**IGB381 – Game Engine Technology**

**Workshop 1 (Week 2) – Introduction to ShaderLab in Unity**

**Aim:**

To introduce the process and components of writing shaders in ShaderLab within Unity.

**Objectives:**

* Create a Shader file with a generic template
* Introduce properties to our shader
* Access shader properties
* Perform a simple shading technique

**Preparation:**

Ensure that Unity 5 is installed correctly on your machine and you have loaded a blank 3D project. This may require you to make a Unity account if you have not already. It is free and worth the time. Additionally, the layout of Unity used in these workshops is set to ‘2 by 3’. You can adjust the layout of Unity anytime by going to Window->Layouts.

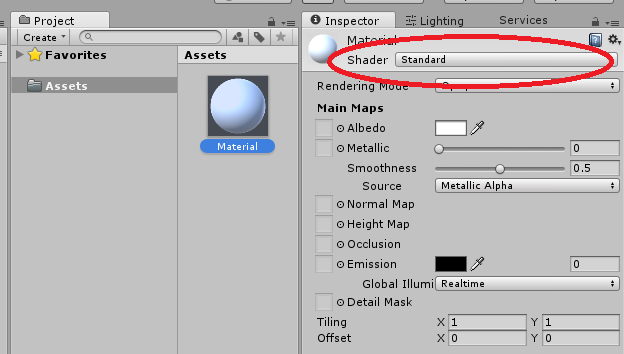
It is assumed that you are familiar with Unity, its controls and UI elements for this unit. If you have never used Unity before, please talk to your workshop tutor for a quick demonstration.

Additionally, make sure you are familiar with the current Lecture material.

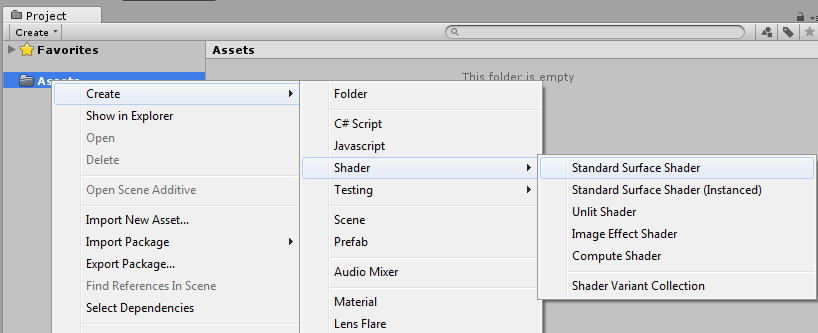
**Step 1: Creating a Shader**

To begin, first of all make a simple material. You can do this easily by right-clicking on the Assets folder in the Project Tab and going to Create->Material. Name this material ‘Material’.

When writing shaders in Unity, it is important to understand their relationship with materials. Within Unity, materials are rendered upon Game Objects using shaders. These materials can have their shader changed or even have multiple shaders applied. If you select your material in the Assets folder, in the Inspector View you will notice it has a shader automatically selected, like so:



Today we are going to make our own shader to apply to this material. Similar to how you make a C# or similar script in Unity, creating a shader can be performed by right-clicking the Assets folder and going to Create->Shader->Standard Surface Shader. Call this new shader ‘Template’.



Open the new Template shader in your editor of choice (e.g. Microsoft Visual Studio or MonoDevelop). You may find MonoDevelop typically formats shader syntax better without tweaking the editor. Inside the new shader you will find some sample shader code already set up. Remove this code and replace it with the following template code:

Shader "Template" {  
  
    Properties {  
  
    }  
  
    SubShader {  
  
        Pass {  
  
        }  
    }  
}

This is the generic structure of every shader you will write while using Unity, forming the backbone of what is called ShaderLab. You may choose to make a backup of this shader template for future use. ShaderLab provides this overarching format for shaders in Unity in order for shaders to be manipulated through the Unity interface and during runtime with scripts.

The shader header defines the name of your shader and what it will be called within Unity when selecting it. Unlike traditional scripts, the shader’s name and file name do not need to match. Additionally, if you wish to introduce a level of hierarchy to your shaders within Unity, you can use the naming convention like so:

Shader "Test/Template" {

…

}

This will store the shader within a sub-folder called ‘Test’ within Unity’s interface, allowing you to manage your shaders more appropriately.

Within the Shader declaration are, first of all, a declaration block for Properties.

Properties {  
…

}

These properties are typically variables associated with the shader that you may wish to adjust through Unity and/or scripting. These variables are declared within the Properties logic block. We will come back to exploring Properties shortly.

After Properties comes the SubShader(s).

SubShader {  
…  
}

SubShaders are where the magic will occur, performing rendering Passes with reference to Properties and adjusting how the material is rendered upon geometric surfaces. Within said passes is where we will place most of the typical CG/HLSL computer graphics language. The innards of Subshaders will be explored in more detail in the weeks to come.

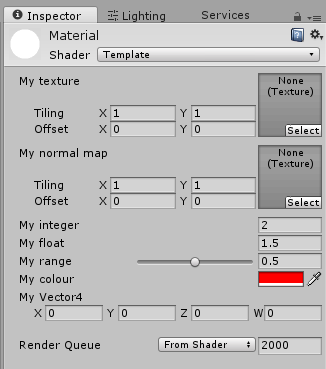
Multiple SubShaders can be written per shader, usually to accommodate older, less powerful GPUs. The first SubShader that can be run on the current GPU is the one that is selected and the others ignored. As you can imagine, writing complex shaders to support multiple GPUs can bloat shader files immensely.

**Step 2: Shader Properties**

Properties in shaders within ShaderLab are similar to public fields in C# scripts; they will appear within the inspector view on any material assigned the shader, and allow you to adjust them as you see fit. The following code will introduce you to types of properties you will use in your shaders. Place this within the Properties logic block in your shader:

Properties {  
        myTexture ("My texture", 2D) = "white" {}  
        myNormalMap ("My normal map", 2D) = "bump" {}  
        myInt ("My integer", Int) = 2  
        myFloat ("My float", Float) = 1.5  
        myRange ("My range", Range(0.0, 1.0)) = 0.5  
        myColor ("My colour", Color) = (1, 0, 0, 1)  
        myVector ("My Vector4", Vector) = (0, 0, 0, 0)  
     }

Notice that these properties are declared in a very similar fashion. The name you give them in script (e.g. myTexture) will be what you will link to with within your SubShader code. The name you give said property in quotations (e.g. “My texture”) is what you will see within Unity’s Inspector view. The property type is also important as this will determine both the default values you can give the property and what you can assign to it within Unity. Notice that ‘2D’ is considered a texture, the format for a Color is [Red, Green, Blue, Alpha] and the dimensions for a Vector are [x, y, z, w]. If you head back into Unity, you will notice your shader will look like the following in the Inspector view:



However, to access these properties within the SubShader, we will need to initialise and link to them using standard computer graphics variable declarations. To do this, add the following to you SubShader logic block, within the initial Pass:

Pass {  
            CGPROGRAM  
  
            sampler2D myTexture;  
            sampler2D myNormalMap;  
            int myInt;  
            float myFloat;  
            float myRange;  
            half4 myColor;  
            float4 myVector;

            ENDCG  
     }

This will give you access to the property of the same name when you manipulate them within the workings of a SubShader Pass. Make note of the naming conventions for the variable types and how they are not the same as what we use for property types. Additionally, note that the name of the variable is the exact same name we have given to the property.

**Step 3: Writing a simple shader**

We will finish off today by introducing you to the vertex and fragment component, writing a shader that simply applies a colour to a material. To start this, we need to first tell the compiler where our computer graphics code begins and ends. Within the body of the Pass logic block, place the following CG code underneath the variables you just declared:  
             #pragma vertex vert               
             #pragma fragment frag

This code gives instruction to Unity/ShaderLab instructing it on what to do with the various vertexes and pixels passed through the rendering pipeline. The code you write between the CGPROGRAM and ENDCG statements are in typical CG/HLSL language, including possible extensions into OpenGL. The #pragma statements indicate the name and shader types that need to be compiled for our surface, vertex or fragment shaders. The data we manipulate within the shader types are ususally set up as structures, containing the custom information necessary for our shader’s computation. As an example, place the following below the last #pragma statement:

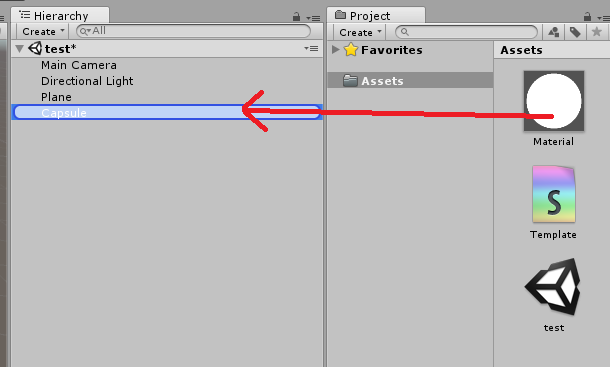
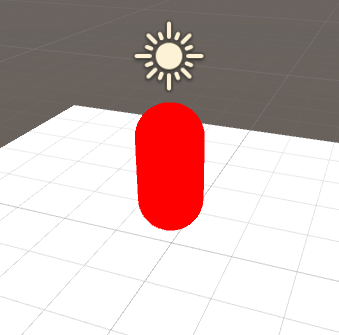
struct vertInput {  
      float4 pos : POSITION;  
     };    
   
     struct vertOutput {  
      float4 pos : SV\_POSITION;  
     };

The first struct contains the positional data to be used within the vertex shader in the form of the second struct to be used for the purposes of our fragment shader. The geometry of our model will first be passed through our vert function which will be set up as a declaration of the second struct, altering vertices if necessary. These vertices are then passed through our frag function, which will decide the colour of our eventual pixel placement. The vertex shader will process the information first and pass the result to the fragment (pixel) shader, so the order is important. Therefore, we can now write the code for the vertex and fragment shader components something like the following. Place this under the last struct:

vertOutput vert(vertInput input) {  
      vertOutput o;  
       o.pos = mul(UNITY\_MATRIX\_MVP, input.pos);  
           return o;  
     }  
   
     half4 frag(vertOutput output) : COLOR {  
           return half4(1.0, 0.0, 0.0, 1.0);   
     }

This code is where we declare what occurs in our vertex (i.e. vert) and fragment (i.e. frag) shader functions that we declared for compilation via #pragma earlier. Take note of the syntax used here and how we are using our structures to assign input parameters for our vertex and fragment shader components. This may seem confusing at first but it will be something we will review in future workshops.

All this shader currently does is use our vert and frag functions to recolour our model’s geometric material red. You can test to see if this is working by adding an object such as a capsule to your scene and applying the Material to it, like so:



This should result in the object with the material applied appearing red. As a quick test, see if you can hook up the ‘myColor’ variable instead of the static RGB setting within the code.

For your reference, the Template shader should look like the following:

