SHADERS LAB 02

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 Two: Specular Lighting

In this section of the shader unit, the student will look at how to accomplish a specular lighting technique in their shaders. This will allow light to give highlights to the portion of the object where the light is shining the most. The student will also move their shading from the vertex program into the fragment program, and observe the differences in the quality of the lighting as well as the performance of the shader.

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 SpecularShader, SpecularMaterial, SpecularCapsule.

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

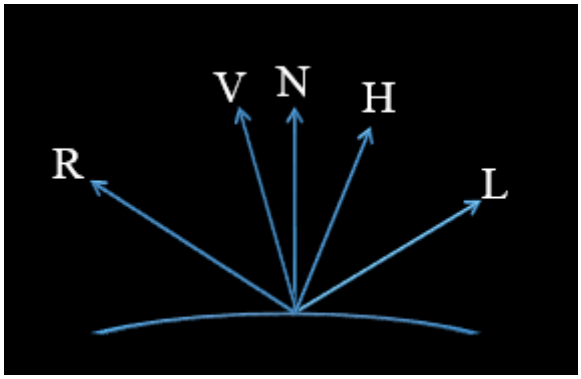
Calculating the specular lighting for your object is much easier once you understand what specular lighting actually is. Specular lighting (more commonly referred to as specular highlighting) is the bright spot of light that appears on shiny objects when they are illuminated.

When you were working with flat lighting, you were just concerned with coloring the object. When you added lambert we added two new things – the ambient light that shines on the object which is a single calculating that corresponds to the strength of the ambient light. There is no real shading that goes into ambient light, just how intense the light is.

You also added a basic diffuse lighting, taking into considering a single light and what direction that single light was shining on the object – that gave you the dark and light parts of the object.

When you want to work with specular highlighting, you are concerned will both the strength of the ambient lighting, the direction of the light, as well as the position of the viewer.

The main idea here is that when light strikes a surface at a certain angle, it is also **reflected** away at the same angle. Remember when you were looking at the lambert shader all the different types of vectors were presented:



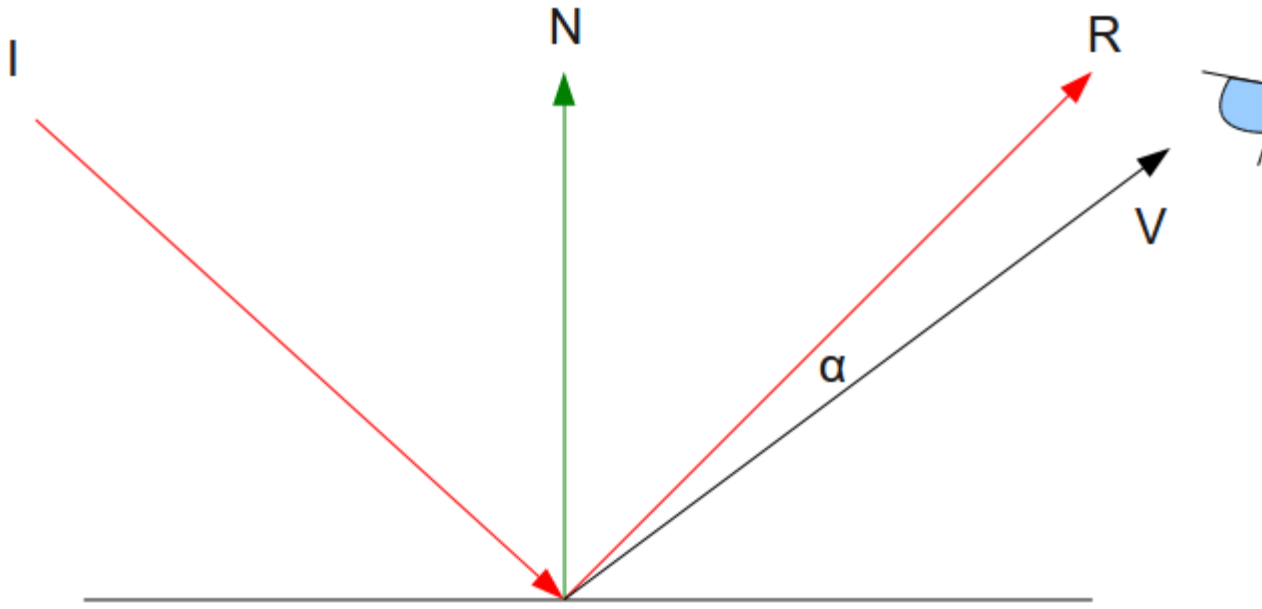
1. The Light Direction (L) corresponds to the direction the light is coming from.
2. The View Direction (V) corresponds to how the user is looking at the object.
3. The Normal Direction (N) is the direction straight away from the surface.
4. The Reflected Direction (R) is the resulting direction if the light direction is reflected across the normal.
5. The Halfway Direction (H) is the vector halfway between the view and the light.

Also recall that you can calculate the distance and angle between two vectors by utilizing the dot product.

The end result of specular highlighting is that the object will look brighter from certain angles and this brightness will diminish as you move away. The “perfect” real world example of specular highlighting is metallic objects. These kinds of objects can sometimes be so bright that instead of seeing the object in its natural colour you see a patch of shining white light which is reflected directly back at you.

However, this type of quality which is very natural for metals is absent in many other materials like wood. Many objects don’t shine, regardless of where the light is coming from and where the viewer is standing. The conclusion is that the specular factor depends more on the object, rather than the light itself.

Considering the following picture:



1. I is the direction the light is coming from
2. N is the object's normal (straight away from the surface)
3. R is the resulting direction if the light direction is reflected across the normal
4. V is how the user is looking at the object.
5. α is the angle between the view direction and the reflected direction.

You can model specular lighting by using the angle 'α'. The idea behind specular light is that the strength of the reflected light is going to be at its maximum along the vector 'R' – this is because the user is viewing at the direct reflection of the light on the shiny object. If this were to be the case (the user is viewing directly at the reflection) then the resulting angle between V and R would be zero. This is the strongest point of the reflection.

As the user's view moves away from the reflection, the angle between the view and the reflection increases. To create a quality looking shader, you will want to create a smooth transition from "super bright" (when the user is staring directly at the reflection) and "less bright" (as the user's view moves away from the reflection). You can do that with the falloff of the light similar to the lambert shading.

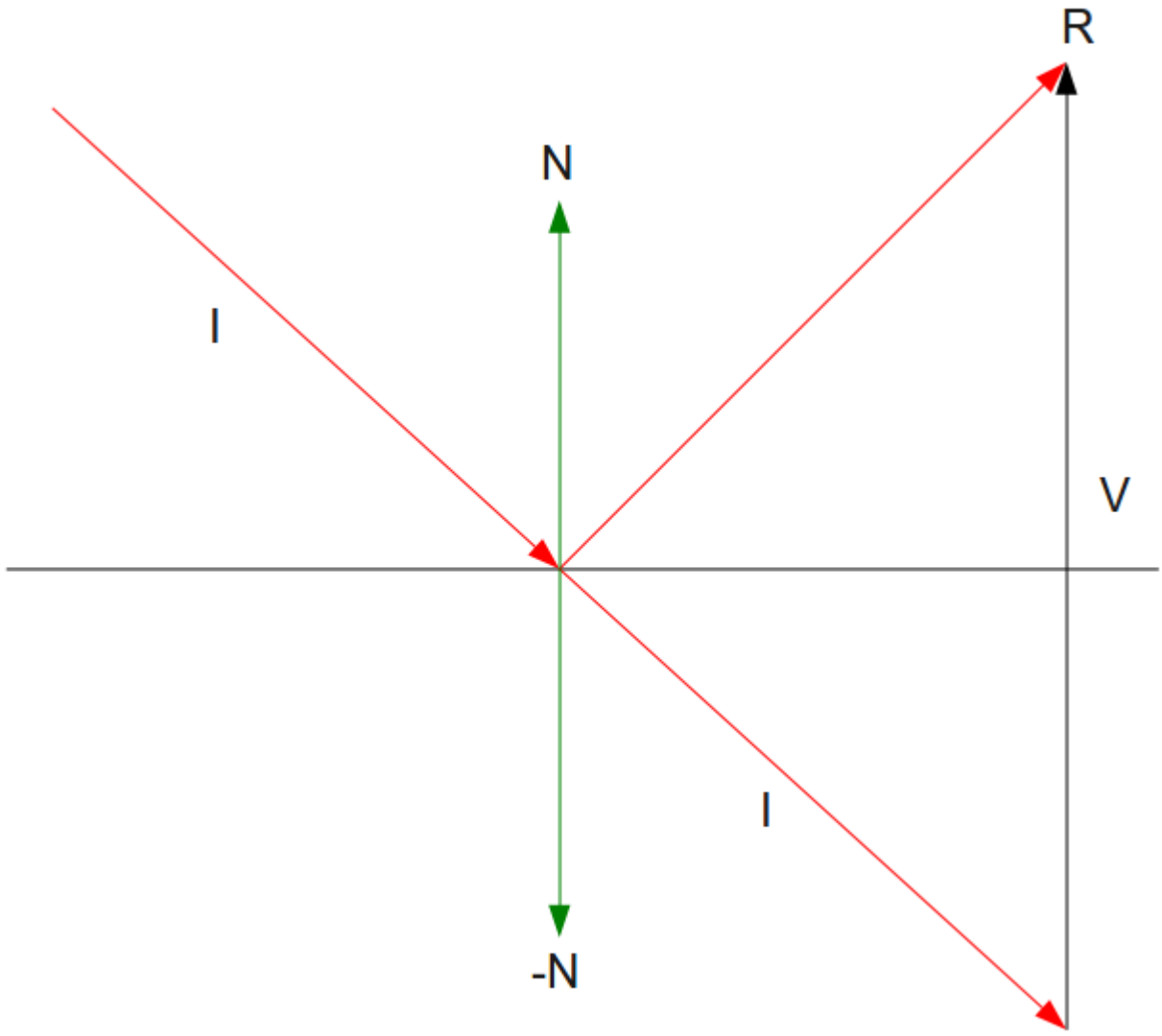
MATH DETAILS

If you are comfortable with matrix multiplication and vector manipulation (adding, subtracting, dot product, and vector projection) and wish to know more of the math behind what is going to happen, feel free to read the next section. Otherwise, you can skip this section and return to it after you have taken Math and Physics for Games and have had the opportunity to explore vector calculations in detail.

To calculate α you will need both 'R' and 'V'. 'V' can be calculated by subtracting the location where the light is hitting from the location of the viewer (both in world space). You already know the location of the viewer, supplied

to you by unity. In order to calculate the reflection vector, you need to know both the normal of the object and the light direction. The light direction is also supplied to you via unity (and any other application you may use for shading), and you already looked at how to calculate the normal direction of the object.

Calculating V can be accomplished using the following:



Where –N is the negative normal of the object.

Vectors have no starting point, and vectors that have the same direction and magnitude are considered equal. Therefore, you can copy the vector I to below the surface, and it will still be considered identical to the original vector I. Your primary objective is to find the vector 'R'. Based on the rules of vector addition, 'R' is equal to 'I' + 'V'. 'I' is already known, so you only need to calculate V at this point.

By using the dot product operation between 'I' and '-N' you can find the magnitude of the vector that is created when 'I' is projected onto '-N'. This magnitude is exactly half of the magnitude of 'V'. Since 'V' has the

same direction as 'N' you can calculate 'V' by multiplying 'N' (whose length has been normalized) by twice that magnitude. This is where the following formula that you will be performing is derived from:

$$R = I + V$$
$$V = 2 * N * (-N \cdot I)$$
$$\Rightarrow R = I + 2 * N * (-N \cdot I) = I - 2 * N * (N \cdot I)$$

Where I is the direction of the light, V is the view direction, N is the normal direction, and R is the reflected light direction over the normal.

Once you have that, you can calculate the specular lighting.

$$\begin{pmatrix} R_{specular} \\ G_{specular} \\ B_{specular} \end{pmatrix} = \begin{pmatrix} R_{light} \\ G_{light} \\ B_{light} \end{pmatrix} * \begin{pmatrix} R_{surface} \\ G_{surface} \\ B_{surface} \end{pmatrix} * M * (R \cdot V)^{p}$$

You start by multiplying the colour of the light by the colour of the surface. This is the same as with ambient and diffuse lighting. The result is multiplied by the specular intensity of the material ('M'). A material which does not have any specular property (e.g. wood) would have a specular intensity of zero which will zero out the result of the equation. Shinier materials such as metal can have increasingly higher levels of specular intensity. After that you multiply by the cosine of the angle between the reflected ray of light and the vector to the view (aka the dot product). Note that this last part is raised to the power of 'P'. 'P' is called the 'specular power' of the 'shininess factor'. It's job is to intensify and sharpen the edges of the area where the specular light is present. The specular power is an attribute of the material, and will have different values depending on the material.

PREVIEWING THE CODE

```
float3 viewDirection = normalize(float3(float4(_WorldSpaceCameraPos.xyz, 1.0) -
mul(_Object2World, input.vertexPos)).xyz);
float3 specularReflection = max(0.0, dot(normalDirection, lightDirection)) * pow(max(0.0,
dot(reflect(-lightDirection, normalDirection), viewDirection)), _Shininess);
```

To break these lines of code down:

float3 viewDirection = normalize(float3(float4(WorldSpaceCameraPos.xyz, 1.0) - mul(Object2World, input.vertexPos)).xyz);

To get the view direction, you need to calculate the distance between the direction the camera is facing and the vertex's position. If you remember from the vector unity in GAME 121, to find the distance between two vectors you simply subtract them.

You already know how to get the vertex position, using mul(UNITY_MATRIX_MVP, input.vertexPos). However, that returns the vertex in object space. What you want is to get the vertex in WORLD space (the same coordinate system that the camera uses). To do that, you use _Object2World in place of UNITY_MATRIX_MVP

To get the camera position, you use _WorldSpaceCameraPos.xyz. You aren't concerened with the w component of the camera, so there is no reason to extract it.

Now you have the camera and the vertex position, and you can simple subtract them. Keep in mind that the vertex position is a float4, and you must subtract vectors with the same number of components (meaning you can't subtract an xyz from an xyz and w). This means that you will need to caste your camera position as a float3 in order to do the subtraction.

Finally, you will normalize this value just to remove any possibilities of artifacts happening.

```
float3 specularReflection = max(0.0, dot(normalDirection, lightDirection)) * pow(max(0.0, dot(reflect(-lightDirection, normalDirection), viewDirection)), _Shininess);
```

This is the one line that will make the entire specular highlighting system work. It just happens to be extremely long.

Building it from scratch:

1. First you need to reflect the light direction based on the surface normal, the reflect function does just that, takes the first ray and reflects it across a specified axis, in this case you use the normal direction.
   - float3 specularReflection = reflect(lightDirection, normalDirection);
2. Next, you need to use a dot product to get your white highlight based on this reflected vector and the view direction.
   - float3 specularReflection = dot(reflect(-lightDirection, normalDirection), viewDirection);
   a. You can use just the dot product to get the highlight, however this results in the highlight being on the wrong side. That is why lightDirection is flipped.
3. Now you can stop the highlight from wrapping around using the a good old max function to limit the dot product to positive values
   - float3 specularReflection = max(0.0,dot(reflect(-lightDirection, normalDirection), viewDirection));
4. Almost done, you can now add some control over the intensity of the specular highlight
   - float3 specularReflection = pow(max(0.0,dot(reflect(-lightDirection, normalDirection), viewDirection)), _Shininess);
   a. You bring in the pow() function, the pow() function will take the first number to the power of the second, for example pow(a,8) = a*a*a*a*a*a*a*a.
   b. Adding a Shininess variable will do nicely here.
5. Finally, you want the specular highlight to fade as it gets closer to the edge. You can use the same dot product from the lambert function to control how much it fades, remember 1 is full and 0 is none
   - float3 specularReflection = max(0.0, dot(normalDirection, lightDirection)) * pow(max(0.0,dot(reflect(-lightDirection, normalDirection), viewDirection)), _Shininess);

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/SpecShader" {
	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"
}
```

You will need to setup a property to allow the user to change the specular colour of the shader. This will be another colour.

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

You will also have to setup a property to change the shininess of the object. This must be named _Shininess for the same reason that you must call you MainTextures MainTex. If you wish to use other built-in functions for the shininess of your object, those functions except the property to be named _Shininess and will except no other property. The shininess will be a float.

```
Properties {
	_Color ("Color", Color) = (1,1,1,1)
	_SpecColour("Specular Color", Color) = (1,1,1,1)
	_Shininess("Shininess", float) = 10

}
```

Next you will need to link these properties into the shader by declaring the variable for it to interface with.

```
//user defined variables
uniform float4 _Color;
uniform float4 _SpecColour;
uniform float _Shininess;
```

You will also want to get the light color for the shader:

```
//unity defined variables
uniform float4 _LightColor0;
```

In your input struct, you will want the position and the normal (since you will layering on a lambert shader!). In your output struct you will need the DX11 position and the color. So far, nothing new.

```
//input struct
```

```
struct inputStruct
{
	float4 vertexPos : POSITION;
	float3 vertexNormal : NORMAL;

};

//output struct
struct outputStruct
{
	float4 pixelPos: SV_POSITION;
	float4 pixelCol : COLOR;
};
```

Next you will go into your vertex function. Inside of your vertex function you will need three pieces of information: the light direction, the normal (so you can reflect the light direction), and the way the player is viewing (so you can create the highlight based on where they are viewing relative to the reflection).

You already explored how to calculate the normal in previous assignments:

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

	float3 normalDirection = normalize(mul(float4(v.normal,
0.0), _World2Object).xyz);
```

The next thing that you will calculate is the view direction.

To get the view direction, you need to calculate the distance between the direction the camera is facing and the vertex's position. If you remember from the vector unity in GAME 121, to find the distance between two vectors you simply subtract them.

You already know how to get the vertex position, using **mul(UNITY_MATRIX_MVP, input.vertexPos)**. However, this returns the vertex in object space. The camera resides in world space. So you will want to get the vertex position in world space: **mul(_Object2World, input.vertexPos)**.

To get the camera position, you use **_WorldSpaceCameraPos.xyz**. You aren't concerened with the w component of the camera, so there is no reason to extract it.

Now you have the camera and the vertex position, and you can simple subtract them. **_WorldSpaceCameraPos.xyz - mul(_Object2World, input.vertexPos)**

Keep in mind that the vertex position is a float4, and you must subtract vectors with the same number of components (meaning you can't subtract an xyz from an xyz and w). This means that you will need to caste your camera position as a float4 in order to do the subtraction.

float4(_WorldSpaceCameraPos.xyz**, 1.0)** - mul(_Object2World, input.vertexPos**)**

However, viewDirection is assigned as a float3! This means that you need to against cast the result of the subtraction as a float3 to drop the w component.

float3(float4(_WorldSpaceCameraPos.xyz, 1.0) - mul(UNITY_MATRIX_MVP, input.vertexPos)**.xyz)**

Finally, you will normalize this value just to remove any possibilities of artifacts happening.

normalize(float3(float4(_WorldSpaceCameraPos.xyz, 1.0) - mul(_Object2World, input.vertexPos))**)**

```
float3 normalDirection = normalize(mul(float4(input.vertexNormal, 0.0), _Object2World).xyz);
float3 viewDirection = normalize(float3(float4(_WorldSpaceCameraPos.xyz, 1.0) - mul(_Object2World, input.vertexPos).xyz));
```

To preview this, you can set the output color of the vertex to the view direction.

```
toReturn.pixelCol = float4(viewDirection, 1.0);
toReturn.pixelPos = mul(UNITY_MATRIX_MVP, input.vertexPos);
```

Don't forget to tell your fragment program to output the color!

```
//fragment program
float4 fragmentFunction(outputStruct input) : COLOR
{
	return input.pixelCol;
}
```

You should see the colour of the object change depending on how you are looking at the object (it is probably a very slight difference, usually between pink and a light orange).

Next, you can calculate the light direction. You should be familiar with calculating the light direction.

```
float3 lightDirection;
float attenuation = 1.0;

lightDirection = normalize(_WorldSpaceLightPos0.xyz);
```

Next is calculating the diffuse reflection. This is the same exact way it was calculated in the lambert shader.

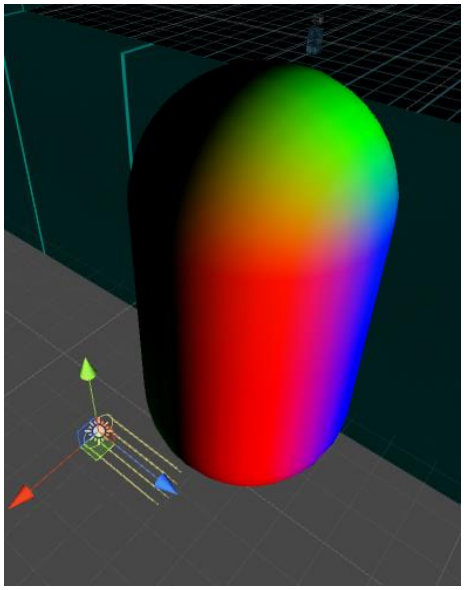
```
        //dot product of the normal and the light direction to calc a value
between 0 and 1
        //if normal is facing the light it gets 1, if facing away it gets a 0.
        float3 diffuseReflection = attenuation * _LightColor0.xyz * max(0.0,
dot(normalDirection, lightDirection));
```

Now onto the new stuff. You will build up the specular reflection piece by piece, and look at what each piece will display as you go along.

The first step is to reflect the normal across the light direction. There is a significant amount of mathematical theory that was detailed in Section Two, so for simplicity you will use the built-in Cg function reflect(reflect this vector, over this axis), and then display the reflection onto the object.

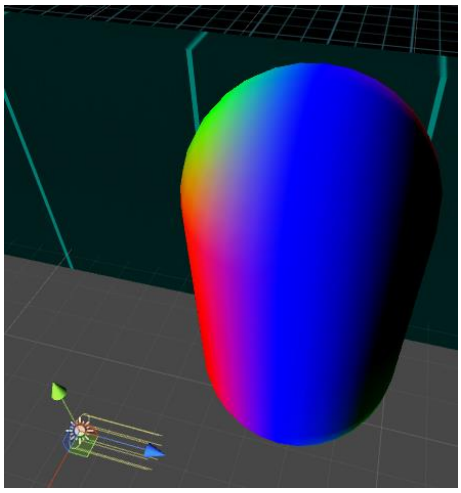
```
                float3 specularReflection = reflect(lightDirection, normalDirection);

                toReturn.pixelCol = float4(specularReflection, 1.0);
```



Your object should look something like the one displayed above. If you notice, the directional light that is shown in the preview is shining onto the object. The specular highlight will appear on the objects where all of the colours come together (the greenish area). However, you should notice something odd about this. When you were going over the theory, the big thing was that the highlight will be the brightest if the user is looking at the reflection of the light. The reflection of the light SHOULD be facing the directional light. To solve this, you just invert the light direction on your calculation.

```
                float3      specularReflection      =      reflect(-lightDirection,
normalDirection);
```



So now that you have the reflection, you want to calculate the angle between the view and the reflection. This will allow you to increase/decrease the intensity of the highlight depending on where the user is viewing the object from (remember: if the user is looking directly at the reflection it will be brighter than if the user is looking more away from the reflection).

```
                float3 specularReflection = reflect(-lightDirection,
normalDirection);
                specularReflection = dot(specularReflection, viewDirection);
```

Test this to see what it looks like.

The next thing will be to make sure that this to a minimum of 0 (meaning that the calculation cant go below zero). This will stop you from getting the highlighting on the reverse side.

Test this to see what it looks like.

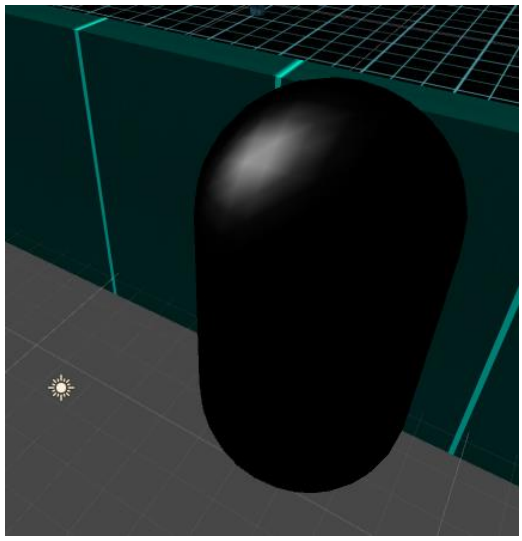
There is still just a little bit of a bleed onto the backside, this helped soften it a little bit but not entirely. So to solve this you can utilize your lambert's method of stopping light from shining on the opposite side of the light:

```
                specularReflection = max(0.0, specularReflection);
                specularReflection = max(0.0, dot(normalDirection,
lightDirection)) * specularReflection;
```

Test this to see what it looks like.

Next will be to get the shininess to affect the object. This is done by putting the calculated light (before softening it) to the power of shininess.

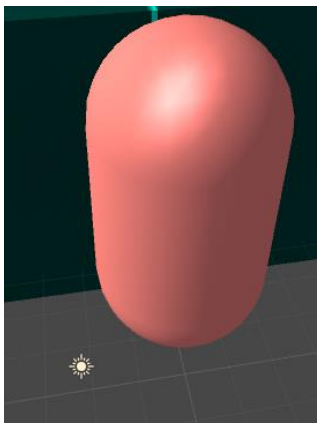
```
                specularReflection = pow(max(0.0, specularReflection), _Shininess);
                specularReflection = max(0.0, dot(normalDirection, lightDirection)) *
specularReflection;
```



Finally you can add the diffuse with the specular with the ambient to get your final lighting, and multiply that by the flat colour to get your final color.

```
                float3 finalLight = specularReflection + diffuseReflection +
UNITY_LIGHTMODEL_AMBIENT;

                toReturn.pixelCol = float4(finalLight * _Color, 1.0);
```



One last thing you can add is the ability to change what color the specular color is! Hopefully you can think of the way to solve this on your own (make sure to multiply it by attenuation as well – attenuation * _SpecColor.rgb).

STEP FOUR: PACKING UP AND MOVING OUT

With the specular shader done, it is about time to move out of the vertex program and into the fragment program. You have created some very interesting shaders in the vertex program, but you can make them look much better if you do the same thing – in the fragment program instead.

There are a few subtle differences between the vertex and the fragment program – most noticeable is the inability to get the position.

So, the first thing that you will want to do is determine what in the vertex function do you need to keep – in this case you need to keep the three lines that every vertex function must have

```
outputStruct toReturn;


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

return toReturn;
```

You also want to keep your world space position calculation – this is a very important calculation in your lighting model, and without it your entire lighting model will break! Mainly because you use the vertex position to calculate the view direction of the player. You don't want to keep the entire view direction however, because you want the view direction to be pixel perfect (hence why you are using the fragment function instead of the vertex function!).

You will also want to keep the normal calculation for calculating the objects normal.

In order to pass these two things into your fragment function, you will need to include them into your output struct:

```
//output struct
struct outputStruct
{
	float4 pixelPos: SV_POSITION;
	float4 pixelCol : COLOR;

	float3 normalDirection : TEXCOORD0;
	float4 pixelWorldPos : TEXCOORD1;
};
```

You use TEXCOORD0 and TEXCOORD1 as the semantic here because you must assign something. TEXCOORD0 to TEXCOORD12 are semantics that can be used to pass values between the vertex and the fragment functions. This is something unique to Cg. After working through the texture lab you know that TEXCOORD0 and TEXCOORD1 are

responsible for texture maps. However, since you are not using texture maps in this shader it is acceptable to just overwrite that information with your own personal information.

There is an upper limit for how many TEXCOORDs you can have. Some systems only support 4, others 12, others 16. It will really depend on the system you are running.

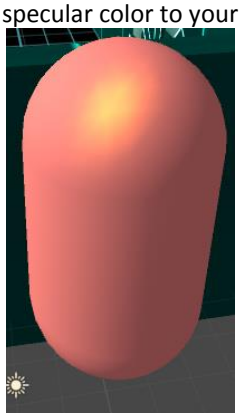
WRAP UP

You now have the ability to add and adjust specular lighting on a shader, as well as move vertex lighting into the fragment program for more detailed effects.

THINK ABOUT IT!

1. How does `max(0.0, specularReflection)` soften the colour bleed onto the backside of the object when using a specular shader?
2. How does `max(0.0, dot(normalDirection, lightDirection))` remove the specular highlight from appearing on the back of the object?

ON YOUR OWN!

a. Add the specular color to your specular shader
   i. 
b. Assign the two variables (toReturn.normalDirection and toReturn.pixelWorldPos) in your vertex function
   i. Hint: Don't assign the ENTIRE view direction – ONLY assign the vertices world position!
c. Move all of your code to the fragment function
   i. With the exception of the three lines every vertex function MUST have, as well as the two variable declarations (toReturn.normalDirection and toReturn.pixelWorldPos).