Introduction

Have you thought about writing cool effects in Unity? In the cutting-edge game making engine, Unity provides great options like Shader Graph, Shader Forge, Amplify Shader Editor . However, mastering how to write custom shader by hands is essential in shader creation process.

In this tutorial , you'll learn:

- What is a shader
- What is ShaderLab
- How to create a basic shader file in Unity editor.
- How to write a flat color shader.

What is a shader?

Shader is a user-defined small set of instructions runs on some stage of a graphics processor to compute or create visuals.

Shaders provide the code for certain programmable stages in rendering pipeline. The rendering pipeline defines certain sections to be programmable. Each of these sections, or stages, represents a particular type of programmable processing. Each stage has a set of inputs and outputs, which are passed from prior stages and on to subsequent stages (whether programmable or not).

In computer graphics, shader receives the input information such as: meshes, textures, materials, lights, etc. After that, GPU follows shader instructions to process input information, generating images and draws it onto screen.

The process is called **Rendering**.

Understanding Types of Shader

Technically, an individual shader doesn't output an entire image, nor do they always do rendering. Besides,different types of shader do different jobs.

The main type is **Pixel Shader** (DirectX term), also named **Fragment Shader**(GLSL term), it processes one pixel or a fragment of an image.

Pixel Shader/ Fragment Shader outputs a single pixel color, the calculates based on that pixel position on a polygon. The program runs over and over in parallel, to generate all polygons pixels. That is how the modern video cards to accelerate rendering, making the rendering fast.

Another kind of shader is so called **Vertex Shaders**, it computes the positions of vertices in the images, While the input data is 3D, computer needs to determine where the vertices appear in 2D before rendering the pixels.

Compute Shaders don't actually render anything, but are simply programs that run on video hardware. These Shaders are a recent innovation – indeed, older video hardware may not even support them.

As with Fragment and Vertex Shaders, a Compute Shader is a short program that the graphics card runs in a massively-parallel fashion. Unlike the other Shaders, Compute Shaders don't output anything visual. Instead, they take advantage of the parallel processing in video cards to do things like cryptocurrency mining. You won't work with Compute Shaders in this tutorial.

To use shader, you need assign it to a material, then video cards knows which shader to process the graphic data.

What is ShaderLab

It's a specific language to Unity.

It can be used to :

- Make it easier to embed shaders in Unity.
- Manager the shader behavior.
- Help you with compatibility management

ShaderLab acts like a wrapper. It has a structure with several sections. An instance of the `Shader` class is called a **Shader object**. A Shader object is a Unity-specific way of working with shader programs; it is a wrapper for shader programs and other information. It lets you define multiple shader programs in the same file, and tell Unity how to use them.You use Shader objects with materials to determine the appearance of your scene.

You can create Shader objects in two ways. Each has its own type of asset:

- You can write code to create a shader asset, which is a text file with the .shader extension.
- You can use Shader Graph to create a Shader Graph asset.

Whichever way you create your Shader object, Unity represents the results in the same way internally.

This is the ShaderLab Signature:

```
Shader "name"
{
  [Properties]
  Subshaders
  [Fallback]
  [CustomEditor]
}
```

Defines a shader. It will appear in the material inspector listed under `name`. Shaders optionally can define a list of properties that show up in material inspector. After this comes a list of `SubShaders` , and optionally a fallback and/or a custom editor declaration.

This example code demonstrates the basic syntax and structure of a `Shader` object. The example Shader object has a single `SubShader` that contains a single pass. It defines Material properties, a `CustomEditor`, and a `Fallback`.

```
Shader "Examples/ShaderSyntax"
{
    CustomEditor = "ExampleCustomEditor"

    Properties
    {
        // Material property declarations go here
    }
    SubShader
    {
        // The code that defines the rest of the SubShader goes here

        Pass
        {
           // The code that defines the Pass goes here
        }
    }

    Fallback "ExampleFallbackShader"
}
```

Shader Editing

You named it the **RedShader**, but at the moment it's more of a white 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 which will be displayed in Unity inspector window defined in the Properties section. As you notice, only one texture declared. Within the capabilities of the target platform, you have the charge of declared properties as your wish.

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 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

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 ... ENDCG

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;
};
```

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

The vertex shader passes information to the fragment shader, through a struct `v2f` . 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.

The 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 **FlatShader**.

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. The code shows you how to do this.

```
fixed4 frag (v2f i) : SV_Target
{
    return 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. The format complies with **RGBA(Red, Green, Blue, Alpha)** fashion. Notice Alpha is mostly going to be ignored, unless you're rendering in the Transparent queue.

```
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

To avoid Hard coding, we refactor the red color shader value into a property under **Properties** block on the top. Here we introduce a property named `_Color` like the code below:

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

A property block has a signature: **_Name ("Description", Type) = default value**. There are many different types of properties, including textures, colors, ranges, and numbers. We'll bring more of them in the future. At this moment, although the shader could compile, but the color you picked won't take any visual effect on the object surface. That's because we haven't declared and used the _Color variable yet. First add the declaration right after the `CGPROGRAM` statement:

```
fixed4 _Color;
```

Then change the return statement in the fragment function make it use the `_Color` variable:

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

Now you know how to bind a property into the shader code. Every shader code in Unity follows ShaderLab structure, ShaderLab encapsulates the HLSL program. To declare the `_Color` variable is for the ShaderLab awareness.

Final Code

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