SHADERS LAB 01

HANDS-ON

In the hands-on shader unit the student will explore creating more complex shaders. These shaders will be written primarily in C for Graphics (Cg), with a little bit of Unity's built-in Shaderlab.

Section One: Introduction to Textures

In this section of the shader unit, the student will look at how to add textures to their shaders. This will allow the student to learn basic texture mapping which can be utilized in more complex tasks such as alpha blending and toon shading.

STEP ONE: SETUP

For the setup for this lab, you will need to create a new unity project (or open the unity project you have been using for in-class examples). You will then need a capsule, a material, and a shader. Keep your naming consisting, and name them something along the lines of TextureShader, TextureMaterial, TextureCapsule.

Assign the material to the capsule, and assign the shader to the material.

Go to the internet and download a simple texture (anything you find appropriate). Import this texture into unity.

Finally, open the shader in MonoDevelop or your preferred IDE.

STEP TWO: THEORY

Adding a texture map to a shader is not as complicated as you might think. There are three main parts to displaying a texture on a shader:

1. Sample the Texture
2. Unwrap the texture coordinates
3. Multiple the texture by the lighting

PREVIEWING THE CODE

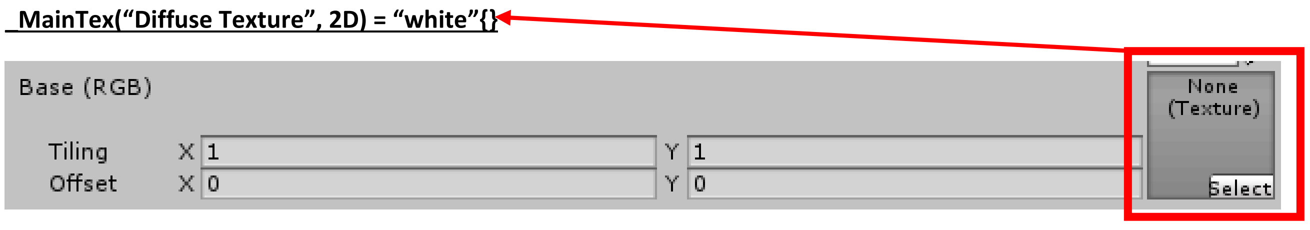
Working with textures is not incredibly difficult, but it does add a little bit more code to your shader.

There are the basic lines of code you will always need when working with a texture (keep in mind that this code will not be coloured!):

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

uniform sampler2D _MainTex;
uniform float4 _MainTex_ST;

float4 textureCoord : TEXCOORD0;
float4 texture : TEXCOORD0;

float4 texture = tex2D(_MainTex, _ MainTex_ST.xy * input.texture.xy + _MainTex_ST.zw);
```

To break these lines of code down:

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



This creates a property for unity to interface with that adds a texture input to the inspector. It is important that it is named _MainTex (specifically for **unity**) because if you wish to use some of unity's built-in components they require that this texture be named _MainTex. If you don't plan on using any of the unity built-in components you can name this whatever you want. After naming it, you assign the display name of the property that the user will see. Next you assign this to a type **2D**. 2D is a type that means "texture". So anytime you are wanting a pure texture, you use the type 2D. Finally you assign it a default value of "white" {}.This default value is a function (which is why you use the curly braces after declaring it), and what it means is to just put in a plain white texture. The other possible default values are:

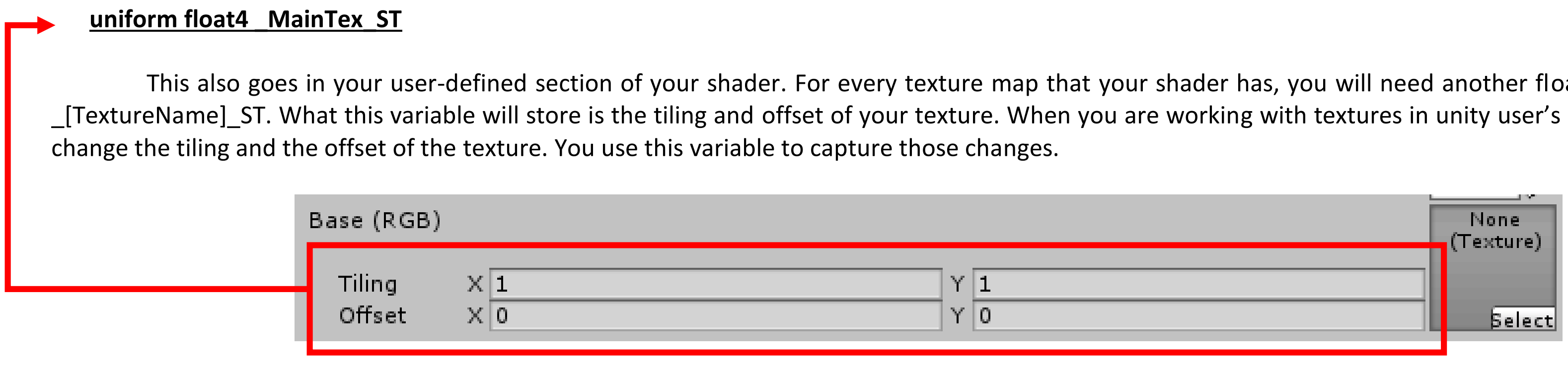
- ""
- "black"
- "gray"
- "bump"

uniform sampler2D _MainTex

This goes in your user-defined section of your shader. After declaring the property, you need to give your shader a way of accessing that property (Similar to how you declare a Color property, but define it as a float4 in the shader). sampler2D is simply the data structure that textures map to (like Colors map to float4's). The sampler2D will convert the texture into something you can actually work with by sampling the texture.

uniform float4 _MainTex_ST

This also goes in your user-defined section of your shader. For every texture map that your shader has, you will need another float4 with the name _[TextureName]_ST. What this variable will store is the tiling and offset of your texture. When you are working with textures in unity user's have the option to change the tiling and the offset of the texture. You use this variable to capture those changes.

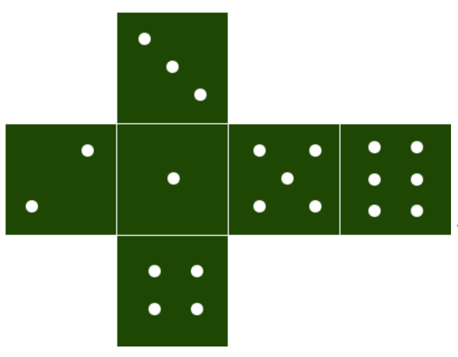
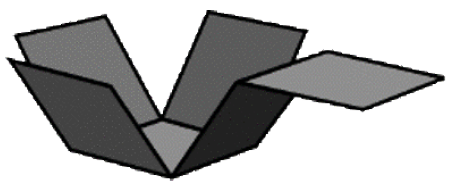
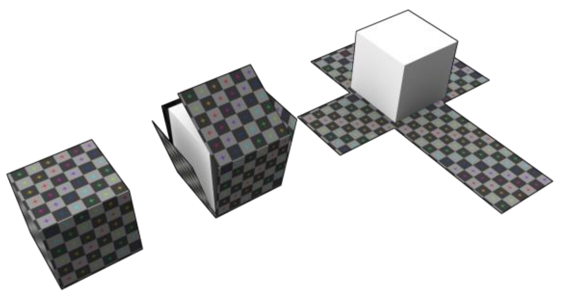


`float4 textureCoord: TEXCOORD0;`
`float4 texture: TEXCOORD0;`

With the texture sampled, you will now have to pass the uv coordinates (how the texture is wrapped around the model) from the vertex to the fragment program.

It is important to understand what a UV coordinate it. When you have a model (like a cube), you typically want to have some sort of special texture on that model (like the sides of a dice). However, the cube is a 3D object and textures are only 2D. The UV coordinate system is responsible for translating from the 2D texture to the 3D model.

So for example, the texture on the left shows a texture for a dice. On the right, you see a cube being wrapped by the texture. The middle image is like an in between state of the wrapping. The one spot on the texture will go on the bottom of the cube, the 6 on the top, and the 3, 2, and 4 on each side. If you can visualize What the UV map does is recognize that the top left corner of the one spot on the texture goes in that one specific corner on the top part of the dice, and on from there. If UV mapping still seems obscure, there are many online resources that can explain it further!

To transfer the UV maps from the vertex to the fragment program, you will need to use the TEXCOORD0 Semantic. There are two TEXCOORD's that you can utilize (TEXCOORD0 and TEXCOORD1). However, it is common practice for TEXCOORD0 to work with texture, and TEXCOORD1 to work with lightmapping.

You will be passing the TEXCOORD0 from the vertex program into the fragment programming for rendering.

float4 texture = tex2D(_MainTex, _MainTex_ST.xy * input.texture.xy + _MainTex_ST.zw);

After passing along the TEXCOORD information form the vertex to the fragment program, you will finally have all of the information that you need in the fragment program to render your texture. The tex2D function is used to unwrap the texture. It takes an input image (_MainTex) and unwraps it to the supplied UV Coordinates, in this case input.texture.xy. You then multiply the UV by the texture tiling (* _MainTex_ST.xy) , and then offset it by the offset that the user defined (+ _MainTex_ST.zw).

The texture offset is stored in the _MainTex_ST.zw, and the tiling is stored in _MainTex_ST.xy .

A word about unwrapping: When you see the word "unwrap" in relation to textures and UV coordinates what is really meant is "map". There is a bit more that goes into it, but basically unwrapping a texture means to link a pixel on the UV Map (and therefore the texture) to a position on the model.

Finally, you will multiple the texture by the color tint and light color to get the shader!

STEP THREE: CODING

With the theory covered, you can now start to write this shader. As in class, most of the theory will be covered again as you go through writing the code for this shader.

Start off by using the same basic shader setup that you have been using in class.

```
Shader "Custom/TextureShader" {
        Properties {
                _Color ("Color Tint", Color) = (1,1,1,1)
        }
        SubShader {
                Pass{

                        CGPROGRAM
                        #pragma vertex vertexFunction
                        #pragma fragment fragmentFunction

                        //user defined variables
                        uniform float4 _Color;

                        //unity defined variables

                        //input struct
                        struct inputStruct
                        {
                                float4 vertexPos : POSITION;

                        };

                        //output struct ?
                        struct outputStruct
                        {
                                float4 pixelPos: SV_POSITION;
                        };

                        //vertex program
                        outputStruct vertexFunction(inputStruct input)
                        {
                                outputStruct toReturn;

                                toReturn.pixelPos = mul(UNITY_MATRIX_MVP, input.vertexPos);
                                return toReturn;
                        }

                        //fragment program
                        float4 fragmentFunction(outputStruct input) : COLOR
                        {
                                return _Color;
                        }


                        ENDCG
                }
        }

        //Fallback
        //FallBack "Diffuse"
}
```

The next thing you will add is a way for the user to input a texture for the model to use. To do this, you will need to add a new property.

You will want to name your property _MainTex so that the texture can be used in some of unity's built-in components without having to mess with variable names.

You will set the type for this variable to "2D". This represents a 2D texture.

You will need to setup a default value for the texture, with a simple white texture being a decent choice.

```
        Properties {
                _Color ("Color", Color) = (1,1,1,1)
                _MainTex("Diffuse Texture", 2D) = "white" {}
        }
```

Next you will need to link this property into the shader by declaring the variable for it to interface with. This variable will have to be of type sampler2D (because sampler2D is what a texture is!).

You will also want to declare another variable that will store the tiling and offset that the user has adjusted for the texture. It is very important that you follow the naming scheme _[Texture Name]_ST.

```
                        //user defined variables
                        uniform float4 _Color;
                        uniform sampler2D _MainTex;
                        uniform float4 _MainTex_ST;
```

With the variables declared, you will now need to pull the UV Map off of the model so you can start to line up the texture with where it fits onto the model.

To do this, you will use the TEXCOORD0 semantic in your input struct (remember information requested from the input struct is automatically supplied if an existing semantic is used!). The TEXCOORD0 semantic represents the first UV map that the model has. Some models may have a second UV map that is typically used to store lightmapping information. If you need to access that second UV map you can do so by using TEXCOORD1.

Since you will be displaying this texture on the model in your game, you also need to notate that you are going to be outputting a texture onto the pixels.

```
                        //input struct
                        struct inputStruct
                        {
                               float4 vertexPos : POSITION;
                               float4 textureCoord : TEXCOORD0;
                        };

                        //output struct
                        struct outputStruct
                        {
                               float4 pixelPos: SV_POSITION;
                               float4 tex: TEXCOORD0;
                        };
```

Next will be actually assigning that texture information in your vertex program. You did pull the information from the vertex using your inputStruct, but you need to assign it to your outputStruct so your fragment program can access it. This is as simple as assigning the textureCoord you got from your inputStruct into the tex variable in your outputStruct.

```
                        //vertex program
                        outputStruct vertexFunction(inputStruct input)
                        {
                               outputStruct toReturn;
                               toReturn.pixelPos = mul(UNITY_MATRIX_MVP, input.vertexPos);

                               //Assigning the information that the inputStruct gathered
                               //to the struct that will be passed into the fragment
                               //program so that the fragment program can use it.
                               toReturn.tex = input.textureCoord;
                               return toReturn;
                        }
```

Finally you can apply the texture to the individual pixels inside of the fragment program. In order to do this, you must fully unwrap the texture.

To unwrap a texture, you will use the tex2D function. Remember that when using this function you are looking purely at a single pixel and a single coordinate on the UV map that that single pixel correlates to.

So, the tex2D function first requires a texture to unwrap, this is easy since your user already supplied this texture (_MainTex).

Then it needs to know what coordinate on the map to assign to the pixel. This is where you need to start thinking about things.

You have a texture coordinate supplied to you already from the input into the fragment function (input.tex). However, this is declared as a float4. A texture coordinate exists soley on a 2d coordinate, so it will not be very useful to give the tex2D function the entire input.tex. Instead of giving tex2D the x, y, z, and w of the texture it would make more sense just to give it the x and the y (what is the z and w axis on a 2d coordinate system anyway?!).

```
          //fragment program
          float4 fragmentFunction(outputStruct input) : COLOR
          {
                 float4 tex = tex2D(_MainTex, input.tex.xy);
```

Note that tex2D RETURNS a float4 (corresponding to rgba) even though it only wants a float2 (representing a coordinate x and y) for it's second parameter!

Next you can return the color of the pixel. Currently you are just returning _Color. However, you now want to return the color of the pixel on the texture that corresponds to the pixel that the fragment function is looking at.

Luckily, you already calculated this! If you were to return float(tex.rgb, 1.0) then you would see that you are returning the color on the UV map that corresponds to the specific pixel that the fragment function is looking at.

However, this isn't exactly what we want. There are two problems with this:

1. If the user changes the colour, it isn't reflected on the texture
2. If the user changes the offset or tiling, it isn't reflected on the texture

Both of these problems are very simple to solve. To reflect the offset and tiling, you just need to adjust the coordinate that you are linking the specific pixel to on the UV map.

The tilling of the texture can also be referred to as the SCALE of the texture. If the user puts in smaller numbers (like 0.5), then the texture becomes larger (because it is showing 50% of the image in the same amount of space that it would normally show 100% of the image). You access the tiling of a texture by using the _[Texture Name]_ST.xy components.

If you are familiar at all with scaling numbers, then it should be very apparent how to proceed with scaling the texture. Remember, you are looking to adjust the coordinate that the pixel is mapped to on the UV map.

You are not familiar with scaling numbers, then what you want to do is multiply the actual coordinate of the UV map by the amount of scaling.

```
   _MainTex_ST.xy * input.tex.xy
```

The next thing is to allow the offset to take affect. When a user puts in an offset, what they are wanting to do is instead of starting in the actual position on the UV map, they want to start a number of pixels to the left/right or up/down. This means if pixel # 155 is usually mapped to the UV coordinate of 0,0 and the user puts in an offset of -5 on the X, then pixel #155 will now be mapped to the UV coordinate of -5, 0 and pixel #156 will be mapped to the UV coordinate of -4, 0 and so on and so forth. You can access the offset by using the _[Texture Name]_ST.zw components.

Again, this is just adjusting where the UV mapping is!

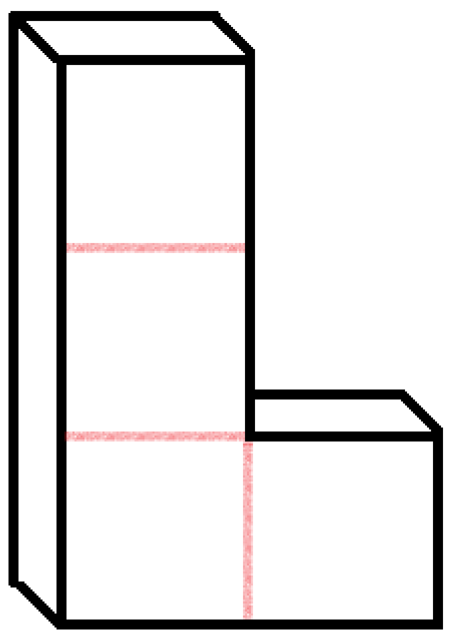
```
   float4 tex = tex2D(_MainTex, _MainTex_ST.xy * input.tex.xy + _MainTex_ST.zw);
```

WRAP UP

You now have the ability to apply a basic texture map onto a model and adjust certain aspects about how the texture is rendered.

THINK ABOUT IT!

1. Draw what a UV map might look like for the following object (Note the red lines are just for scale reference and not part of the object)



   a.
2. What meaning does a UV have in relation to a texture and/or a model?
3. What does it mean when you say that you will sample a texture?
4. What does it mean when you say that you are unwrapping a texture?
5. If I had a sampler2D named _RawrTex, what would I name the next variable to capture the tiling and offset of that sampler2D?

ON YOUR OWN!

1. Add the color tint to your shader so that the texture is affected by the color that the user supplies.
2. Flip the tiling and the offset, so that the offset works like tiling should and tiling works like offset should.
3. Add lambert lighting to this shader (Two options)
   a. Make the lambert lighting effect the actual color of each pixel on the texture
   b. Make the lambert lighting layer on top of the texture shader