Unity 5.x Shaders and Effects Cookbook

Unity 5.x Shaders and Effects Cookbook is the first to bring you the secrets of creating Shaders for Unity3D—guiding you through the process of understanding vectors, how lighting is constructed with them, and how textures are used to create complex effects without heavy math.

We start with essential lighting and finish by creating stunning screen effects like those in high quality 3D and mobile games. You’ll discover techniques including normal mapping, image-based lighting, and animating your models inside a Shader. We explore the secrets behind some of the most powerful techniques such as physically based rendering. With this interesting Cookbook, what seems like a dark art today will be second nature by tomorrow.

What this book will do for you...

- Understand physically based rendering to fit the aesthetic of your game
- Enter the world of post-processing effects to make your game look visually stunning
- Add life to your materials, complementing Shader programming with interactive scripts
- Design efficient Shaders for mobile platforms without sacrificing their realism
- Use state-of-the-art techniques such as volumetric explosions and fur shading
- Build your knowledge by understanding how Shader models have evolved and how you can create your own
- Discover what goes into the structure of Shaders and why lighting works the way it does
- Master the math and algorithms behind the most used lighting models

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Master the art of Shader programming to bring life to your Unity projects
In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'Creating Your First Shader'
- A synopsis of the book’s content
- More information on *Unity 5.x Shaders and Effects Cookbook*
About the Authors

Alan Zucconi is a passionate developer, author, and motivational speaker, recognized as one of Develop's "30 under 30." His expertise has been built over the past 10 years, while he dedicated his time to academia and the gaming industry. He started his independent career to fully explore his creativity, tearing down the wall between art and gaming. Prior to that, he worked at Imperial College London, where he discovered his passion for teaching and writing. His titles include the gravity puzzle, ORBITALIS, and the upcoming time travel platformer, Still Time.

Kenneth Lammers has over 15 years of experience in the gaming industry, working as a character artist, technical artist, technical art director, and programmer. Throughout his career, he has worked on titles such as Call of Duty 3, Crackdown 2, Alan Wake, and Kinect Star Wars. He currently owns and operates Ozone Interactive along with his business partner, Noah Kaarbo. Together, they have worked with clients such as Amazon, Eline Media, IGT, and Microsoft.

Kenny has worked for Microsoft Games Studios, Activision, and Surreal, and has recently gone out on his own, operating CreativeTD and Ozone Interactive.

Kenny authored the first version of Unity Shaders and Effects Cookbook by Packt Publishing, and was very happy to be a part of the writing, updating and reviewing of this book.
Unity 5.x Shaders and Effects Cookbook is your guide to becoming familiar with the creation of shaders and post effects in Unity 5. You will start your journey at the beginning, creating the most basic shaders and learning how the shader code is structured. This foundational knowledge will arm you with the means to progress further through each chapter, learning advanced techniques such as volumetric explosions and fur shading. This edition of the book is written specifically for Unity 5 and will help you to master physically-based rendering and global illumination to get as close to photorealism as possible.

By the end of each chapter, you will have gained new skill sets that will increase the quality of your shaders and even make your shader writing process more efficient. These chapters have been tailored so that you can jump into each section and learn a specific skill from beginner to expert. For those who are new to shader writing in Unity, you can progress through each chapter, one at a time, to build on your knowledge. Either way, you will learn the techniques that make modern games look the way they do.

Once you have completed this book, you will have a set of shaders that you can use in your Unity 3D games as well as the understanding to add to them, accomplish new effects, and address performance needs. So let's get started!

What this book covers

Chapter 1, Creating Your First Shader, introduces you to the world of shader coding in Unity 4 and 5.

Chapter 2, Surface Shaders and Texture Mapping, covers the most common and useful techniques that you can implement with Surface Shaders, including how to use textures and normal maps for your models.

Chapter 3, Understanding Lighting Models, gives you an in-depth explanation of how shaders model the behavior of light. The chapter teaches you how to create custom lighting models used to simulate special effects such as toon shading.
Chapter 4, *Physically Based Rendering in Unity 5*, shows you that physically-based rendering is the standard technology used by Unity 5 to bring realism to your games. This chapter explains how to make the most out of it, mastering transparencies, reflective surfaces, and global illumination.

Chapter 5, *Vertex Functions*, teaches you how shaders can be used to alter the geometry of an object; this chapter introduces vertex modifiers and uses them to bring volumetric explosions, snow shaders, and other effects to life.

Chapter 6, *Fragment Shaders and Grab Passes*, explains how to use grab passes to make materials that emulate the deformations generated by these semi-transparent materials.

Chapter 7, *Mobile Shader Adjustment*, helps you optimize your shaders to get the most out of any devices.

Chapter 8, *Screen Effects with Unity Render Textures*, shows you how to create special effects and visuals that would otherwise be impossible to achieve.

Chapter 9, *Gameplay and Screen Effects*, tells you how post-processing effects can be used to complement your gameplay, simulating, for instance, a night vision effect.

Chapter 10, *Advanced Shading Techniques*, introduces the most advanced techniques in this book, such as fur shading and heatmap rendering.
Creating Your First Shader

This chapter will cover some of the more common diffuse techniques found in today's Game Development Shading Pipelines. In this chapter, you will learn about the following recipes:

- Creating a basic Standard Shader
- Migrating Legacy Shaders from Unity 4 to Unity 5
- Adding properties to a shader
- Using properties in a Surface Shader

Introduction

Let's imagine a cube that has been painted white uniformly. Even if the color used is the same on each face, they will all have different shades of white depending on the direction that the light is coming from and the angle that we are looking at it. This extra level of realism is achieved in 3D graphics by shaders, special programs that are mostly used to simulate how light works. A wooden cube and a metal one may share the same 3D model, but what makes them look different is the shader that they use. Recipe after recipe, this first chapter will introduce you to shader coding in Unity. If you have little to no previous experience with shaders, this chapter is what you need to understand what shaders are, how they work, and how to customize them.

By the end of this chapter, you will have learned how to build basic shaders that perform basic operations. Armed with this knowledge, you will be able to create just about any Surface Shader.
Creating a basic Standard Shader

Every Unity game developer should be familiar with the concept of components. All the objects that are part of a game contain a number of components that affect their look and behavior. While scripts determine how objects should behave, renderers decide how they should appear on the screen. Unity comes with several renderers, depending on the type of object that we are trying to visualise; every 3D model typically has MeshRenderer. An object should have only one renderer, but the renderer itself can contain several materials. Each material is a wrapper for a single shader, the final ring in the food chain of 3D graphics. The relationships between these components can be seen in the following diagram:

Understanding the difference between these components is essential to understand how shaders work.
Getting ready

To get started with this recipe, you will need to have Unity 5 running and must have created a new project. There will also be a Unity project included with this cookbook, so you can use that one as well and simply add your own custom shaders to it as you step through each recipe. With this completed, you are now ready to step into the wonderful world of real-time shading!

How to do it...

Before getting into our first shader, let's create a small scene for us to work with. This can be done by navigating to GameObject | Create Empty in the Unity editor. From here, you can create a plane to act as a ground, a couple of spheres to which we will apply our shader, and a directional light to give the scene some light. With our scene generated, we can move on to the shader writing steps:

1. In the Project tab in your Unity editor, right-click on the Assets folder and select Create | Folder.

If you are using the Unity project that came with the cookbook, you can skip to step 4.

2. Rename the folder that you created to Shaders by right-clicking on it and selecting Rename from the drop-down list or selecting the folder and hitting F2 on the keyboard.

3. Create another folder and rename it to Materials.

4. Right-click on the Shaders folder and select Create | Shader. Then right-click on the Materials folder and select Create | Material.

5. Rename both the shader and material to StandardDiffuse.

6. Launch the StandardDiffuse shader in MonoDevelop (the default script editor for Unity) by double-clicking on it. This will automatically launch the editor for you and display the shader code.

You will see that Unity has already populated our shader with some basic code. This, by default, will get you a basic Diffuse shader that accepts one texture. We will be modifying this base code so that you can learn how to quickly start developing your own custom shaders.
7. Now let's give our shader a custom folder from which it's selected. The very first line of code in the shader is the custom description that we have to give the shader so that Unity can make it available in the shader drop-down list when assigning to materials. We have renamed our path to Shader "CookbookShaders/StandardDiffuse", but you can name it to whatever you want and rename it at any time. So don't worry about any dependencies at this point. Save the shader in MonoDevelop and return to the Unity editor. Unity will automatically compile the shader when it recognizes that the file has been updated. This is what your shader should look like at this point:

```csharp
Shader "CookbookShaders/StandardDiffuse" {
    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  
    }
    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;

        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;
        }
    }
}
```
8. Technically speaking, this is a **Surface Shader** based on **physically-based rendering**, which Unity 5 has adopted as its new standard. As the name suggests, this type of shader achieves realism by simulating how light physically behaves when hitting objects. If you are using a previous version of Unity (such as Unity 4), your code will look very different. Prior to the introduction of physically-based shaders, Unity 4 used less sophisticated techniques. All these different types of shader will be further explored in the next chapters of this book.

9. After your shader is created, we need to connect it to a material. Select the material called **StandardDiffuse** that we created in step 4 and look at the **Inspector** tab. From the **Shader** drop-down list, select **CookbookShaders | StandardDiffuse**. (Your shader path might be different if you chose to use a different path name.) This will assign your shader to your material and make it ready for you to assign to an object.

   ![Shader assignment](image)

   To assign a material to an object, you can simply click and drag your material from the **Project** tab to the object in your scene. You can also drag a material onto the **Inspector** tab of an object in the Unity editor to assign a material.

   The screenshot of an example is as follows:

   ![Shader example](image)

   Not much to look at at this point, but our shader development environment is set up and we can now start to modify the shader to suit our needs.
Creating Your First Shader

How it works...

Unity has made the task of getting your shader environment up and running, which is very easy for you. It is simply a matter of a few clicks and you are good to go. There are a lot of elements working in the background with regard to the Surface Shader itself. Unity has taken the Cg shader language and made it more efficient to write by doing a lot of the heavy Cg code lifting for you. The Surface Shader language is a more component-based way of writing shaders. Tasks such as processing your own texture coordinates and transformation matrices have already been done for you, so you don't have to start from scratch any more. In the past, we would have to start a new shader and rewrite a lot of code over and over again. As you gain more experience with Surface Shaders, you will naturally want to explore more of the underlying functions of the Cg language and how Unity is processing all of the low-level graphics processing unit (GPU) tasks for you.

All the files in a Unity project are referenced independently from the folder that they are in. We can move shaders and materials from within the editor without the risk of breaking any connection. Files, however, should never be moved from outside the editor as Unity will not be able to update their references.

So, by simply changing the shader's path name to a name of our choice, we have got our basic Diffuse shader working in the Unity environment, with lights and shadows and all that by just changing one line of code!

See also

The source code of the built-in shaders is typically hidden in Unity 5. You cannot open this from the editor like you do with your own shaders.

For more information on where to find a large portion of the built-in Cg functions for Unity, go to your Unity install directory and navigate to Unity45\Editor\Data\CGIncludes. In this folder, you can find the source code of the shaders shipped with Unity. Over time, they have changed a lot. UNITY DOWNLOAD ARCHIVE (https://unity3d.com/get-unity/download/archive) is the place to go if you need to access the source codes of a shader used in a different version of Unity. After choosing the right version, select Built in shaders from the drop-down list, as shown in the following image. There are three files that are of note at this point—UnityCG.cginc, Lighting.cginc, and UnityShaderVariables.cginc. Our current shader is making use of all these files at the moment:
Chapter 10, Advanced Shading Techniques, will explore in-depth how to use GcInclude for a modular approach to shader coding.

### Migrating Legacy Shaders from Unity 4 to Unity 5

It is undeniable that graphics in videogames have changed massively over the last 10 years. Every new game comes with cutting-edge techniques that are getting us closer to achieving real-time photorealism. We should not be surprised by the fact that shaders themselves have changed massively throughout the lifetime of Unity. This is one of the major sources of confusion when approaching shaders for the first time. Prior to Unity 5, mainly two different shaders were adopted: **Diffuse** and **Specular**. As the names suggest, they were used for matte and shiny materials, respectively. If you are already using Unity 5, you can skip this recipe. This recipe will explain how to replicate these effects using Unity 5.

### Getting ready

The starting point of this recipe is having a workspace made in Unity 4, which uses some of the built-in shaders that were originally provided. If you are to start a new game, there is no doubt that you should use the latest version of Unity. However, if your project is already in the later stages of development with an older version, you should be very careful before migrating. Many things have changed behind the curtains of the engine, and even if your built-in shaders will most likely work without any problem, your scripts might not. If you are to migrate your entire workspace, the first thing that you should do is take backup. It is important to remember that saving assets and scenes is not enough as most of the configuration in Unity is stored in its metadata files. The safest option to survive a migration is to duplicate the entire folder that contains your project. The best way of doing this is by physically copying the folder from File Explorer (Windows) or Finder (Mac).
Creating Your First Shader

How to do it...

There are two main options if you want to migrate your built-in shaders: upgrading your project automatically or switching to Standard Shaders instead.

Upgrading automatically

This option is the easiest one. Unity 5 can import a project made with an earlier version and upgrade it. You should notice that once the conversion is done, you will not be able to use Unity 4; even if none of your assets may have changed directly, Unity metadata has been converted. To proceed with this, open Unity 5 and click on OPEN OTHER to select the folder of your old project. You will be asked if you want to convert it; click on Upgrade to proceed. Unity will reimport all of your assets and recompile all of your scripts. The process might last for several hours if your project is big. Once the conversion is done, your built-in shaders from Unity 4 should have been replaced with their legacy equivalent. You can check this from the inspector of your materials that should have changed (for instance) from Bumped Diffuse to Legacy Shader/Bumped Diffuse.

Even if Diffuse, Specular, and the other built-in shaders from Unity 4 are now deprecated, Unity 5 keeps them for backward compatibility. They can be found in the drop-down menu of a material under the Legacy Shaders folder.

Using Standard Shaders

Instead of using the Legacy Shaders, you might decide to replace them with the new Standard Shaders from Unity 5. Before doing this, you should keep in mind that as they are based on a different lighting model, your materials will most likely look different. Unity 4 came with more than eighty different built-in shaders divided in six different families (Normal, Transparent, Transparent Cutout, Self-Illuminated, and Reflective). In Unity 5, they are all replaced by the Standard Shader introduced in the previous recipe. Unfortunately, there is no magic recipe to convert your shaders directly. However, you can use the following table as a starting point to understand how the Standard Shader can be configured to simulate Unity 4 Legacy Shaders:

<table>
<thead>
<tr>
<th>Shader</th>
<th>Unity 4</th>
<th>Unity 4 (Legacy)</th>
<th>Unity 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuse</td>
<td>Diffuse</td>
<td>Legacy Shader/Diffuse</td>
<td>Standard</td>
</tr>
<tr>
<td></td>
<td>Lambert</td>
<td>Lambert</td>
<td>Physically-based rendering: Metallic Workflow</td>
</tr>
<tr>
<td>Specular</td>
<td>Specular</td>
<td>Legacy Shader/Specular</td>
<td>Standard (Specular setup)</td>
</tr>
<tr>
<td></td>
<td>Blinn-Phong</td>
<td>Blinn-Phong</td>
<td>Physically-based rendering: Specular Workflow</td>
</tr>
</tbody>
</table>
You can change the shader used by your old material using the **Shader** drop-down menu in **Inspector**. All you need to do is simply select the appropriate Standard Shader. If your old shader used textures, colours, and normal maps, they will be automatically used in the new Standard Shader. You might still have to configure the parameters of the Standard Shader to get as close to your original lighting model as possible. The following picture shows the ubiquitous **Stanford bunny** rendered with a Legacy Diffuse Shader (right), converted Standard Shader (left), and Standard Shader with **Smoothness** set to zero (middle):

You can change the shader used by your old material using the **Shader** drop-down menu in **Inspector**. All you need to do is simply select the appropriate Standard Shader. If your old shader used textures, colours, and normal maps, they will be automatically used in the new Standard Shader. You might still have to configure the parameters of the Standard Shader to get as close to your original lighting model as possible. The following picture shows the ubiquitous **Stanford bunny** rendered with a Legacy Diffuse Shader (right), converted Standard Shader (left), and Standard Shader with **Smoothness** set to zero (middle):

### Migrating custom shaders

If you have written custom shaders in Unity 4, chances are that this will work straightaway in Unity 5. Despite this, Unity has made some minor changes in the way shaders work, which can cause both errors and inconsistencies. The most relevant and important one is the intensity of the light. Lights in Unity 5 are twice as bright. All the Legacy Shaders have been rewritten to take this into account; if you have upgraded your shaders or switched to Standard Shaders, you will not notice any difference. If you have written your own lighting model, you will have to be sure that the intensity of the light is not multiplied by two any more. The following code is used to ensure this:

```csharp
// Unity 4
c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten * 2);
// Unity 5
c.rgb = s.Albedo * _LightColor0.rgb * (diff * atten);
```
Creating Your First Shader

If you haven't written a shader yet, don't panic: lighting models will be extensively explained in Chapter 3, *Understanding Lighting Models*.

There are several other changes in the way Unity 5 handles shaders compared to Unity 4. You can see all of them in Shaders in Unity 5.0 at http://docs.unity3d.com/Manual/UpgradeGuide5-Shaders.html.

### How it works...

Writing shaders is always a trade-off between realism and efficiency; realistic shaders require intensive computation, potentially introducing a significant lag. It's important to use only those effects that are strictly required: if a material does not need specular reflections, then there is no need to use a shader that calculates them. This has been the main reason why Unity 4 has been shipped with so many different shaders. The new Standard Shader of Unity 5 can potentially replace all of the previous shaders as it incorporates normal mapping, transparency, and reflection. However, it has been cleverly optimized so that only the effects that are really necessary are calculated. If your standard material does not have reflections, they will not be calculated.

Despite this, the Standard Shader is mainly designed for realistic materials. The Legacy Diffuse and Specular shaders, in comparison, were not really designed for realistic materials. This is the reason switching from Legacy to Standard Shaders will mostly introduce slight changes in the way your objects are rendered.

### See also

- *Chapter 3, Understanding Lighting Models*, explores in-depth how the Diffuse and Specular shaders work. Even if deprecated in Unity 5, understanding them is essential if you want to design new lighting models.
- *Chapter 4, Physically Based Rendering in Unity 5*, will show you how to unlock the potential of the Standard Shader in Unity 5.

### Adding properties to a shader

Properties of a shader are very important for the shader pipeline as they are the method that you use to let the artist or user of the shader assign textures and tweak your shader values. Properties allow you to expose GUI elements in a material's Inspector tab without you having to use a separate editor, which provides visual ways to tweak a shader.
With your shader opened in MonoDevelop, look at the block of lines 2 through 7. This is called the Properties block. Currently, it will have one property in it called _MainTex. If you look at your material that has this shader applied to it, you will notice that there is one texture GUI element in the Inspector tab. These lines of code in our shader are creating this GUI element for us.

Again, Unity has made this process very efficient in terms of coding and the amount of time it takes to iterate through changing your properties.

**Getting ready**

Let's see how this works in our current shader called StandardDiffuse, by creating our own properties and learning more about the syntax involved. For this example, we will refit the shader previously created. Instead of using a texture, it will only use its color and some other properties that we will be able to change directly from the Inspector tab. Start by duplicating the StandardDiffuse shader. You can do this by selecting it in the Inspector tab and pressing Ctrl + D. This will create a copy called StandardDiffuse2.

You can give a friendlier name to your shader in its first line. For instance, Shader "CookbookShaders/StandardDiffuse" tells Unity to call this StandardDiffuse shader and move it to a group called CookbookShaders. If you duplicate a shader using Ctrl + D, your new file will share the same name. To avoid confusion, make sure to change the first line of each new shader so that it uses a unique alias.

**How to do it...**

Once the StandardDiffuse2 shader is ready, we can start changing its properties:

1. In our Properties block of our shader, remove the current property by deleting the following code from our current shader:
   
   ```csharp
   _MainTex ("Albedo (RGB)", 2D) = "white" {} 
   ```

2. As we have removed an essential property, this shader will not compile until the other references to _MainTex are removed. Let's remove this other line:
   ```csharp
   sampler2D _MainTex; 
   ```

3. The original shader used _MainTex to color the model. Let's change this by replacing the first line of code of the surf() function with this:
   ```csharp
   fixed4 c = _Color; 
   ```
4. When you save and return to Unity, the shader will compile, and you will see that now our material's **Inspector** tab doesn't have a texture swatch anymore. To complete the refit of this shader, let's add one more property and see what happens. Enter the following code:

```csharp
_AmbientColor ("Ambient Color", Color) = (1,1,1,1)
```

5. We have added another color swatch to the material's **Inspector** tab. Now let's add one more to get a feel for other kinds of properties that we can create. Add the following code to the **Properties** block:

```csharp
_MySliderValue ("This is a Slider", Range(0,10)) = 2.5
```

6. We have now created another GUI element that allows us to visually interact with our shader. This time, we created a slider with the name **This is a Slider**, as shown in the following screenshot:

![Inspector GUI](image.png)

Properties allow you to create a visual way to tweak shaders without having to change values in the shader code itself. The next recipe will show you how these properties can actually be used to create a more interesting shader.

While properties belong to shaders, the values associated with them are stored in materials. The same shader can be safely shared between many different materials. On the other hand, changing the property of a material will affect the look of all the objects that are currently using it.
How it works...

Every Unity shader has a built-in structure that it is looking for in its code. The Properties block is one of those functions that are expected by Unity. The reason behind this is to give you, the shader programmer, a means of quickly creating GUI elements that tie directly into your shader code. These properties that you declare in the Properties block can then be used in your shader code to change values, colors, and textures. The syntax to define a property is as follows:

```
Properties
{
  [AmbientColor] ["Ambient Color", Color] = [1,1,1,1]
}
```

Let's take a look at what is going on under the hood here. When you first start writing a new property, you will need to give it a Variable Name. The variable name is going to be the name that your shader code is going to use in order to get the value from the GUI element. This saves us a lot of time because we don't have to set up this system ourselves.

The next elements of a property are the Inspector GUI Name and Type of the property, which is contained within parentheses. The Inspector GUI Name is the name that is going to appear in the material's Inspector tab when the user is interacting with and tweaking the shader. The Type is the type of data that this property is going to control. There are many types that we can define for properties inside of Unity shaders. The following table describes the types of variables that we can have in our shaders:

<table>
<thead>
<tr>
<th>Surface Shader property types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range (min, max)</td>
</tr>
<tr>
<td>Color</td>
</tr>
<tr>
<td>2D</td>
</tr>
<tr>
<td>Rect</td>
</tr>
<tr>
<td>Cube</td>
</tr>
<tr>
<td>Float</td>
</tr>
<tr>
<td>Vector</td>
</tr>
</tbody>
</table>
Finally, there is the **Default Value**. This simply sets the value of this property to the value that you place in the code. So, in the previous example image, the default value for the property named `_AmbientColor`, which is of the Color type, is set to a value of 1,1,1,1. As this is a color property expecting a color that is RGBA or float4 or r, g, b, a = x, y, z, w, this color property, when first created, is set to white.

**See also**


**Using properties in a Surface Shader**

Now that we have created some properties, let’s actually hook them up to the shader so that we can use them as tweaks to our shader and make the material process much more interactive.

We can use the properties' values from the material's **Inspector** tab because we have attached a variable name to the property itself, but in the shader code, you have to set up a couple of things before you can start calling the value by its variable name.

**How to do it...**

The following steps show you how to use the properties in a Surface Shader:

1. To begin, let's remove the following lines of code, as we deleted the property called `_MainTex` in the *Creating a basic Standard Shader* recipe of this chapter:
   
   ```
   _MainTex ("Albedo (RGB)", 2D) = "white" {}  
   sampler2D _MainTex;  
   fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
   ```

2. Next, add the following lines of code to the shader, below the `CGPROGRAM` line:
   
   ```
   float4 _AmbientColor;  
   float _MySliderValue;
   ```

3. With step 2 complete, we can now use the values from the properties in our shader. Let's do this by adding the value from the `_Color` property to the `_AmbientColor` property and giving the result of this to the `o.Albedo` line of code. So, let's add the following code to the shader in the `surf()` function:
   
   ```
   void surf (Input IN, inout SurfaceOutputStandard o) {  
   fixed4 c = pow(((_Color + _AmbientColor),  
   _MySliderValue));  
   o.Albedo = c.rgb;  
   o.Metallic = _Metallic;
   ```
4. Finally, your shader should look like the following shader code. If you save your shader in MonoDevelop and re-enter Unity, your shader will compile. If there were no errors, you will now have the ability to change the ambient and emissive colors of the material as well as increase the saturation of the final color using the slider value.

Pretty neat, huh!

Shader "CookbookShaders/StandardDiffuse3" {
    // We define Properties in the properties block
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
        _AmbientColor("Ambient Color", Color) = (1,1,1,1)
        _MySliderValue("This is a Slider", Range(0,10)) = 2.5
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        // We need to declare the properties variable type inside of the
        // CGPROGRAM so we can access its value from the properties block.
        CGPROGRAM
        #pragma surface surf Standard fullforwardshadows
        #pragma target 3.0
        struct Input {
            float2 uv_MainTex;
        }

        fixed4 _Color;
        float4 _AmbientColor;
        float _MySliderValue;

        void surf (Input IN, inout SurfaceOutputStandard o) {
            // We can then use the properties values in our shader
            fixed4 c = pow((_Color + _AmbientColor),
                            _MySliderValue);
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }
    }
}
Creating Your First Shader

```c
ENDCG
}
Fallback "Diffuse"
}
```

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

The `pow(arg1, arg2)` function is a built-in function that will perform the equivalent math function of power. So, argument 1 is the value that we want to raise to a power and argument 2 is the power that we want to raise it to.

To find out more about the `pow()` function, look at the Cg tutorial. It is a great free resource that you can use to learn more about shading and get a glossary of all the functions available to you in the Cg shading language:

http://http.developer.nvidia.com/CgTutorial/cg_tutorial_appendix_e.html

The following screenshot demonstrates the result obtained using our properties to control our material's colors and saturation from within the material's **Inspector** tab:
How it works...

When you declare a new property in the Properties block, you are providing a way for the shader to retrieve the tweaked value from the material's Inspector tab. This value is stored in the variable name portion of the property. In this case, _AmbientColor, _Color, and _MySliderValue are the variables in which we are storing the tweaked values. In order for you to be able to use the value in the SubShader{} block, you need to create three new variables with the same names as the property's variable name. This automatically sets up a link between these two so that they know they have to work with the same data. Additionally, it declares the type of data that we want to store in our subshader variables, which will come in handy when we look at optimizing shaders in a later chapter.

Once you have created the subshader variables, you can then use the values in the surf() function. In this case, we want to add the _Color and _AmbientColor variables together and take it to a power of whatever the _MySliderValue variable is equal to in the material's Inspector tab.

The vast majority of shaders start out as Standard Shaders and get modified until they match the desired look. We have now created the foundation for any Surface Shader you will create that requires a diffuse component.

Materials are assets. This means that any change made to them while your game is running in the editor are permanent. If you have changed the value of a property by mistake, you can undo it using Ctrl + Z.

There's more...

Like any other programming language, Cg does not allow mistakes. As such, your shader will not work if you have a typo in your code. When this happens, your materials are rendered in unshaded magenta:
Creating Your First Shader

When a script does not compile, Unity prevents your game from being exported or even executed. Conversely, errors in shaders do not stop your game from being executed.

If one of your shaders appears as magenta, it is time to investigate where the problem is. If you select the incriminated shader, you will see a list of errors displayed in its Inspector tab:

Despite showing the line that raised the error, it rarely means that this is the line that has to be fixed. The error message shown in the previous image is generated by deleting the `sampler2D _MainTex` variable from the `SubShader{}` block. However, the error is raised by the first line that tries to access such a variable.

Finding and fixing what's wrong with code is a process called debugging. The most common mistakes that you should check for are as follows:

- A missing bracket. If you forgot to add a curly bracket to close a section, the compiler is likely to raise errors at the end of the document, at the beginning, or a new section.
- A missing semicolon. One of the most common mistakes but luckily one of the easiest to spot and fix. Errors are often raised by the following line.
- A property that has been defined in the `Properties` section but has not been coupled with a variable in the `SubShader{}` block.
- Conversely to what you might be used to in C# scripts, floating point values in Cg do not need to the followed by an f: it’s 1.0, not 1.0f.
The error messages raised by shaders can be very misleading, especially due to their strict syntactic constraints. If you are in doubt about their meaning, it's best to search the Internet. Unity forums are filled with other developers who are likely to have encountered (and fixed) your problem before.

**See also**

More information on how to master Surface Shaders and their properties can be found in *Chapter 2, Surface Shaders and Texture Mapping*. If you are curious to see what shaders can actually do when used at their full potential, have a look at *Chapter 10, Advanced Shading Techniques*, for some of the most advanced techniques covered in this book.
Where to buy this book

You can buy Unity 5.x Shaders and Effects Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.