LEARN THE FUNDAMENTALS OF SHADER WRITING IN UNITY IN THIS COMPLETE TUTORIAL SERIES.

In this Unity tutorial series we will be giving you a basic introduction to shader scripting, showing you how to write your own vertex/fragment shaders from scratch in unity.
I have learnt the hard way how to write shaders through books and experimentation, trial and error. The goal of this series is to take everything I have learned and structure it into easy to follow, ordered lessons that work in unity 4, with clear instructions on not only how to write clean and efficient code, but also how to debug and optimize your shaders. Effectively this is the series I wish I had when I started learning CG.

In this beginner section we will be covering how to deal with colors and textures, how to write complete lighting models from scratch and even delve into normal mapping.

This text tutorial is accompanied by a complete video. I recommend watching the video and referring to the text as reference.

Be sure to read the tutorial carefully, and post any questions you may have if you get stuck, as there is a lot to cover. I have referenced some material from the following books throughout this series:
http://en.wikibooks.org/wiki/Cg_Programming/Unity
http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html
http://http.developer.nvidia.com/GPUGems/gpugems_pref02.html
http://http.developer.nvidia.com/GPUGems2/gpugems2_frontmatter.html
http://http.developer.nvidia.com/GPUGems3/gpugems3_pref02.html

I certainly recommend reading those books as they are fantastic and cover a lot more than what you will find here, what I have done is broken everything down into easy to follow, ordered lessons that will work in unity 4.

NOTE: This tutorial requires very basic knowledge of scripting such as variables and functions. If you have no scripting experience I recommend you first watch our Introduction to Scripting tutorial in the getting started section.

PART 2 - LAMBERT SHADING

So we have our nice shader set up, but all we can do is control the color which is rather boring. Let's take a look at adding in some lambertian shading to our shader. We introduce a few new elements here such as working with lights and normals.

HOW DOES IT WORK?

The way lighting works in shaders is we need to get the direction of the light, and add the lights color to the object relative to the objects normals. This is for a single directional light, we will discuss point lights and multiple lights later on.

So how do we get this light direction?
There is a built in float4 variable in cg called _WorldSpaceLightPos0, this contains the rotation of the directional light in the first three values, while the last variable is 0. For point lights the first 3 contain the position in world space while the last variable is 1. More on that later but for now we are focusing on directional lights.

So how to we get the normal direction?
We can get the world space normal direction by assigning it the semantic NORMAL, but we can't use this yet, we need to multiply it by a built in matrix _World2Object to get it in object space which is then usable for lighting.

So how do we create lighting from the light and normal direction?
This is where the dot product comes into play, the dot product calculates the angle of the normal relative to the light direction, the dot is 1 if the normal direction is facing towards the light, -1 if it is facing away, and 0 if it is facing perpendicular. The actual mathmatical formula of a dot product(wikipedia) is: the dot product of vectors [a, b, c] and [d, e, f] is ad + be + cf.

If you want to learn more about how the dot product works there is a lot of information on wikipedia.

THE SHADER CODE

```
Shader “unityCookie/introduction/2 - Lambert - vertex” {
   Properties {
      _Color (“Color”, Color) = (1.0,1.0,1.0,1.0)
   }
   SubShader {
      Pass {
         Tags { “LightMode” = “ForwardBase” }
         CGPROGRAM
         #pragma vertex vert
         #pragma fragment frag

         //user defined variables
         uniform float4 _Color;

         //Unity defined Variables
         uniform float4 _LightColor0;

         //Base Input Structs
         struct vertexInput{
            float4 vertex : POSITION;
            float3 normal : NORMAL;
         };
         struct vertexOutput {
            float4 pos : SV_POSITION;
            float4 col : COLOR;
         };

         //vertex function
         vertexOutput vert(vertexInput v){

            vertexOutput o;

            float3 normalDirection =  normalize( float3( mul( float4(v.normal, 0.0), _World2Object).rgb ));
            float3 lightDirection;
            float atten = 1.0;

            lightDirection = normalize(float3(_WorldSpaceLightPos0.rgb));

            float3 diffuseReflection = atten * float3(_LightColor0.rgb) * float3(_Color.rgb) * max(0.0,
dot(normalDirection, lightDirection));

            o.col = float4(diffuseReflection, 1.0);
            o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
            return o;
         }

         //fragment function
            float4 frag(vertexOutput i) : COLOR
         {
            return i.col;
         }
         ENDCG
      }
   }
   //fallback commented out during development
   //Fallback “Diffuse”
}
```

Ok so there is some new stuff in here, let’s do a breakdown.

BREAKDOWN

Tags { "LightMode" = "ForwardBase" }
For lighting to even begin to work in unity 4 after revision 4.0.0f7 we need to specify our light mode. We will talk more about this in part 5 when we discuss multiple lights and point lights.

uniform float4 _LightColor0;
A couple of things, the uniform keyword is not necessary in unity but is in other software, it defines this as an initial value and that the variable is coming from outside of the CGPROGRAM.
Next the _LightColor0 gets the color of the light source.

float3 normal : NORMAL;
We want to get the NORMAL semantic to use in our vertex program, fairly straightforward.

float3 normalDirection = normalize(float3(mul(float4(v.normal, 0.0), _World2Object).rgb));
Ok so this one looks a little more complicated than it is, let's rebuild this line:
We want to multiply the vertex normal by the _World2Object matrix but for this we need the normal to be a float4 as the _World2Object is a float4x4 matrix and these need to match for the mul function to work, to do this we create a float4 with v.normal for the first 3 values and 0.0 for the last value as it is merely a placeholder for the mul function to work.
mul(float4(v.normal, 0.0), _World2Object)
Now we need to make this a float3 again, we can do this in the same way as we made it a float4 by creating a float3 and specifying which values to take, in this case I use rgb, but xyz would also work. rgba and xyzw correspond to the components of the float4, you can use either but cannot mix and match:
float3(mul(float4(v.normal, 0.0), _World2Object).rgb)
Now the value here is not limited, it could be 1.8 or 146.3 so we need to normalize it, normalizing a value squishes it to fit between 0 and 1.
normalize(float3(mul(float4(v.normal, 0.0), _World2Object).rgb));
This makes a lot more sense now doesn't it.

float atten = 1.0;
We define our attenuation as 1 for a directional light, this will become important later when we start dealing with point lights and light falloff.

lightDirection = normalize(float3(_WorldSpaceLightPos0.rgb));
We now need to get our light direction which is as easy as taking the first 3 components of _WorldSpaceLightPos0 and normalizing them.

float3 diffuseReflection = atten * float3(_LightColor0.rgb) * float3(_Color.rgb) * max(0.0, dot(normalDirection, lightDirection));
Now we get into the cool stuff and can actually calculate the lighting.
Once again this looks more complicated than it actually is, atten is just 1 here so can be ignored for now. Let's rebuild this:
dot(normalDirection, lightDirection);
We take the dot product of our normal and light directions which will return 1 when the vertex is facing the light, o when it is facing perpendicular and -1 when it is facing away. We can use this to fade the light as it faces further away, but we need to remove the negative values:
max(0.0, dot(normalDirection, lightDirection));
The max() function will return the maximum of the two values, in this case setting a value to 0.0 will remove all negative values. Now let's add some color:
*float3(_LightColor0.rgb) * float3(_Color.rgb)*
We start with our flat color as defined in the properties, we only want a float3 so we must define this as such.
and we multiply this with our light color, we can then multiply our color by our lighting to get the final look.
*float3(_LightColor0.rgb) * float3(_Color.rgb) * max(0.0, dot(normalDirection, lightDirection));*

o.col = float4(diffuseReflection, 1.0);
Now we need to return a color, and it has to include an alpha channel so we create a float4 and assign 1 to the alpha

channel. We will deal with transparency in the intermediate series.

return i.col;
With all the lighting done in the vertex shader and passed to the fragment shader via the vertexOutput struct we need to return the final color. we do this by merely returning the input.

Great! now we have some awesome Lambertian lighting, but it's quite dark on the side opposite the light, this is where we can add in some ambient lighting.

WHAT IS AMBIENT LIGHTING?

Ambient lighting is effectively a way we can increase light to our entire scene at once, the ambient lighting is defined under Edit->Render settings and can give a color tint to every object in the scene. Ambient light is caused by having light bouncing from all angles and therefore having no defined ligh source.

IMPLEMENTATION

We can use the built in ambient light in our code using UNITY_LIGHTMODEL_AMBIENT which is a float4.
I like to split my lighting up into different variables before adding it for readibilty.

```
Shader "unityCookie/introduction/2b - Ambient - vertex" {
   Properties {
      _Color ("Color", Color) = (1.0,1.0,1.0,1.0)
   }
   SubShader {
      Pass {
         Tags { "LightMode" = "ForwardBase" }
         CGPROGRAM
         #pragma vertex vert
         #pragma fragment frag

         //user defined variables
         uniform float4 _Color;

         //Unity defined Variables
         uniform float4 _LightColor0;

         //Base Input Structs
         struct vertexInput{
            float4 vertex : POSITION;
            float3 normal : NORMAL;
         };
         struct vertexOutput {
            float4 pos : SV_POSITION;
            float4 col : COLOR;
         };

         //vertex function
         vertexOutput vert(vertexInput v){

            vertexOutput o;

            float3 normalDirection =  normalize( float3( mul( float4(v.normal, 0.0), _World2Object).rgb ));
            float3 lightDirection;
            float atten = 1.0; //attenuation = 1 for directional lights

            //ambient Light
            float3 ambientLight = UNITY_LIGHTMODEL_AMBIENT.rgb;
            lightDirection = normalize(float3(_WorldSpaceLightPos0.rgb));

            float3 diffuseReflection = atten * float3(_LightColor0.rgb) * max(0.0, dot(normalDirection,
lightDirction));

            float3 lightFinal = ambientLight + diffuseReflection;

            o.col = float4(lightFinal * float3(_Color.rgb), 1.0);
            o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
            return o;
        }

        //fragment function
        float4 frag(vertexOutput i) : COLOR
        {
            return i.col;
        }
        ENDCG
    }
  }
  //fallback commented out during development
  //Fallback “Diffuse”
}
```

So there we have it, some nice Lambertian shading with a nice ambient co-efficient to it. That’s it for this part, in the next part we will take a look at specular highlights.