diffuse and specular terms:

```
half3 color =   diffColor * (gi.diffuse + light.color * diffuseTerm)
                + specularTerm * light.color
                + gi.specular * FresnelLerp (specColor, grazingTerm, nv);
```

This looks deceptively simple, but quite a lot is going on underneath. What happens behind the scenes is that a cubemap of the current scene is captured from the point of view of a reflection probe. This cubemap represents the incoming light into a point. Then it needs to be processed according to whether it's for diffuse or specular reflection, and according to your BRDF(s).

If your BRDF supports roughness, the reflection will be blurred at a higher roughness. That can be achieved by using different mips and having different convolution applied to the cubemap mips, depending on roughness value.

Another necessary feature is blending between the reflection probes in a scene; otherwise, the jump between two reflection probes would be jarring.

Finally, *box projection* is an extremely useful feature, as otherwise in many situations the reflections wouldn't match the position of what is being reflected, as they're supposed to. [1]

Evaluating a Cubemap

After capturing, the cubemap cannot be used for reflections as it is. It needs to be prefiltered according to the roughness value, and with an eye to scale its color values, to make it behave more realistically. Using the cubemap as it is might overwhelm the rest of the shading and definitely isn't physically correct.

Filtered environment maps are supposed to store *reflected radiance,* which is obtained by integrating all incoming light, multiplied by the BRDF, and by the cosine between the normal and the lighting. [2]

In the course notes for the talk *Physically Based Shading in Call of Duty: Black Ops,* Dimitar Lazarov details one way of doing that scaling, which he calls *environment map normalization*. [3]

According to the course notes, the filtering is two-fold:

- For each mip level, a Gaussian filter with increasing size should be used, corresponding to higher roughness. So you can get the appropriate level of roughness in a reflection by choosing the right mip. This depends on how the BRDF handles roughness.
- The cubemap needs to be normalized to be used for reflections, which is done by dividing the radiance (see Figure 14-4) at each *texel* (one pixel in a texture) by the average irradiance at the place of capture (this can be done entirely at runtime, when sampling the cubemap).

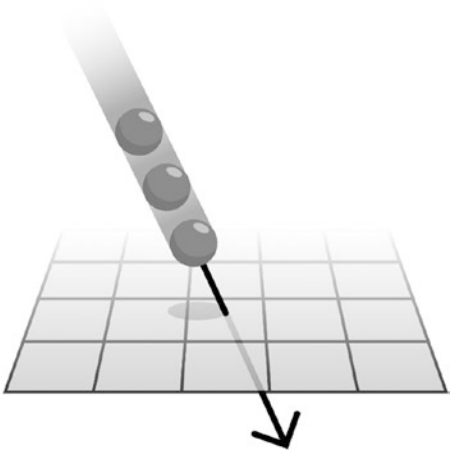

Figure 14-4. *Radiance, all light incoming into to one point, from one direction*

[1]For a good overview of blending between cubemaps and box projection, read: https://seblagarde.wordpress.com/2012/09/29/image-based-lighting-approaches-and-parallax-corrected-cubemap/

[2]For more information on how filtering for environment maps has evolved in time, see http://jankautz.com/courses/GameCourse/05_EnvMaps.pdf from SIGGRAPH 2006

[3]http://blog.selfshadow.com/publications/s2013-shading-course/lazarov/s2013_pbs_black_ops_2_notes.pdf

Irradiance and radiance are related: irradiance is equal to the integral of radiance, over the hemisphere incoming light directions, weighted by $n \cdot l$

$$Irradiance = \int_{\Omega} Radiance(l)(n \cdot l)dl$$

To use the cubemap in the shader of a material to implement reflections, you need to choose which mip to sample, according to the roughness of the material.

When you do sample it using the correct mip, you'll need to scale its value, according to how much light is incoming at that point. As we said, this scaling can be done either at runtime or offline. We're going to look at how to do it at runtime in a custom lighting shader.

Cubemap Processing Programs

As mentioned, it's normal now to do the cubemap filtering at runtime, but there are also various programs that can process a cubemap so it can be used for IBL. Some of them are open source, some are closed source but freeware and some are commercial.

In the case of the open source ones, you'll be able to read and modify the processing code. You can learn from it and adapt it to the specific BRDF you want. This can be a convenient testbed, before implementing the same mechanism in a game engine.

- `cmftStudio`: Open source
- `Knald Lys`: Commercial
- `IBLBaker`: Open source
- `CubeMapGen`: Open source but old

We won't go into further details about this, but this should be enough to get you started, should you want to implement your own equivalent of reflection probes at some point. For some helpful math advice on how to calculate necessary parts of cubemap processing, see http://www.rorydriscoll.com/2012/01/15/cubemap-texel-solid-angle/.

Let's make these concepts less abstract by looking at some code. The following code used to work in Unity 4, but it might not work in recent Unity versions. It's included just to give you an idea of the kind of code required, rather than to serve as a reference implementation.

Before being able to use this technique, you need to set up light probes or something equivalent. Unity used to offer a function that taps into the light probes, as a result encoding the irradiance at every light probe in the scene into spherical harmonics, which we can use.

After setting light probes up, we can start coding a custom lighting shader that supports reflections. First we need to declare a new property and the variable for it in the custom lighting shader where we're going to use the cubemap:

```
_Cube("Reflection Map", Cube) = "white" {}

[...]

samplerCUBE _Cube;
```

Then we need to make a way to get the extra `float3` of the cubemap color into the custom lighting function. We can do that by adding an extra member in `SurfaceOutputCustom`:

```
struct SurfaceOutputCustom {
   [...]
   fixed3 CubeColor;
};
```

We also need to calculate the normal in World Space, and the reflection direction in World Space in the vertex shader, and pass it on to the custom lighting function. To do that, remember to add them to the `Input` struct:

```
struct Input {
    fixed2 uv_MainTex;
    fixed3 worldNormal;
    fixed3 worldRefl;
    INTERNAL_DATA
};
```

Having done this setup, we can sample the cubemap in the `surf` function.

To do that, we need to match roughness value of the BRDF to a mipmap level. Say we have 10 mipmap levels available, and the roughness values range from 0, to 1. That means mip-0 represents roughness 0, mip-1 will represent roughness 0.1, and so on. `iblLod` in the following code stands for the operation of finding the right mip-level:

```
fixed3 skyR = WorldReflectionVector(IN, o.Normal);
fixed4 cubeColor = texCUBElod(_Cube, fixed4(skyR, iblLOD));
```

Before assigning the sampled cubemap color to the `CubeColor SurfaceOutputCustom` member, we need to rescale it using the irradiance contained in the spherical harmonics:

```
fixed3 local_irradiance = saturate(ShadeSH9 (fixed4(IN.worldNormal,1.0)));
```

Finally, we set the scaled color to the `CubeColor` member:

```
o.CubeColor = cubeColor / local_irradiance;
```

In the custom lighting function, we need to complete the missing steps so we can use the reflection color. In the course notes for *Physically Based Shading in Call of Duty: Black Ops,* there is a heavily optimized environment BRDF function that you could adapt for your needs, or see also the footnotes [4] and [5].

In the custom lighting function, we calculate the indirect specular by passing the necessary data to an environment BRDF function, and then we add it to our final color:

```
fixed3 indSpec = EnvironmentBRDF(_RoughnessS, NdotV, linearSpecColor) * s.CubeColor; //Fresnel?
[...]
return fixed4(firstLayer + indSpec), 1);
```

[4] A good compendium about IBL, included many code samples, taken from a number of papers: https://chetanjags.wordpress.com/2015/08/26/image-based-lighting/

[5] See Section 4.9: Image Based Lights: *https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf*

We haven't included a Fresnel in the indirect specular, but using one would improve it.

That was most of the key necessary shader code, but to build an entire reflection system, you'll still need to implement box projection, a system to position probes and capture cubemaps, and a system to blend between probes. Hopefully this was enough to give you a working idea on how to proceed.

Summary

This chapter went through the process of implementing a technique that approximates a phenomenon that we wouldn't be able to implement in real-time (until GPUs step up even more, at the least). While doing that, you saw where and how to pick and implement techniques that may be useful to you. The chapter also included a brief overview of how image based lighting is implemented.

Next

The next chapter talks about what you need to consider to make your shader convenient for artists to use.

PART III

Shader Development Advice

Now you are a seasoned Unity shader developer, so it's time to reflect on the art of writing good code and usable shaders.

You will learn the most common usability mistakes that developers make that put off artists from using their shaders. You will learn how to debug and profile problematic and slow shaders.

You will learn how ubershaders came to be, and why they're a solution to some big problems encountered in building complex shader systems.

You will learn where to find up-to-date, bleeding edge information about the game and movie industry, so you can keep up with the never-ending tide of progress.

CHAPTER 15

Making Shaders Artists Will Use

While chasing down physically based perfection, the user experience from the point of view of the artists is something that's easy to overlook. Complex BRDFs have complex settings, and the artists whom you're going to give the shaders are not likely to know as much as you do about how to use them.

Descriptive names are often not a priority, and sometimes we end up with downright confusing names. One bright example is the Roughness input for the Unity Standard shader. The concept is commonly known as Roughness, but in the Unity Standard shader it's called Smoothness on the user side, and to compound the confusion, it's called `Glossiness` on the code side. It's not a huge deal, but naming things in a confusing way has never helped anyone.

This chapter explains a few useful tips to make sure your effort of making great lighting models includes taking your end users, the artists, into account.

The UX of the Disney BRDF

Most of the currently BRDFs used by game engines and offline renderers, such as Blender's Cycles, are inspired by Disney's efforts to streamline a BRDF so it can be used for most materials and is easy for the legions of artists at Disney to use.

When you are part of a small studio, it's easy enough to walk over to the artists' side of the office and personally walk them through the way you intended them to use the shader you just made. When your studio employs more than 15 or so people, that becomes pretty much impossible. What you need to do then is make the shader controls as intuitive as possible, so that they are almost self-explanatory. But how do you achieve that? Let's look at what you should *not* do, which will help to get you on the right path.

Typical Problem #1: Too Many Settings

Each setting included in a shader requires some cognitive load from the users. It's easy to squander all your user's goodwill in the name of flexibility and granular control. As the number of possible settings piles up, they're almost guaranteed not to be used, because the time needed to figure out their effect by tinkering will be too high.

It's particularly easy to get to that point when you're composing BRDFs without consolidating their inputs. For example, assume you're using a specular BRDF with roughness, and you want to use the OrenNayar BRDF as its diffuse. OrenNayar will need a separate roughness variable, a diffuse roughness, unless you find a way to have the specular and the diffuse roughness match up, in this way consolidating the two variables into one.

Even though I do extol the virtues of the Disney BRDF, when you implement all of it, it does go near to the "too many settings" point (see Figure 15-1).

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_15

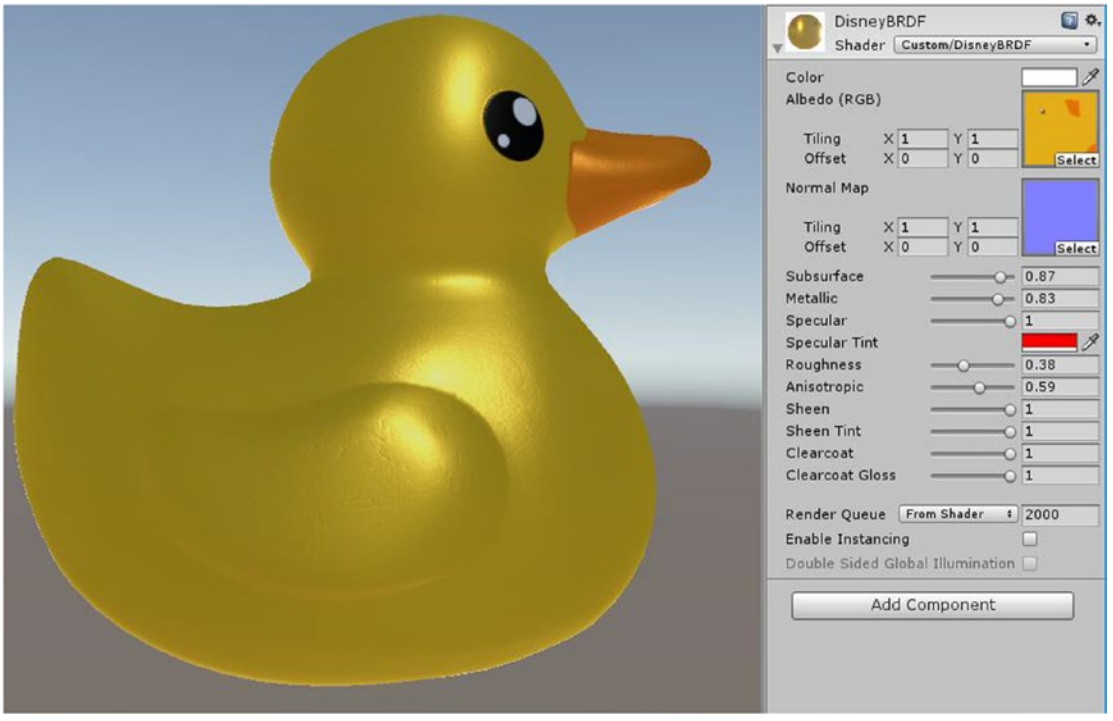

Figure 15-1. *The Disney BRDF, flirting with having too many settings*

Typical Problem #2: The Effect of a Setting Is Unclear

The effects of some variation in settings might depend on the value of another one to be visible. As an example, in the Ashikhmin Shirley BRDF, you have four settings—`Rs`, `Rd`, `nu`, and `nv` (see Figure 15-2).

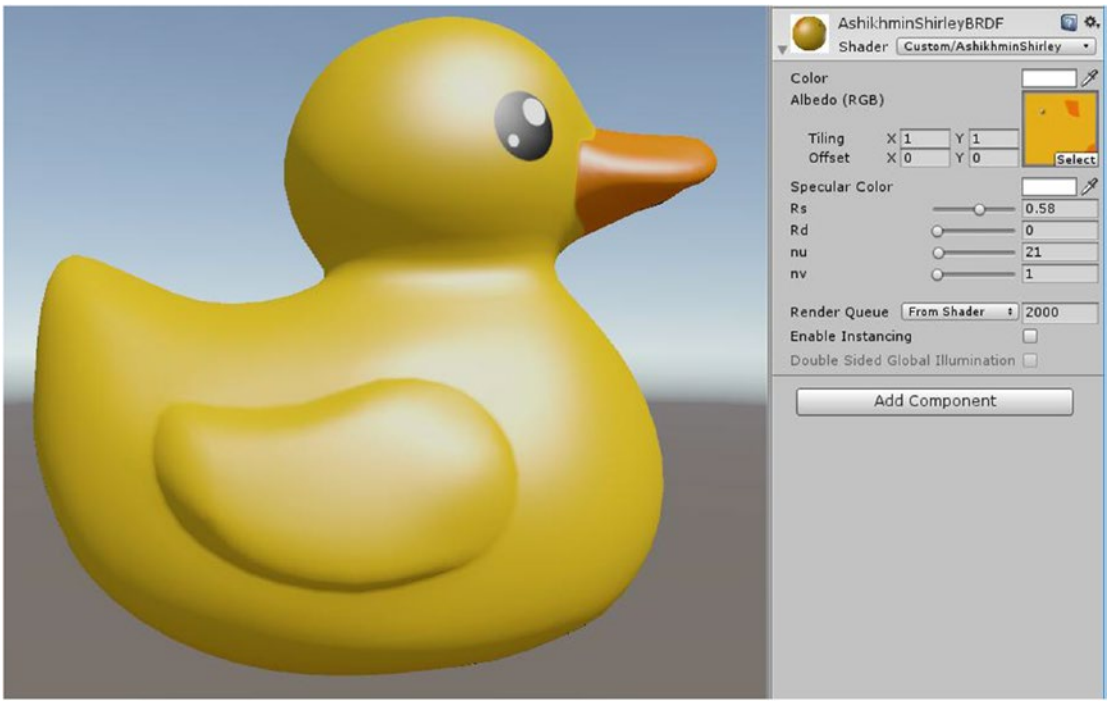

Figure 15-2. *Note how the naming of the variables gives no clue of their roles*

Those names mean pretty much nothing to anyone who hasn't read the paper. And that's not a great start. But worse, it's hard to figure out how `Rd` is useful just by tinkering with it.

Also, if a setting doesn't have any apparent effect in most cases, you probably want to either set it to a fixed value in code or get rid of it entirely. Either way, the artist shouldn't have to deal with it. And this leads us directly to the next common problem: dependencies between settings.

Typical Problem #3: Settings Dependencies

This problem is when, in order to achieve one desired effect, you have to change multiple properties. This chain of changes is error prone and needs to be documented, and it also takes space in the artist's mind, causing distraction and frustration.

This might not be entirely solvable, but it can at least be better documented, listing the expected case in which the interaction of light setting, and certain values or ranges of shader setting, will lead to some setting not making any difference. Something to aspire to.

Typical Problem #4: Unclear Compacting of Textures

If you're just using three channels out of a texture, you might be very tempted to put in the remaining channel something else, such as specularity or roughness. This is all good, as long as it's clear what you're doing. If it's not clear, that channel will probably get random data, or be ignored, possibly creating scene-specific bugs in various places in the game, which can be surprisingly hard to debug.

A while ago, there used to be some ambiguity in the Unity Standard shader, but they duly corrected it. Still, in the Specular setup, you need to remember to choose where the smoothness value is held, between the alpha from the albedo texture and the one for the specular texture (see Figure 15-3).

Figure 15-3. *One potential bug-creating setting, if overlooked*

Typical Problem #5: Strange Ranges

When offering a setting that can take a range of values, unless the specific range values are relevant to the artists in some way, it's better to stick to a 0 to 1 range. Arbitrary range numbers can be confusing and result in more errors. See Figure 15-4.

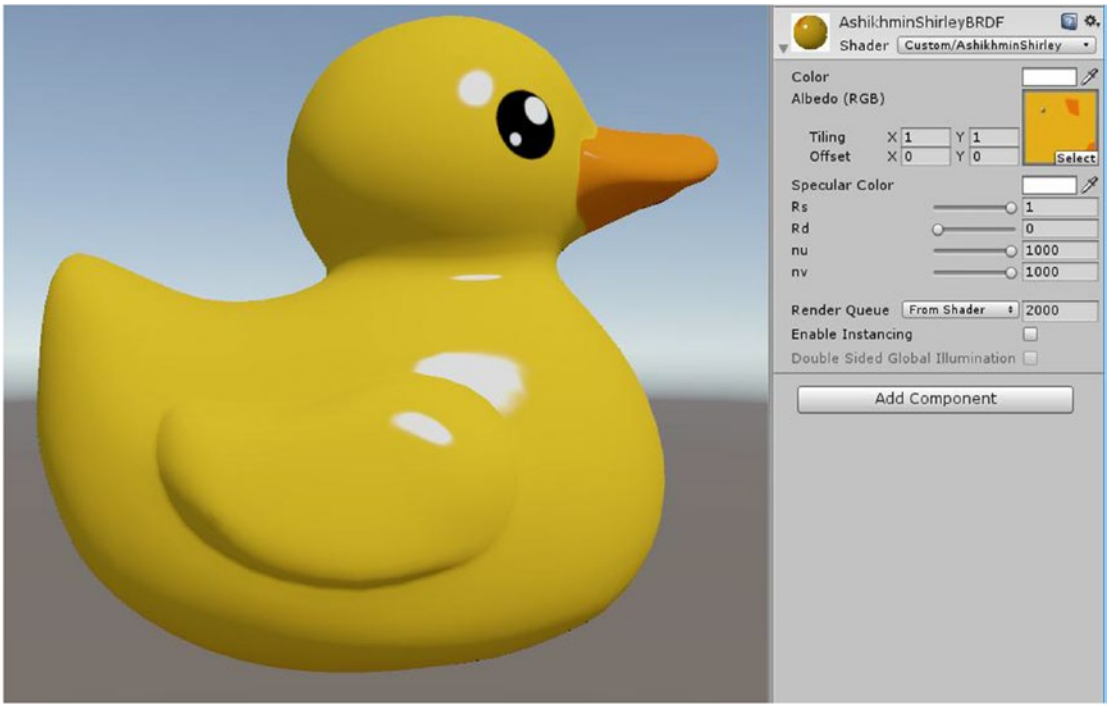

Figure 15-4. *Note the differing ranges in this shader—0 to 1 and 1 to 1000*

Positive Example: Disney BRDF in Blender

The Disney BRDF has been implemented in many renderers, both real-time and offline. Offline renderers are more focused on artists in general and have fewer restrictions because they don't need to be real time. That makes them a good source of inspiration to make our real-time shading implementation more realistic and user friendly.

Many offline renderers are costly, sometimes complex to set up, and might require expensive 3D programs to support them, such as Maya or 3DS Max. There is one that is easily obtainable and includes an implementation of the principled Disney BRDF. It's the open source physically based render included in Blender, called *Cycles*.

In Figure 15-5, you can see the Disney BRDF as implemented in Cycles. Note the self-explanatory settings names that they use and how they help make this BRDF friendlier to use than many others.

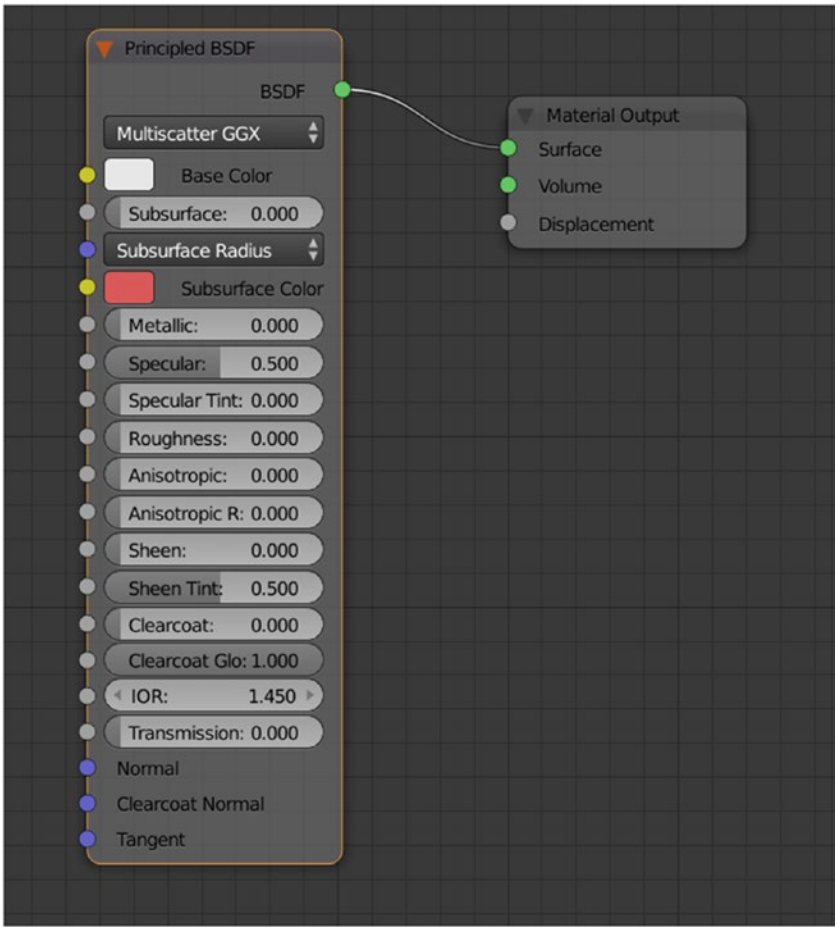

Figure 15-5. *The Disney BRDF implemented in Blender Cycles*

Summary

This chapter listed some common usability problems that arise when developing lighting models and discussed a few ways to avoid them.

Next

The next chapter talks about complexity again, but this time it's on the code side. We'll talk about how ubershaders are a way to help you deal with the many different variations needed within a shader, and show you how the Unity Standard shader uses this technique.

CHAPTER 16

Complexity and Ubershaders

As you may well have noticed from the chapter on hacking a custom lighting model within the Standard shader, once you start to build up functionality upon functionality and need to support many different platforms with very different capabilities, the complexity of shader code can get very high, very fast.

A characteristic that most shader systems share is the need to be flexible, while the language itself doesn't really provide much flexibility. Unity is experimenting with writing shaders in C#, so this may change in the future. But for now, your best bet is to somehow abuse the built-in features of the Cg language, namely *pragmas* and *ifdefs*, and build what are generally known as *ubershaders*.

What Is an Ubershader?

An ubershader is a monolithic shader that includes all the functionality you might need and turns features on and off at compile time (not at runtime; that'd be very expensive!).

It's what happens when you don't want to keep rewriting slightly different shaders over and over again. You start by writing from scratch any shader that you might need in your game. That way you'll end up with a lot of repeated code. Repeated code makes keeping all the different shaders up to date hard, and you have to consciously make the effort of backporting any improvement made on one shader to all the old ones that use similar code. Meanwhile, the artists end up with too many shaders to choose from, resulting in wasted time and confusion.

To solve these problems, you'll likely want to consolidate common functionality between all your shaders in `.cginc` files. While doing that, you'll start using `ifdefs` so you can maximize the reuse of the shader code.

The result of this process of consolidation is keeping all the functionality in the library code and only using single `.shader` files as a repository of on and off switches. You need a `.shader` file to use your shader in the scene.

The Standard Shader

The Standard shader is a ubershader. You can gather that from the large number of include files, and the fact that the main shader file is built out of turning `ifdefs` on and off (see Listing 16-1).

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_16

Listing 16-1. The First Pass Included in the First Sub-Shader of the Standard Shader

```
SubShader
{
    Tags { "RenderType"="Opaque" "PerformanceChecks"="False" }
    LOD 300

    // ------------------------------------------------------------------
    //  Base forward pass (directional light, emission, lightmaps, ...)
    Pass
    {
        Name "FORWARD"
        Tags { "LightMode" = "ForwardBase" }

        Blend [_SrcBlend] [_DstBlend]
        ZWrite [_ZWrite]

        CGPROGRAM
        #pragma target 3.0
        #pragma shader_feature _NORMALMAP
        #pragma shader_feature _ _ALPHATEST_ON _ALPHABLEND_ON _ALPHAPREMULTIPLY_ON
        #pragma shader_feature _EMISSION
        #pragma shader_feature _METALLICGLOSSMAP
        #pragma shader_feature ___ _DETAIL_MULX2
        #pragma shader_feature _ _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
        #pragma shader_feature _ _SPECULARHIGHLIGHTS_OFF
        #pragma shader_feature _ _GLOSSYREFLECTIONS_OFF
        #pragma shader_feature _PARALLAXMAP

        #pragma multi_compile_fwdbase
        #pragma multi_compile_fog
        #pragma multi_compile_instancing

        #pragma vertex vertBase
        #pragma fragment fragBase
        #include "UnityStandardCoreForward.cginc"
        ENDCG
    }
```

As you might remember, this isn't even a surface shader, it's an Unlit shader. We have so far kept away from the most intricate parts of the Standard shader code, because editing them can be quite time -consuming. But we're going to take a look at some of that here.

You can see in Listing 16-2 the code that chooses which vertex and fragment functions are going to be used. You might think, "but that happens in the main shader, when using the pragmas `#vert` and `#frag`," and you'd be right, and wrong. That is one choice, but this is another choice, enforced at compile time. There is a set of `frag` and `vert` functions for the `UNITY_STANDARD_SIMPLE` version of the shader, and one for the normal, more feature-ful one. They are hosted in different `.cginc` files.

Listing 16-2. Choosing Which Vertex and Fragment Functions to Use

```
#ifndef UNITY_STANDARD_CORE_FORWARD_INCLUDED
#define UNITY_STANDARD_CORE_FORWARD_INCLUDED

#if defined(UNITY_NO_FULL_STANDARD_SHADER)
#   define UNITY_STANDARD_SIMPLE 1
#endif

#include "UnityStandardConfig.cginc"

#if UNITY_STANDARD_SIMPLE
    #include "UnityStandardCoreForwardSimple.cginc"
    VertexOutputBaseSimple vertBase (VertexInput v) { return vertForwardBaseSimple(v); }
    VertexOutputForwardAddSimple vertAdd (VertexInput v) { return vertForwardAddSimple(v); }
    half4 fragBase (VertexOutputBaseSimple i) : SV_Target { return fragForwardBaseSimple
    Internal(i); }
    half4 fragAdd (VertexOutputForwardAddSimple i) : SV_Target { return fragForwardAddSimple
    Internal(i); }
#else
    #include "UnityStandardCore.cginc"
    VertexOutputForwardBase vertBase (VertexInput v) { return vertForwardBase(v); }
    VertexOutputForwardAdd vertAdd (VertexInput v) { return vertForwardAdd(v); }
    half4 fragBase (VertexOutputForwardBase i) : SV_Target { return fragForwardBaseInternal(i); }
    half4 fragAdd (VertexOutputForwardAdd i) : SV_Target { return fragForwardAddInternal(i); }
#endif

#endif // UNITY_STANDARD_CORE_FORWARD_INCLUDED
```

When we use pragmas to define which functions we are going to use, we are not referring directly to them, but we are directing the compile-time control flow to another choice, between another two sets of functions.

It's impressive code, but it can be hard to follow and devilish to modify. Anytime you change something somewhere in the `include` files, you need to consider what `ifdef` it will impact, and if you're not careful you might break paths different from the one you're editing at the moment. If you compound that with lacking tools such as unit testing, you can see how tricky this ubershader business can get.

This pretty much sums up every ubershader written with a language such as Cg, which hasn't been built to deal with this much complexity. When Cg was invented, it was an alternative to GPU assembly languages, and the shader length was limited.

What Causes Complexity in Shaders?

The complexity in shaders comes from having to support many different platforms that can only accommodate part of the features and might need to use cheaper approximations. The difference in platforms is quite vast, since Unity supports many versions of Direct3D, OpenGL, OpenGL ES, and Vulkan.

The "magic" of Unity is the relative ease of supporting vastly different platforms, from consoles, to mobile phones, to high-end PCs. But in reality, each of these platforms requires a lot of attention from the shader side, unless you're using at least part of the Unity shader system.

There are also various platform-specific bugs that need to be worked around. You can find traces of them sprinkled through the Unity shader codebase, such as the example in `Lighting.cginc` shown in Listing 16-3.

Listing 16-3. Example of Platform-Specific Bug in `Lighting.cginc`.

```
// NOTE: some intricacy in shader compiler on some GLES2.0 platforms (iOS) needs 'viewDir' & 'h'
// to be mediump instead of lowp, otherwise specular highlight becomes too bright.
```

Ubershader Gotchas

As mentioned, this style of shader writing requires great care, because a change made anywhere in the `includes` can snowball and create problems for any of the supported platforms.

Another consequence of the complexity is the great number of shaders that end up getting generated, that can then make your project compilation slower. The entire ubershader is compiled to a great number of single platform-specific shaders, one for each combination of the various defines and features that can be turned on and off, producing a combinatorial explosion of variants.

Ubershader Advantages

So far I've mostly pointed out how ubershaders are complex and hard to deal with. But the only real alternative to them is the proliferation of the number of independent shaders, and that's not a good thing either.

Having a great number of shaders leads to confusion in the pipeline, as artists need to be guided in the choice of which one to use. Having an ubershader leads to having a few shader choices and dealing with the rest through options for the few shaders.

Having an ubershader can save coding time, because you only need to edit a few files in order to update all the shader variations when you make an improvement.

The industry has converged on ubershaders, which most likely means that they are the best tradeoff between ease of use and complexity that we have at the moment.

Summary

This chapter provided an overview of ubershaders, including what they are, how they are used, and some tips on how to control the tangle.

Next

The next chapter explains how to proceed with shader debugging, including some of the tools available and some of the things to watch out for.

CHAPTER 17

When Shading Goes Wrong

You have written a great lighting model that you're proud of, but it goes horribly wrong on your test device. Or on some user's device. Or you have an unexplainable bug on your machine, which goes away when you deploy to your platform. If any of this has happened to you, welcome to the wonderful and painful world of shader debugging.

Debugging gets increasingly hard the more steps of separation you have from your development platform. Debugging when your target platform is the same as the PC you're using is easier. If you're targeting a different device you have to compile and deploy to it, and then access the debugger remotely. Also, the fragmentation of your platforms matters; for example, you might be developing an Android game, and since Android is notoriously fragmented, you may get a report of graphics bugs on devices that are unavailable to you, making debugging an impossibility.

This chapter discusses some of the tools and techniques you can use to debug and profile your shaders.

Common Tricks

Something that you can do without any debugging tools is edit the shader you're debugging, so you're visualizing the inputs to the different parts of the shader.

For example, your shader may be wrong, or the data that is the input to it may be wrong. Maybe the Normal Map is not recognized as such, or the 3D model has been broken by exporting to FBX.

Especially when your shader is working in other scenes or with other models, checking that the data is actually correct, is a good first step. Otherwise, you might spend days chasing a nonexistent bug. Discontinuities among the normals (such as "pinches") are a common problem with models and very problematic for the shading. You might need to get Unity to recalculate the normals. Or you could use a specific shader to show the normals of the model instead of the shaded model and see for yourself what they look like (see Listing 17-1).

Listing 17-1. A shader which visualizes the normals

```
Shader "Custom/UnlitShaderDebugNormals"
{
        SubShader
        {
                Pass
                {
                        CGPROGRAM
                        #pragma vertex vert
                        #pragma fragment frag
```


C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_17

```
                        #include "UnityCG.cginc"

                        struct appdata
                        {
                                float4 vertex : POSITION;
                                float3 normal : NORMAL;
                        };

                        struct v2f
                        {
                                float4 vertex : SV_POSITION;
                                float3 worldNormal : TEXCOORD0;
                        };

                        v2f vert (appdata v)
                        {
                                v2f o;
                                o.vertex = UnityObjectToClipPos(v.vertex);
                                float3 worldNormal = UnityObjectToWorldNormal(v.normal);
                                o.worldNormal = worldNormal;
                                return o;
                        }

                        fixed4 frag (v2f i) : SV_Target
                        {
                                return float4(i.worldNormal*0.5+0.5, 1.0f);
                        }
                        ENDCG
                }
        }
}
```

This shader will not calculate any lighting; it will just visualize the normals. See Figure 17-1 for how that looks. As you might have noticed, this is an Unlit shader, because making a surface shader that does this is quite inconvenient. You'd have to implement the normals visualizing as a custom lighting model, and you'd have to keep at least one member in the Input data structure and implement the global illumination function.

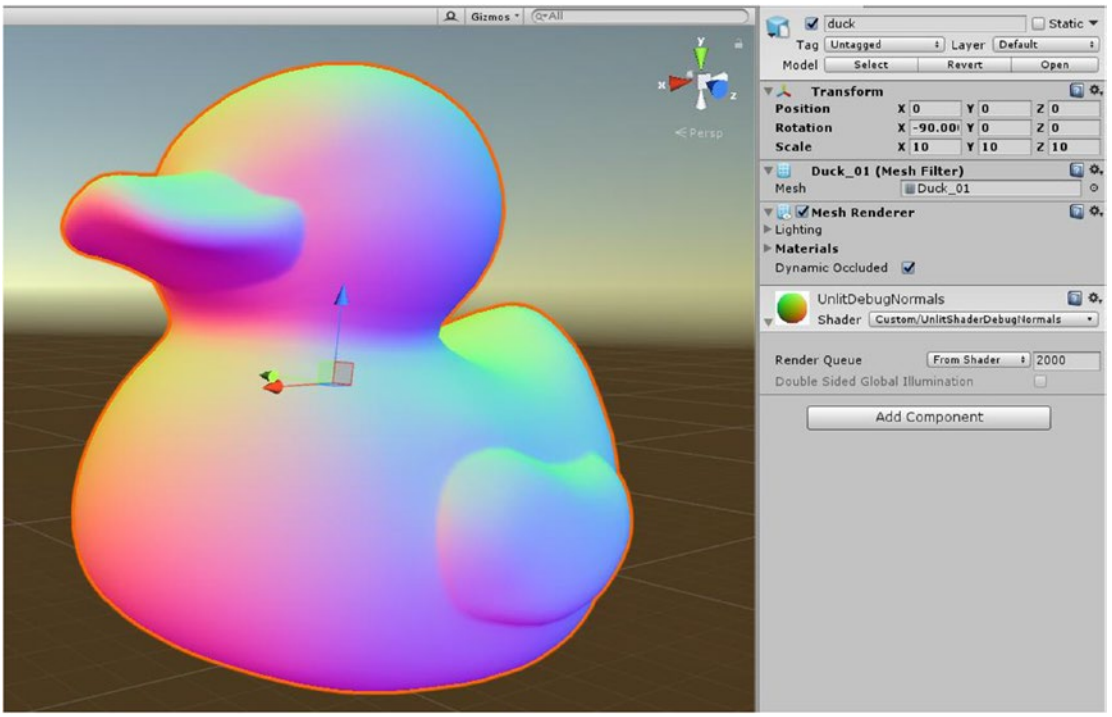

Figure 17-1. *The Normal Debugging shader applied to the duck mesh*

Imagine a multipass image effect. The error could be at any point in the passes, so you first want to narrow it down to the specific pass. You can change the script to output the intermediate passes on-screen. You could also use a Frame Debugging tool, if it's available.

Debugging Tools

You first tool you should consider using is contained in Unity—the Unity Frame Debugger (see Figure 17-2). Choose Window ➤ Frame Debugger to open it. It allows you to see each of the rendering steps and to check your meshes, shader properties, and more. You can jump to any point in the sequence of rendering steps, which can be very useful when you have a complex scene.

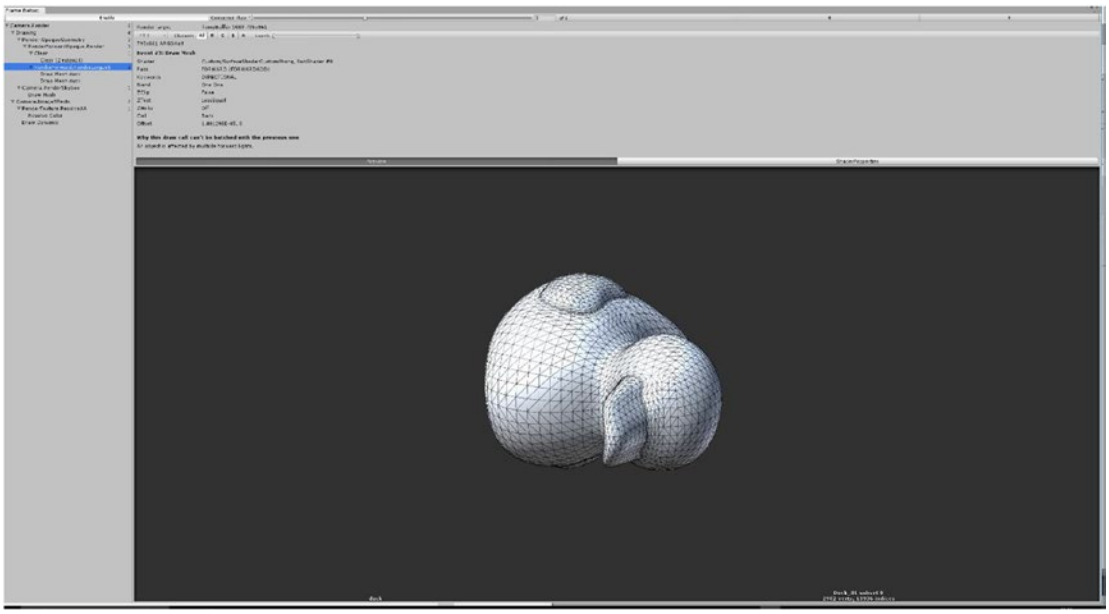

Figure 17-2. *The Unity Frame Debugger*

Sadly, the Frame Debugger is not available on every platform: at least iOS and WebGL can't use it. Under iOS you should be able to use the equivalent tool for XCode, the OpenGL ES Frame Debugger. The Frame Debugger is mainly a tool to check for correctness and to figure out what exactly your renderer is doing.

Things to watch for include: verify your shader properties, check that the mesh is the correct one, check which passes of a certain shader have been rendered, check the `RenderTexture`, verify the order in which things are rendered, and check when the screen is being cleared.

Depending on your target platform, there are many other Frame Debugging tools that you may want to use instead of the built-in one, either because it isn't available on your deployment platform, or because you want other features. *RenderDoc* is a popular PC tool that is integrated in Unity. If you want to use RenderDoc to debug your frame, right-click on the Game tab and choose Load RenderDoc. The RenderDoc icon will appear next to the frame scale slider. When you click on it, RenderDoc will open, containing a capture of your frame.

RenderDoc (see Figure 17-3) gives you access to an amazing amount of information, including the pipeline state, information about what the rasterizer is doing, texture samplers, inputs to vertex shader, frame statistics, and much more. It's well worth trying it out and exploring the possibilities.

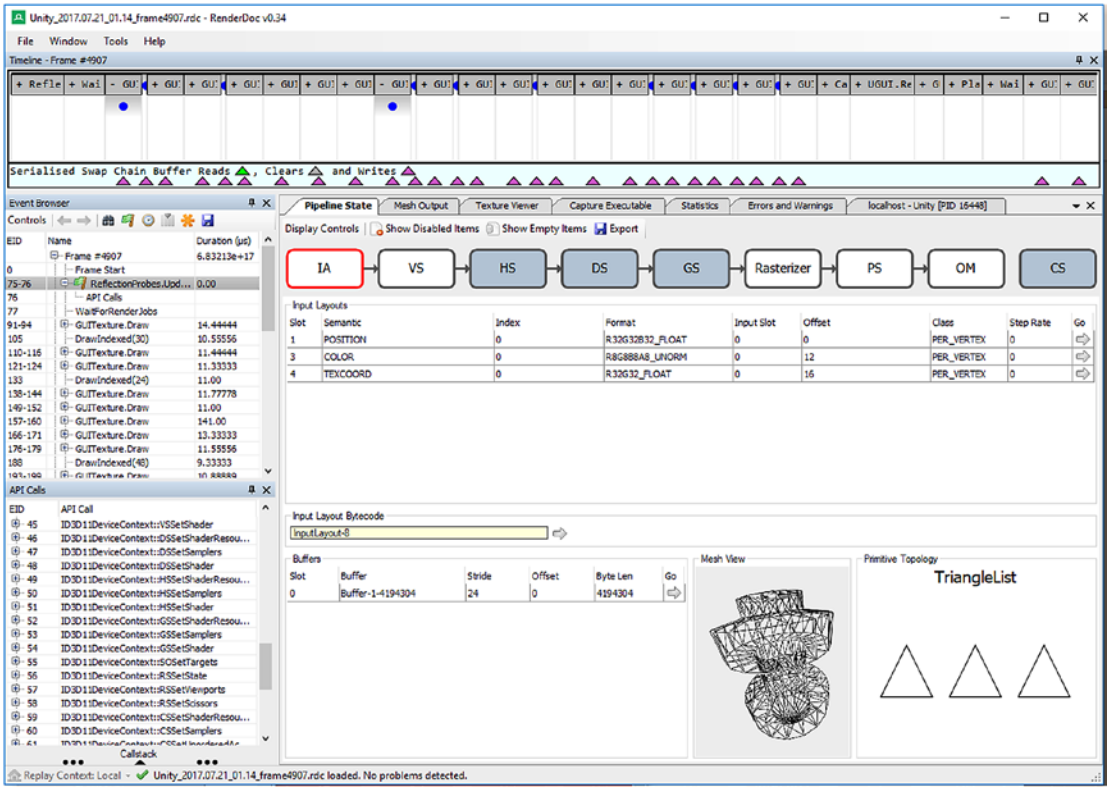

Figure 17-3. *RenderDoc*

You can also use the Visual Studio Graphics Debugger, Tegra Graphics Debugger, NVIDIA Nsight, GPU PerfStudio, and more, depending on your platform.

Looking at the Generated Shader Code

What you've written in Cg and what actually ends up running on your deployment platforms, are not necessarily the same. There are many steps your code goes through that could end up changing the meaning of your program, especially when you're targeting different platforms at the same time. When you have a problem you can't pin down, you should take a look at the generated and/or compiled code that your platform is actually running.

You can look at the final code that generated from your shader, and the `.cginc` files it includes, by clicking on your shader and looking at the inspector. Figure 17-4 shows your options. This is a Surface Shader, and you can choose to see the intermediate code generated or go straight to the compiled code for different platforms.

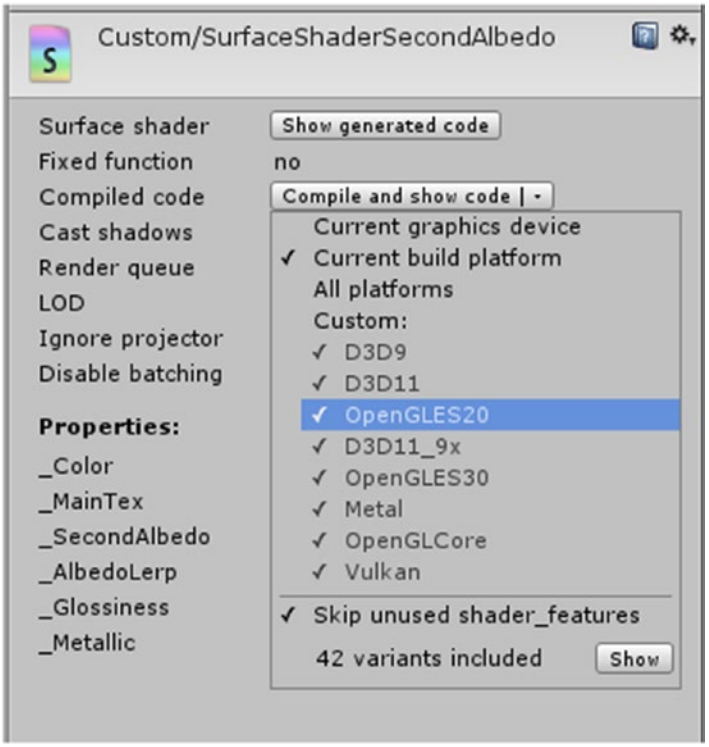

Figure 17-4. *Obtaining compiled code for your Surface Shader*

That compiled code is much lower level and it may be quite hard to understand, so that's probably something you want to try as a last measure.

Performance Profiling

Sometimes the shaders behave as we want, but they are far too slow. For that, you want to consider the general performance of the entire scene, which you can check out by using the Unity Profiler (see Figure 17-5).

Figure 17-5. *The Unity Profiler*

The Profiler can analyze performance in many subsystems of your game, but for the shaders you're mainly going to care about the Rendering and the Global Illumination parts. Let's look at some of the information you can get out of the Rendering Profiler:

- *Number of batches*: Unity attempts to combine as many objects as possible, to minimize changes in the rendering settings. In the details, batches are broken down by type (Static, Dynamic, and GPU Instancing). Fewer is better.

- *Number of drawcalls*: A drawcall is a call to the graphics API, where object data is sent to be rendered. Fewer is better.
- *Number of SetPass calls*: SetPass calls are information sent by the CPU to the GPU. Fewer is better.
- *Number of triangles/vertices*: As it says on the tin. It can help you keep an eye on spikes in on-screen primitives.
- How much VRAM is being used.
- *RenderTexture number, memory used, and number of switches*: RenderTextures are used in post processing, and you can also use them to process images on the GPU. If you have many active ones, they may eat up most of your VRAM.
- Number of shadow casters.
- Number and size of sent vertex buffer objects.

In general, you want to keep the numbers of all these items as small as possible, but how many you can afford really depends on the platform you're targeting. Some of them may have a more dramatic effect than others. For example, mobile game development used to be very drawcall-sensitive, although it's somewhat less so now.

One key issue to determine to optimize your game is whether your game is CPU- or GPU-bound, because they require different interventions. In the CPU Usage part of the Unity Profiler, all tasks executed for each frame are listed, with timings in milliseconds, and what percentage of the frame time they take up. You can drill down within `Camera.Render` until you get to `Render.Mesh`. You might want to create a test scene with just your shader, which will make it easier to figure out what it's doing, but that may also backfire because some performance problems arise only within the context of a larger scene and then disappear in a simpler scene.

If you are GPU-bound and you have determined that one of your shaders is too slow, it's time to dig out the frame-specific tools.

Keep in mind that not all shader debuggers give you useful information about the performance of your shader. In RenderDoc you can get timings per single drawcall by choosing the Time Durations for the Drawcalls option (a clock icon) within the Event Browser toolbar. In some other tools, you can only get frame percentage, which is not as useful, and still in others there are no timings at all. You need to spend some time to get to know the tools available for your use case and learn to get as much as possible out of them.

If you think you are using too many drawcalls, you should make sure you're helping Unity to batch your meshes together as much as possible. For example, set every `GameObject` that isn't supposed to move to `static`, and use the same material for as many meshes as possible, as meshes that use different materials can't be batched.

Summary

This chapter discussed many different tools that will help you profile and debug your shaders, what data you can get out of them, and some useful debugging techniques.

Next

The next chapter is about how to keep your knowledge up to date, by chasing the never-ending flow of new discoveries and inventions in the games and graphics industries.

CHAPTER 18

Keeping Up with the Industry

The game industry is in perennial and relentless progress. The insurmountable issues that you incurred last year are quite likely to have been overcome by a fellow game developer by now. But how do you keep abreast of the latest developments in graphics programming for game development?

Conferences

One way to keep up with progress is to go to game conferences, or at least watch the videos of the talks once they're published. The main ones that are relevant to graphics programmers and Unity developers are these:

- *Game Developer Conference, aka GDC*. Generally held in the United States, it's arguably the most relevant game developer conference in the world, mainly attended by AAA developers. Attending doesn't come cheap, though. You can get the videos of the talks afterward by subscribing to the GDC Vault (this is around $400/yearly).
- *Siggraph*. This conference is entirely focused on graphics innovation. Generally held in North America. Mainly attended by graphics programmers, graphics researchers, and artists, it includes an animation festival. It's not cheap to attend, but Siggraph membership is only around $45/year and it will give you access to most video recordings and papers from the conference.
- *Unite*. Unity organizes many different yearly conferences, each held on a different continent, generally in North America, Amsterdam, Japan, India, and Australia. All are devoted entirely to Unity. The cost is moderate, and there'll be at least one that's not too far from you. The videos are posted publicly online after some time.
- *Digital Dragons*. A yearly gamedev conference held in Krakow, Poland, with a good percentage of graphics programming talks, including many game postmortems. It's cheap to attend, and its videos are posted publicly online.
- *Eurographics, the equivalent of Siggraph for Europe*. It's not cheap either, and it's somewhat more geared toward research.

There are many more gamedev conferences, but they tend to not focus on graphics as much as these do. There are also other conferences that focus more of the GPU side of things, such as the GPU Technology Conference.

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2_18

Books

The bleeding edge of graphics programming techniques used in current AAA games tends to get collected into yearly books, papers, and articles. There have been various series in the past:

- *GPU Gems* (1 to 3)
- *ShaderX* (1 to 7)
- *GPU PRO* (1 to 7)
- *GPU Zen*, the current series

They are well worth the effort to buy and read, but are mainly about advanced techniques, so be warned.

Online Communities

Slack, Reddit, and discord are some of the online communities where gamedevs and graphics programmers converge. Some of them are the `/r/GraphicsProgramming/` subreddit, the `/r/Unity3D/` subreddit, as well as the Unity forums, especially the `graphics-experimental-previews` one.

Web Sites

There is an immense quantity of relevant content on the Internet. We list a few here, which are by no means exhaustive:

- `http://blog.selfshadow.com/publications/`: A lot of Siggraph material is collected here
- `https://labs.unity.com/`: the Unity Labs collect the latest research coming from Unity
- `http://filmicworlds.com/`
- `http://aras-p.info/`
- `https://seblagarde.wordpress.com/`
- `http://c0de517e.blogspot.co.uk/`
- `http://blog.tobias-franke.eu/`
- `https://bartwronski.com/`
- `http://bitsquid.blogspot.co.uk/`
- `http://casual-effects.blogspot.co.uk/`
- `http://kesen.realtimerendering.com/`
- `http://graphicscodex.com`
- `https://www.scratchapixel.com/`

Social Media

Many graphics programmers have Twitter accounts, where they post new screenshots from their research/games/tests. Here is a random sample of people worth following, in random order:

- @shadercat & @doppioslash: This is me and my shadercat account, where I post new articles and anything cool related to graphics programming that I come across
- @SebLagarde: Director of rendering research at Unity and ex-senior graphic programmer at DICE/Frostbite and Dontnod
- @zalbard: PBR programmer at Unity Labs
- @thefranke: GI graphics monkey at Unity Technologies
- @EricLengyel: Game development and mathematics author and creator of http://sluglibrary.com, http://opengex.org, @TombstoneEngine, and @The31stGame
- @kenpex: DAY: Rendering technical director. NIGHT: Creative coder and photographer. Sometimes I write on c0de517e. My opinions are NOT my own
- @sehurlburt: Graphics engineer and entrepreneur. At Binomial making Basis, a texture compressor, with @richgel999 Prev Oculus, Unity. VR/C++/graphics
- @nlguillemot: Gamedev, rendering, GL, DX, and C++. Graphics @ University of Victoria
- @Reedbeta: Real-time graphics and game programmer. Amateur violinist and physicist, sci-fi nerd, cat dad, and coffeeholic
- @baldurk: Creator of @RenderDoc—Vulkan, D3D11, D3D12, and OpenGL graphics debugger. Previously at Unity, Crytek UK
- @iquilezles: I do things, mostly graphics and math and images
- @self_shadow: Senior rendering engineer, Lucasfilm Advanced Development Group
- @eric_heitz: Research scientist at Unity Technologies
- @KostasAAA: Lead graphics programmer at @RadiantWorlds, working on @SkySaga: Infinite Isles. Ex-Blitz Games Studios, ex-Rare
- @aras_p: Code plumber and devtools engineer at Unity (http://unity3d.com), personal stuff at http://aras-p.info, ask me at http://ask.fm/aras_pr
- @FilmicWorlds: the account of John Hable, who popularized linear lighting in games, and has worked on the Uncharted series
- @MichalDrobot: Principal rendering engineer at Infinity Ward—personal stuff, private opinions
- @BartWronsk: Software engineer at Google Daydream. Ex-gamedev, worked at Sony Santa Monica, Ubisoft Montreal, and CD Projekt Red

Keep in mind that these are personal accounts. I tried to include only people who Tweet a lot about graphics programming, but no guarantees.

Conclusion

So this is it—the last chapter, the end of the book. I hope it has been a useful journey, and that you'll be able to use this information to make better games and better art. For any feedback, please give me a shout on Twitter `@doppioslash` or `@shadercat`.

Good luck with your physically based shading!

Index

C. Doppioslash, *Physically Based Shader Development for Unity 2017*,
https://doi.org/10.1007/978-1-4842-3309-2

D, E

F

G

H, I, J

K

L

M

N

O

P, Q

R

S

T

U

V, W, X, Y, Z