**IGB381 – Game Engine Technology**

**Workshop 4 (Week 5) – Lighting an Object**

**Aim:**

To write a simple pixel shader that lights an object based on a point light source. This workshop should help establish a better understanding of the work required and steps necessary for completing Assignment 1, including areas to expand and research into.

**Objectives:**

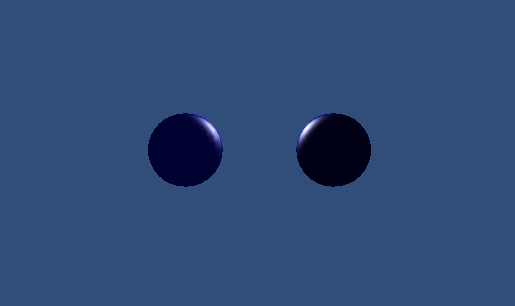
* Develop your understanding of writing a basic shader for lighting purposes
* Understand how to link and access tangible light sources inside Unity scenes within your your scripts
* Suggest ways to implement multiple light sources and types within a shader

**Preparation:**

Please make sure you have completed the workshop tutorials prior to this one as they contain materials and scripts designed for the first Assignment. Also, review the last few Lectures regarding Cg/HLSL code. In addition, it would be beneficial to have read over the Assignment brief and be aware of the marking criteria used.

**Step 1: Diffuse Lighting**

Last week you set up the scene for Assignment 1, including some basic geometry and various light sources. This week, we will explore a simple lighting technique, using a basic point light. This technique, as well as the techniques found in the Phong and Gourad shaders from last week’s SimpleLighting scene should be considered in your assignment.



As you may have noticed, the Phong and Gourad shaders from SimpleLighting do not actually rely on a tangible light source and its properties within the scene. The lightsource is actually artificial, and set up within the scripts attached to the objects. The scene we will set up today will refer to actual lights in your scene, and should act as a prototype for implementing lighting in Assignment 1.

Make a new scene in Unity and put a simple game object within it (e.g. sphere or cube). Set the light source already provided by default to a Point Light. Make a new material and assign a new shader to it, using a familiar setup for achieving a simple fixed colour.

Shader "Lighting" {

Properties {

\_Colour("Main Colour", color) = (1,1,1,1)

}

SubShader {

Pass {

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

half4 \_Colour;

struct vertInput {

float4 pos : POSITION;

};

struct vertOutput{

float4 pos : SV\_POSITION;

};

vertOutput vert(vertInput input){

vertOutput o;

o.pos = mul(UNITY\_MATRIX\_MVP, input.pos);

return o;

}

half4 frag(vertOutput output) : COLOR {

return \_Colour;

}

ENDCG

}

}

}

This shader code is similar to what you used in Week 2, just with some of the unnecessary elements removed and renamed.

Let’s walk through turning this simple fixed colour shader into a lighting based one. To do this, we first need to add a Tag to our initial pass, setting the LightMode to ForwardAdd. This will allow certain ShaderLab keywords to understand what light sources we are dealing with in our scene. Add the following Tag block to your shader just within the initial Pass declaration and before the CGPROGRAM line:

Tags{ "LightMode" = "ForwardAdd" }

We will also introduce an #include statement that we have employed before, but not explained. Place the following underneath the #pragma statements in your shader:

#include "UnityCG.cginc"

This statement will include a variety of helper functions and data structures in our shader for use, including information regarding existing light sources within our scene.

To implement lighting, we must first understand some information about the normals of the surfaces we are rendering lighting to. To do this, we need to add some additional variables that can keep track of the surface normal and colour intensity of the normal we are rendering. Add the following float4 to your script, underneath the \_Color declaration:

float4 \_LightColor0;

As well as adjusting the first struct, vertInput, to also have the following line of code:

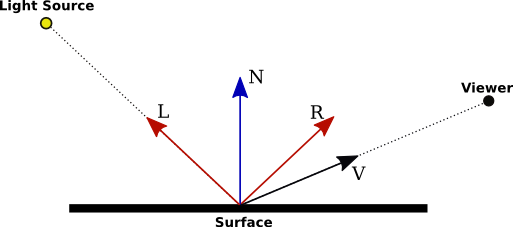
float3 nor : NORMAL;

And finally add the following to the second struct, vertOutput:

half4 col : COLOR;

Now when we jump into our vertex and fragment shaders, we have some additional information to use. As described in the Lecture, if a surface normal is facing away from a light source, the amount of light it should receive is minimal, if any.

What we are going to do now is a process similar to what was described in the Lecture. To add diffuse lighting, we need to first calculate the dot product between the surface normal and the light direction. If the directions are facing the same direction, the surface would be completely unlit. If the directions are facing each other, the surface would receive the maximum amount of light (disregarding attenuation based on distance, for the moment). Anything in between would receive a degree of lighting based on the angle the viewer is observing the surface. This process is illustrated better below:



To set this up, we shall first set up a series of vectors to manipulate, first transforming the surface normal to world space. Add this code to your vertex shader, underneath the first line ‘vertOutput o’:

float4 normal = float4(input.nor, 0.0);

float3 n = normalize(mul(normal, unity\_WorldToObject));

We now also need to take into consideration the direction of the light source. This can be done in a variety of ways (**which you should research for your assignment!**) but because we have set the LightMode to ForwardAdd, we can access our point light by talking to \_WorldSpaceLightPos0. Add this immediately after the line you just wrote:

float3 l = normalize(\_WorldSpaceLightPos0);

We now have all the information we need to calculate the dot product of our surface and light vectors. This can be done by including the following:

float3 NdotL = max(0.0, dot(n, l));

Now we have a lighting value. We can now use this value in conjunction with the object’s colour and the colour of the light we are using. By default, declaring \_LightColor0 before means we will automatically access the colour of the first/closest light source to our object. We can also access our Object’s colour through the use of the property \_Color we set up previously, which we change manually. Let’s blend these two colours together, using our dot product as a coefficient. Add the following lines of code to calculate and adjust the colour of our surface:

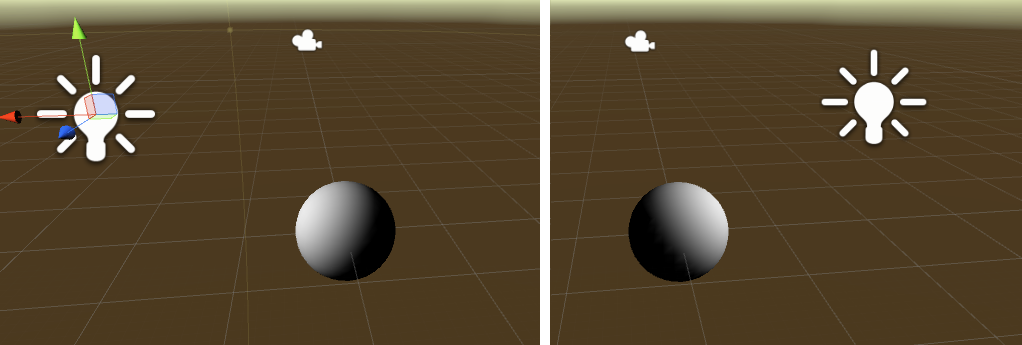
float3 d = NdotL \* \_LightColor0 \* \_Color;

o.col = float4(d, 1.0);

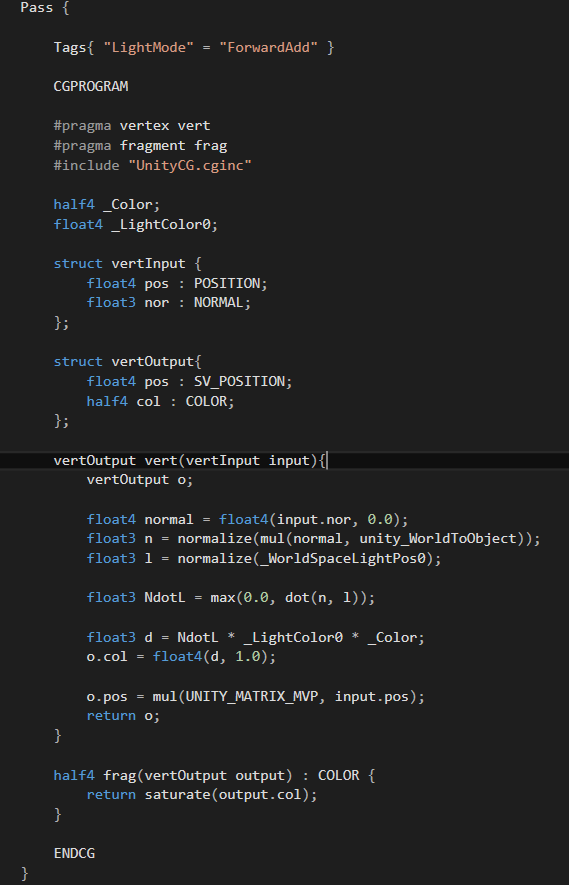
Any other aspects of the object (e.g. texture?) should ideally be considered here too. Finally, change return statement in the fragment shader ‘frag’ to instead return the following, instead of \_Color:

return saturate(output.col);

Your object should now light up appropriate, and change its diffuse lighting colours based on the position of the point light source in your scene, like the following.



You should try tweaking the colour of your object as well as the light source to see how their properties interact. For your reference, your shader’s Pass should also look like the following at this point:



**Step 2: Ambient Lighting**

The problem with diffuse lighting is that we do not take into consideration some of the ambient lighting available within our scenes, resulting in object surfaces that are completely black when facing away from light sources. To address this, we can add a couple more lines to our vertex shader to consider Unity’s ambient light value from the Render settings and multiply it with the shader’s colour value. This can be achieved by first adding a new float3 to perform this calculation, just after the dot product NdotL gets calculated:

float3 a = UNITY\_LIGHTMODEL\_AMBIENT \* \_Color;

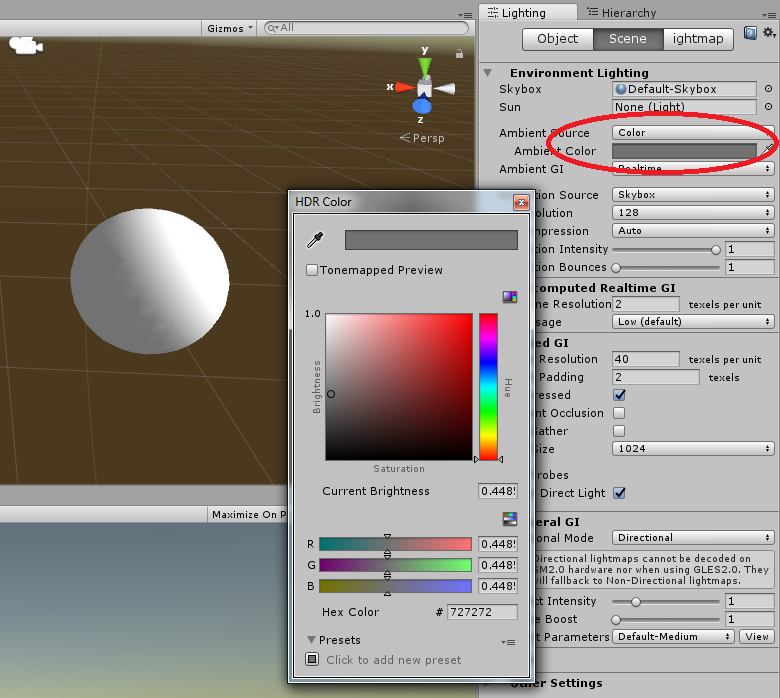
Now, just after when we blend the colours of the surface together, we can add this new float to the blend. Add this line just after you calculate the colour blend ‘d’:

float4 c = float4(d + a, 1.0);

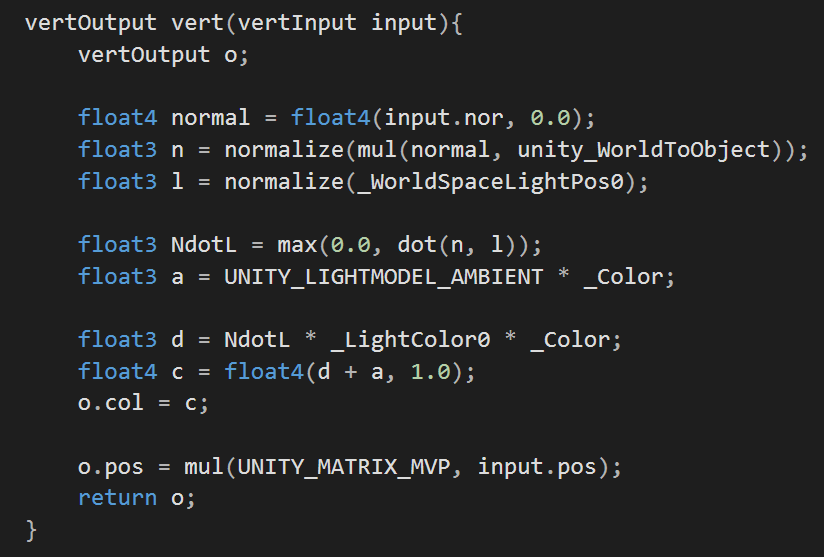
Finally, replace our original colour setting of ‘o’ to now be the following instead:

o.col = c;

This will result in our object gaining an additional amount of ambient light which you can control in the Render settings within the Lighting tab (Window->Lighting).



For reference, your vertex shader should now look like the following:



**Step 3: Specular Highlights**

Specular highlights in the real world are actually just normal reflections, but these reflections are usually quite expensive to calculate. It is generally easier to simulate them with specular highlights, essentially showcasing brighter parts of a model with reference to a reflection vector. To start, let’s first add an additional two properties to our shader. Add the following to the Properties block at the start of our script:

\_SpecColor("Specular Color", Color) = (1,1,1,1)

\_Shininess("Shininess", Float) = 10

Follow this by adding access to these properties within your Cg code by adding this after the \_Color and \_LightColor0 declarations:

float \_Shininess;

float4 \_SpecColor;

Specular highlighting is much like basic diffuse shading, just with a strong highlighted component. Highlights are typically drawn in places where the reflection vector between the light direction and normal vector is pointing towards the view direction. We can do this by first adding a new vector and normalizing the camera position. We will also create a reflection vector based on the direction of the light and the normal of the surface. Add the following code below the line “float3 d = NdotL \* \_LightColor0 \* \_Color;” within the vertex shader.

float3 v = normalize(\_WorldSpaceCameraPos);

float3 r = reflect(-l, n);

With this reflection vector, we now know where the light rays will bounce off of the surfaces, so we want to draw highlights on these areas. To do this, we need to calculate the dot product between the reflection vector and the view direction to determine if the reflection vector is pointing towards the camera. Add this immediately after the last line you inserted:

float RdotV = max(0.0, dot(r, v));

If the dot product is 1, the vector is pointing directly towards the camera. If it is less than 1, the vector isn’t pointing towards it at all. This will be a standard conditional statement within our shader where we can employ the use of our shininess variable to control how much the highlight will be spread. For the sake of demonstrating this clearly, we can employ the use of an expression, raising the power of the shininess considerably:

float3 s = float3(0, 0, 0);

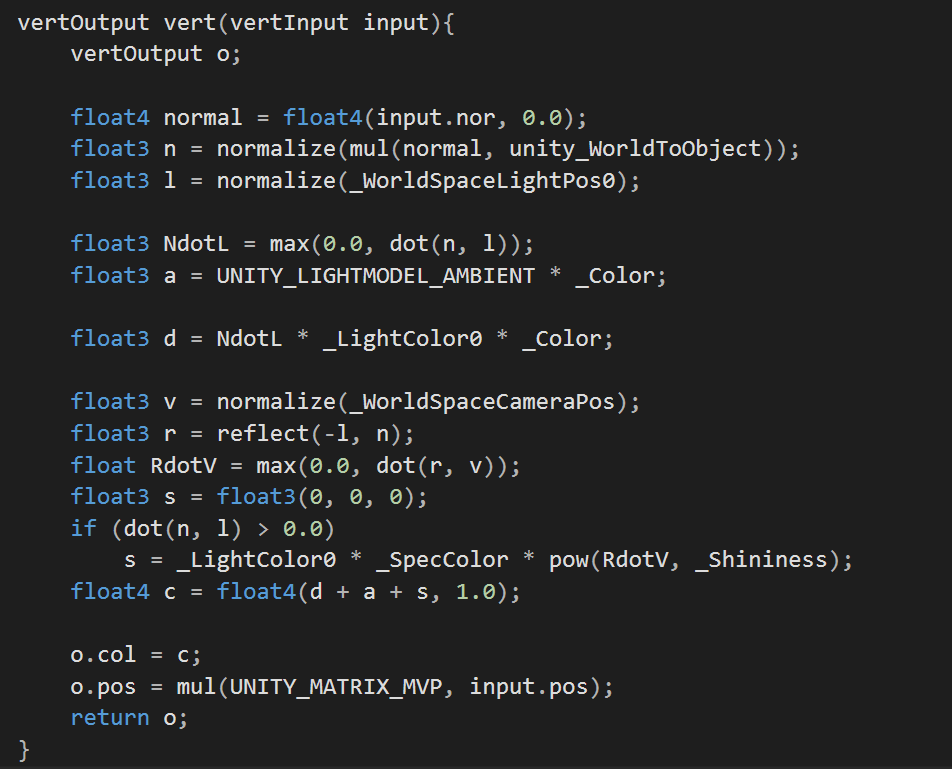
if (dot(n, l) > 0.0)

s = \_LightColor0 \* \_SpecColor \* pow(RdotV, \_Shininess);

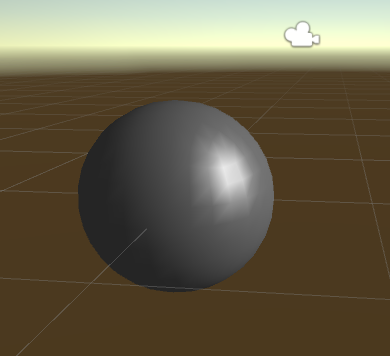
Finally, we will now need to change the line “float4 c = float4(d + a, 1.0);” to include this new shininess component ‘s’. This can be easily done by adding it to the combined values, like so:

float4 c = float4(d + a + s, 1.0);

When you have finished, your vertex shader should now look like the following, as reference:



It should also give you these kind of visual results, providing you use a dark enough base colour on your object.



Congratulations! You have actually just programmed the Phong lighting model from scratch! You should use this lighting model in your assignment when dealing with the orbiting sun (point light). Do consider additional aspects of this lighting model, including properties such as attenuation and introducing basic diffuse and normal mapped textures. Please refer to Week 2’s MyFirstProject for ways of implementing this.

**Step 4: Assignment Research**

Perhaps the most difficult part of your assignment will involve writing a shader that handles multiple light sources, including considering their properties (e.g. point light vs spotlight) and accounting for how an object is lit when said multiple sources interact with it. You should expect to spend some time researching how you can write such a shader, and how you can do so efficiently, making use of various keywords and variables supported by ShaderLab and its #include files. There are many excellent online resources that will guide you through this process.