# Tutorial – Direct Lighting

3-D models commonly contain surface normal information along with other information such as vertex position. These normals can be used within a shader to correctly light a model by determining if it is facing towards the light or not.

In this tutorial we will be adding normal to the Mesh class that we created in previous tutorials, then drawing it using a simple Phong diffuse shader. Then we will use the same shader to draw an OBJMesh and extend the shader to incorporate a specular component and use the OBJ's materials.

## Adding Normals to our Mesh class

If you have completed the previous Rendering Geometry with OpenGL and Materials and Textures tutorials then your Mesh class will currently make use of Position and Texture Coordinate vertex attributes. We now need to add normals. Our Vertex structure includes them as the second property:

```
	struct Vertex {
		glm::vec4 position;
		glm::vec4 normal;
		glm::vec2 texCoord;
	};
```

We can easily add Normals to our initialiseQuad() method. Since the quad is laying flat on the ground, facing "up", we simply need to set all 6 vertices to have a normal that faces "up", which is a unit vector in the direction (0, 1, 0, 0) for XYZW:

```
	vertices[0].normal = { 0, 1, 0, 0 };
	vertices[1].normal = { 0, 1, 0, 0 };
	vertices[2].normal = { 0, 1, 0, 0 };
	vertices[3].normal = { 0, 1, 0, 0 };
	vertices[4].normal = { 0, 1, 0, 0 };
	vertices[5].normal = { 0, 1, 0, 0 };
```

We then enable the vertex attribute in the same way we enabled them for texture coordinates. We'll enable attribute 1 (the 2nd attribute) and set it to use 4 GL_FLOAT elements. We can then use the parameter GL_TRUE to automatically normalise the vector when it is transferred to the GPU. The bytes between one set of normal data to the next is the sizeof(Vertex), and the final parameter is how many bytes into the Vertex struct is the normal property, which comes just after the first vec4, so 16 bytes:

```
	// enable second element as normal
	glEnableVertexAttribArray(1);
	glVertexAttribPointer(1, 4, GL_FLOAT, GL_TRUE,
	                      sizeof(Vertex), (void*)16);
```

While we're updating our Mesh class we can also enable the normals within the generic initialise() method. We simply need to add the previous code snippet to enable the attribute, right after we enabled the position attribute.

Add normals to any other initialise methods you have added to the Mesh class. How could we do that for a sphere? What direction would the normal vectors face? What about for a pyramid?

## Phong Diffuse Lighting Shader

The Phong lighting model is a simple, though looks complex, model that approximates basic diffuse and specular lighting.

$$I_p = k_a i_a + \sum_{m \in lights} \left(k_d \left(\hat{L}_m \cdot \hat{N}\right) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^a i_{m,s}\right)$$

For now we will just set up the diffuse portion, and later we will add in specular.

We start by creating a Vertex Shader. This shader needs to make use of a position so that we can see the geometry, but it also needs a normal vector that we can use in the lighting equation:

*Vertex Shader (phong.vert):*

```
// classic Phong vertex shader
#version 410

layout( location = 0 ) in vec4 Position;
layout( location = 1 ) in vec4 Normal;

uniform mat4 ProjectionViewModel;

void main() {
	gl_Position = ProjectionViewModel * Position;
}
```

We will be implementing the lighting on a per-fragment basis, so the Vertex Shader simply needs to transform the normal and send it through to the Fragment Shader stage. But how do we transform a normal?

Our Normals all begin in local-space, or model-space, but we need it to be in the same space as our lights, which right now will be in world-space. To transform a position from local to world space we typically transform it by the model transform, but that doesn't quite work for a normal. This is because any translation or scale included in the transform will ruin the vector. Instead, we need to use what is referred to as a normal transform, or normal matrix.

The normal matrix is a 3x3 matrix, or a 4x4 matrix with a translation of (0,0,0,1), and is the inverse-transpose of the model's transform. What this results in is a matrix that will rotate the normal correctly to follow the model's orientation, but it wont adversely scale or translate the normal vector when it is transform.

We can add the normal matrix as a uniform into out shader, and use it to transform the input Normal, and store the output in an out variable, which we'll call vNormal, for the fragment stage.

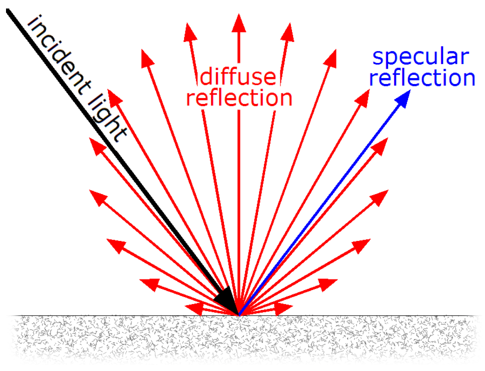
```
// classic Phong vertex shader
#version 410

layout( location = 0 ) in vec4 Position;
layout( location = 1 ) in vec4 Normal;

out vec3 vNormal;

uniform mat4 ProjectionViewModel;

// we need this matrix to transform the normal
uniform mat3 NormalMatrix;

void main() {
      vNormal = NormalMatrix * Normal.xyz;
      gl_Position = ProjectionViewModel * Position;
}
```

When rendering we simply bind the matrix along with any other transform that was needed, for example:

```
      // bind transform
      auto pvm = m_projectionMatrix * m_viewMatrix * m_modelTransform;
      m_phongShader.bindUniform("ProjectionViewModel", pvm);

      // bind transforms for lighting
      m_phongShader.bindUniform("NormalMatrix",
              glm::inverseTranspose(glm::mat3(m_modelTransform)));
```

For now, our Vertex Shader is set up, at least to be able to handle diffuse lighting, so it's time to look at the Fragment Shader.

We've sent through the normal as an input to the fragment shader, so to be able to calculate how much diffuse reflection there is we will need just one more variable, the light's direction, or incident.



We bind the light as a uniform vec3, but we can do one of two things; bind the direction the light is traveling or bind the direction the light is coming from. We will bind the direction the light is traveling, which we will do in a moment.

To calculate Phong diffuse we need a Lambert Term, which is simply a dot product between the direction the light is coming from and the surface normal. The result can be thought of as a percentage of how much diffuse reflection there is at that point on the surface, which will be a grayscale value. For lighting, we need this value to be within 0 to 1, so anything below 0 will be clamped to 0, and any floating-point error that results in greater than 1 will be clamped at 1.

Let’s begin by setting up a Fragment Shader that has the input vNormal, a uniform vec3 for the light’s direction, and calculates the dot product and uses that as the output colour:

*Fragment Shader (phong.frag):*

```
// classic Phong fragment shader
#version 410

in vec3 vNormal;

uniform vec3 LightDirection;

out vec4 FragColour;

void main() {

	// ensure normal and light direction are normalised
	vec3 N = normalize(vNormal);
	vec3 L = normalize(LightDirection);

	// calculate lambert term (negate light direction)
	float lambertTerm = max( 0, min( 1, dot( N, -L ) ) );

	// output lambert as grayscale
	FragColour = vec4( lambertTerm, lambertTerm, lambertTerm, 1 );
}
```

Before we go any further we should check that the lighting works, so let’s set up an application with it running.

To do this, add the Vertex and Fragment shaders listed above to your project and create a Mesh that contains the normals set up as previously mentioned.

We will also need data to represent a light. We’ll set up a struct, called Light, and give it direction information, then create an instance of a light. We’ll be adding more information to our light soon:

```
class Application3D : public aie::Application {
public:

	Application3D();
	virtual ~Application3D();

	virtual bool startup();
	virtual void shutdown();

	virtual void update(float deltaTime);
	virtual void draw();

protected:

	glm::mat4	m_viewMatrix;
	glm::mat4	m_projectionMatrix;

	aie::ShaderProgram	m_phongShader;
	Mesh			m_quadMesh;
	glm::mat4		m_quadTransform;

	struct Light {
		glm::vec3 direction;
	};

	Light			m_light;
};
```

Load the shader just as you have in previous tutorials, and initialise the Mesh. The transform should be a suitable transform with whatever scale, rotation and translation that you want.

For the light, we'll set it up to rotate during the application's update, this way we'll see the dynamic effect that lighting has. The following code snippet rotates the light around the Z-axis, in the XY plane, that gives a "sunrise / sunset" effect as the light rotates, and would go within the applications update() method:

```
	// query time since application started
	float time = getTime();

	// rotate light
	m_light.direction = glm::normalize(vec3(glm::cos(time * 2),
	                                        glm::sin(time * 2), 0));
```

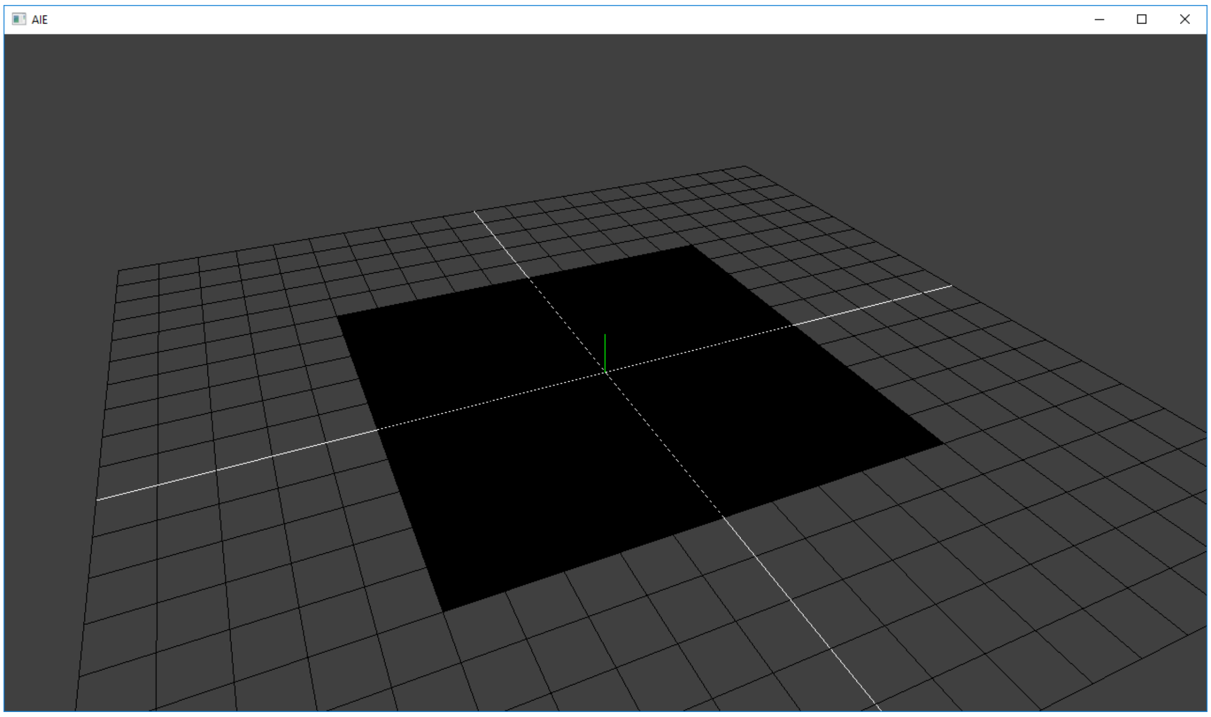
When we render our Mesh we first need to:

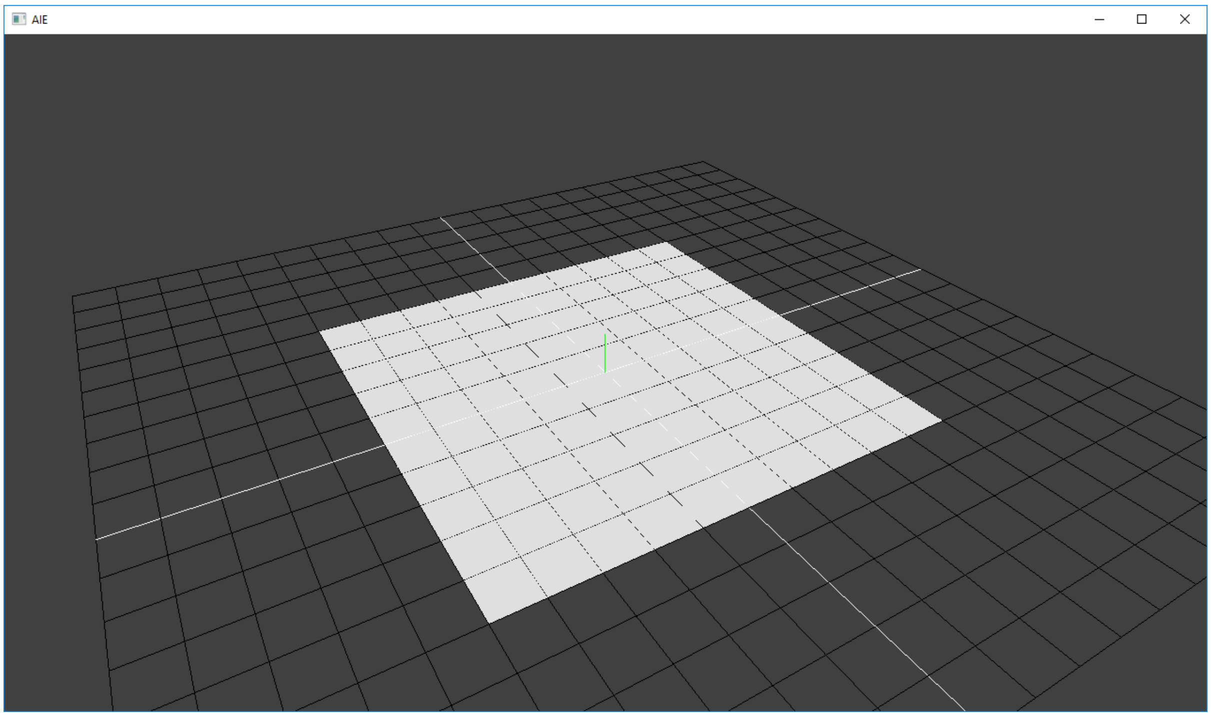
1. Bind the Phong shader
2. Bind the light's direction
3. Bind the Projection x View x Model transforms
4. Bind the Normal matrix
5. Draw the mesh

For example:

```
        // bind phong shader program
        m_phongShader.bind();

        // bind light
        m_phongShader.bindUniform("lightDirection", m_light.direction);

        // bind transform
        auto pvm = m_projectionMatrix * m_viewMatrix * m_quadTransform;
        m_phongShader.bindUniform("ProjectionViewModel", pvm);

        // bind transforms for lighting
        m_phongShader.bindUniform("NormalMatrix",
                    glm::inverseTranspose(glm::mat3(m_quadTransform)));

        // draw quad
        m_quad.draw();
```

If it has been set up and implemented correctly you should see the quad mesh colour changing from black to white as the light rotates:





Not all lights are white, so we need to include colours within our shaders.

Phong lighting includes Ambient, Diffuse and Specular colours. The Ambient is global, but the Diffuse and Specular are specific to each light. We should add the diffuse and specular to our Light struct, but also add a variable to store the ambient colour, and then bind them all to the shader.

In our Phong equation the light properties are called:

- Ambient is called Ia
- Diffuse is called Id
- Specular is called Is

So we will bind the variables to shader uniforms that use that naming convention.

Add the additional properties to the Light struct and add the ambient light. Within the application's startup() method you should set the light's colours to a suitable value. For this example, we will use yellow for the light, and a dark gray for the ambient:

```
	struct Light {
		glm::vec3 direction;
		glm::vec3 diffuse;
		glm::vec3 specular;
	};

	Light		m_light;
	glm::vec3	m_ambientLight;
```

```
	m_light.diffuse = { 1, 1, 0 };
	m_light.specular = { 1, 1, 0 };
	m_ambientLight = { 0.25f, 0.25f, 0.25f };
```

We then need to extend our Fragment Shader to include these new uniforms. For now we will just make use of the diffuse colour as we will incorporate specular and ambient in our next step.

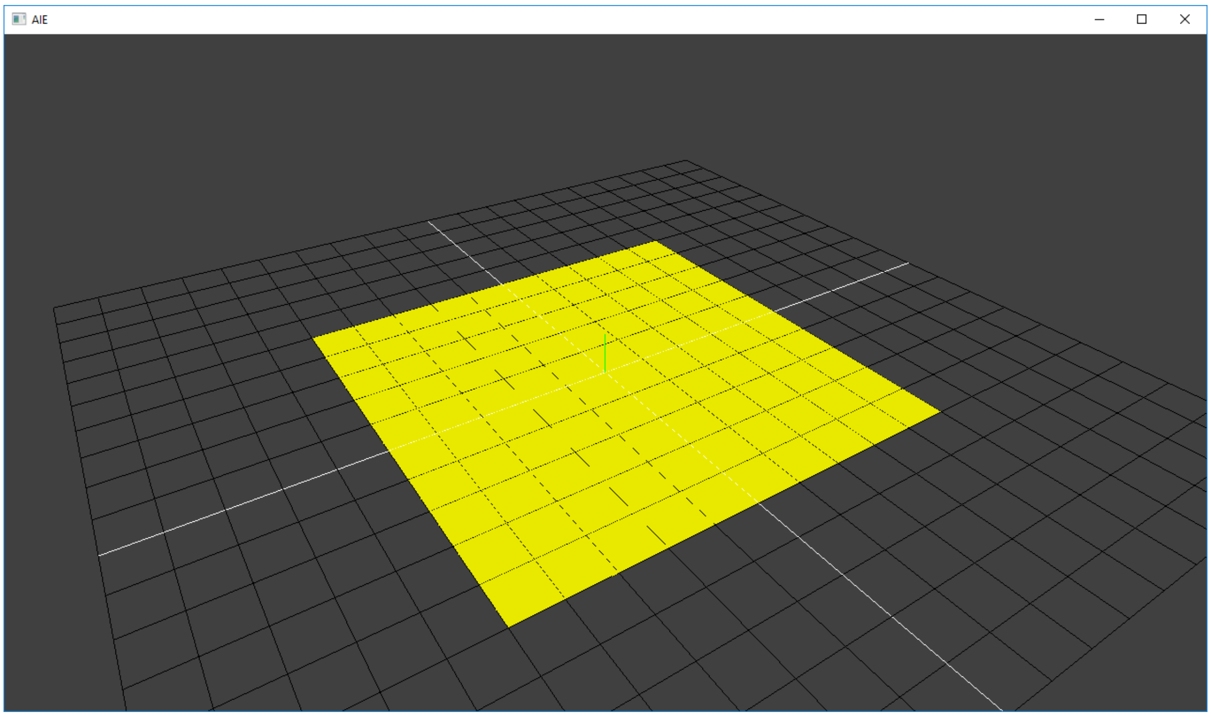
To make use of the colour we simply multiply the lambert term by the diffuse colour, and incorporate that into the output fragment colour:

```
// classic Phong fragment shader
#version 410

in vec3 vNormal;

uniform vec3 Ia; // ambient light colour

uniform vec3 Id; // diffuse light colour
uniform vec3 Is; // specular light colour
uniform vec3 LightDirection;

out vec4 FragColour;

void main() {

	// ensure normal and light direction are normalised
	vec3 N = normalize(vNormal);
	vec3 L = normalize(LightDirection);

	// calculate lambert term (negate light direction)
	float lambertTerm = max( 0, min( 1, dot( N, -L ) ) );

	// calculate diffuse
	vec3 diffuse = Id * lambertTerm;

	// output final colour
	FragColour = vec4( diffuse, 1 );
}
```

We then need to bind the additional uniforms when we render:

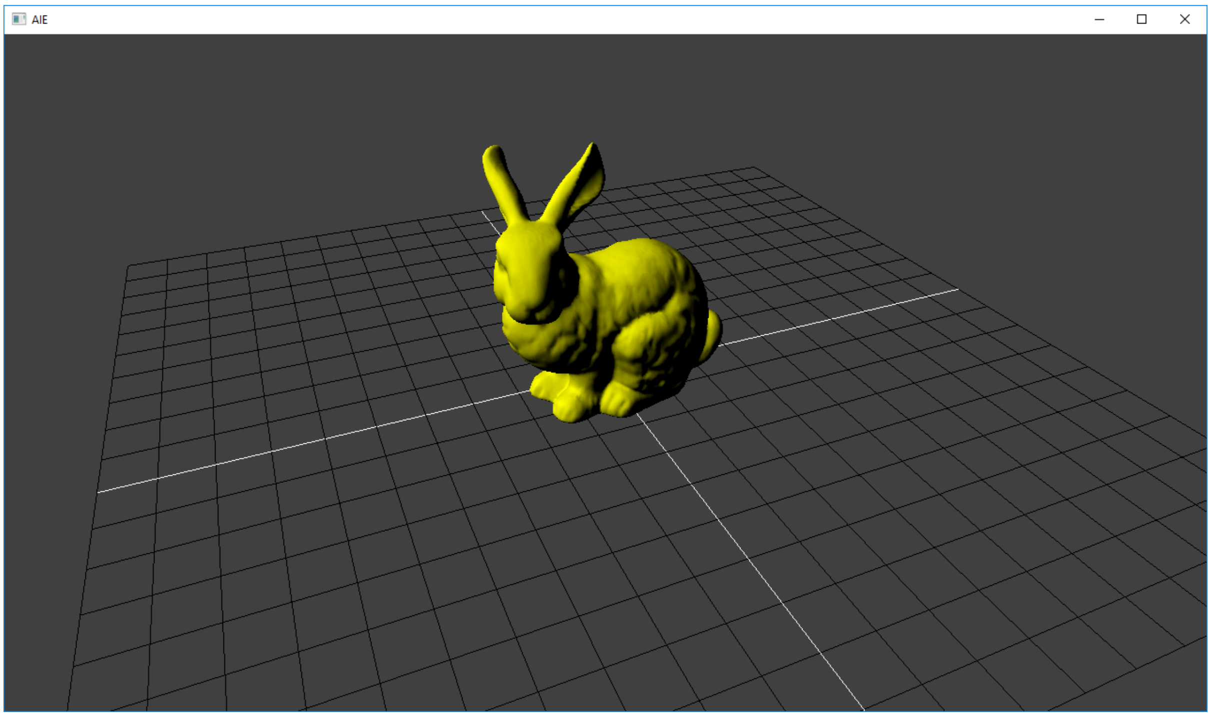
```

	// bind light
	m_phongShader.bindUniform("Ia", m_ambientLight);
	m_phongShader.bindUniform("Id", m_light.diffuse);
	m_phongShader.bindUniform("Is", m_light.specular);
	m_phongShader.bindUniform("lightDirection", m_light.direction);
```

This should result in us now seeing the quad illuminated yellow instead of white:



## Ambient, Specular and Lighting OBJ Meshes

The above shaders will also work when rendering an OBJMesh that had normal. The Stanford models provided on Canvas include detailed models with normals, which can be lit and appear in the following way:



To fully implement the Phong lighting model we need to include the ambient light and the specular reflection, but we also need to include the model's surface material properties. These properties have been called Kd for diffuse, Ka for ambient, and Ks for specular colour. Surface specular properties also include a specular power used to control how intense or "sharp" the specular reflection is.

To begin with, let's include ambient light.

To do so we need to include the model's ambient colour as a uniform within our shader. OBJMesh will automatically bind this colour if the shader makes use of it. It will also automatically bind the materials diffuse, specular, and a specularPower value, if the shader uses it.

Add the following as uniforms to our fragment shader:

```
uniform vec3 Ka; // ambient material colour
uniform vec3 Kd; // diffuse material colour
uniform vec3 Ks; // specular material colour
uniform float specularPower; // material specular power
```

To incorporate ambient within our shader we simply multiply the material ambient by the light ambient, then add it to the colour that we output. While we're at it we can also include the surface diffuse colour by multiplying it by the light's diffuse and the lambert term:

```
        // calculate each colour property
        vec3 ambient = Ia * Ka;
        vec3 diffuse = Id * Kd * lambertTerm;

        FragColour = vec4( ambient + diffuse, 1);
```

Update your Fragment shader to make use of those code snippets. You will see that the lighting for the Stanford Bunny doesn't change much. That's because it's surface colour was a grey already. Try using the Stanford Dragon instead.

The only thing left for us to include is the Specular lighting.

Specular lighting is a little more complex:

$$specular = k_s(\hat{R}_m \cdot \hat{V})^a i_{m,s}$$

It makes use of a vector from the mesh's surface to the camera, called $\hat{V}$, a reflected vector that is the light's direction reflected around the surface normal, called $\hat{R}_m$, in addition to specular colour properties, and a specular power.

To get a vector to the camera we need to know:

- The position of the surface we are trying to light
- The position of the camera

We can send the camera's position to the shader as a uniform:

```
	m_phongShader.bindUniform("cameraPosition",
					vec3(glm::inverse(m_viewMatrix)[3]));
	// or
	m_phongShader.bindUniform("cameraPosition",
					m_camera->getPosition());
```

But to get the surface position we need to send the position from the Vertex Shader to the Fragment Shader, but it needs to be in the same space as the camera. To do that we can send the model transform to the Vertex Shader as a uniform and multiply the position by it and store it in a variable that we output to the Fragment Shader:

```
// classic Phong vertex shader
#version 410

layout( location = 0 ) in vec4 Position;
layout( location = 1 ) in vec4 Normal;

out vec4 vPosition;
out vec3 vNormal;

uniform mat4 ProjectionViewModel;

// we need this matrix to transform the position
uniform mat4 ModelMatrix;

// we need this matrix to transform the normal
uniform mat3 NormalMatrix;

void main() {
	vPosition = ModelMatrix * Position;
	vNormal = NormalMatrix * Normal.xyz;
	gl_Position = ProjectionViewModel * Position;
}
```

Next we need to update our Fragment Shader.

We can calculate the reflection vector by calling the GLSL method reflect() and pass in the light's direction vector and the surface normal.

We can calculate the vector from the surface to the camera by simply subtracting the input vertex position from the camera's uniform position, and then normalise the result.

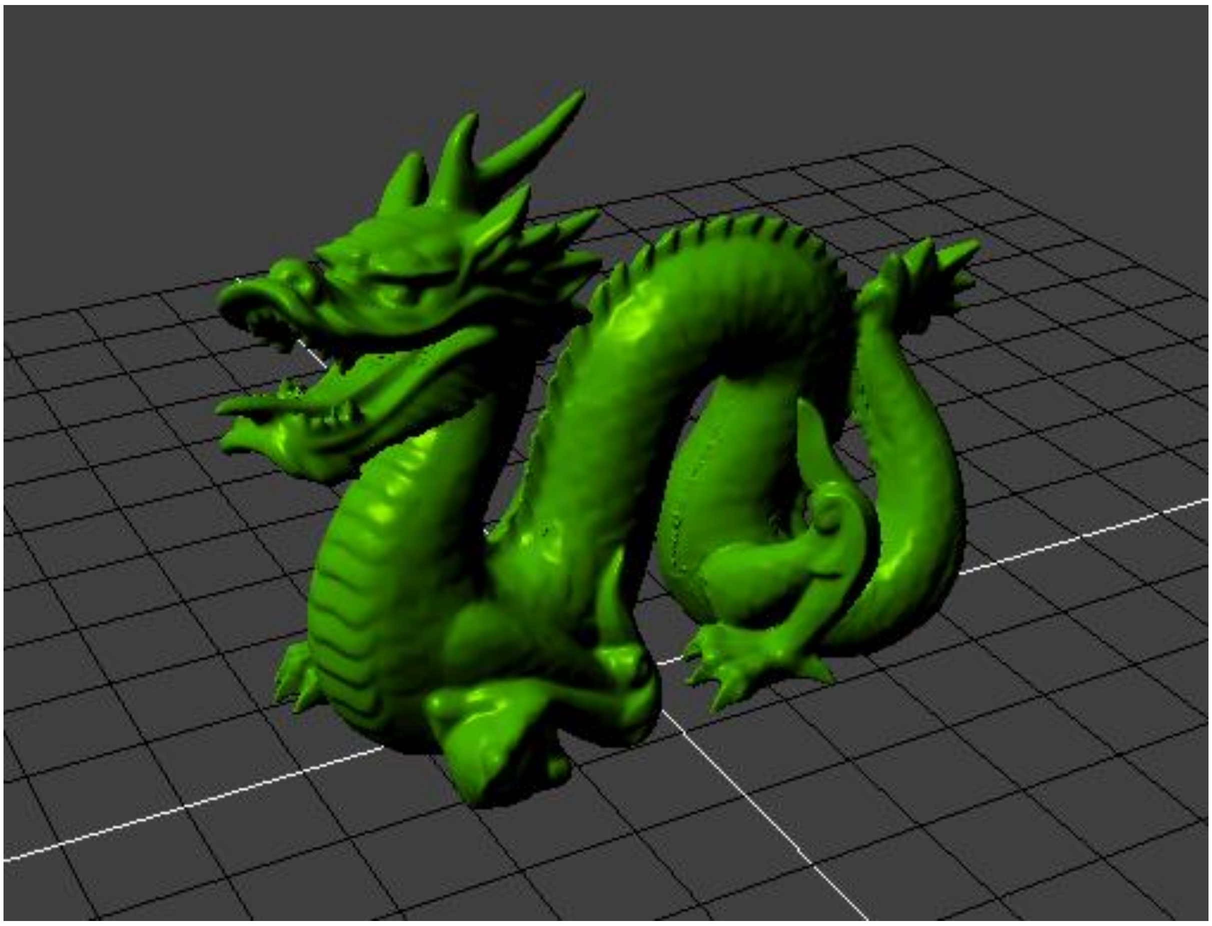
We then perform a dot product between these two vectors, clamp the result between 0 and 1, and then raise the result by the material's specular power. This is the specular term.

The final specular colour is simply the material's specular multiplied by the light's specular, multiplied by the specular term:

*Final Fragment Shader (phong.frag):*

```
// classic Phong fragment shader
#version 410

in vec4 vPosition;
in vec3 vNormal;

uniform vec3 Ka; // ambient material colour
uniform vec3 Kd; // diffuse material colour
uniform vec3 Ks; // specular material colour
uniform float specularPower; // material specular power

uniform vec3 Ia; // ambient light colour

uniform vec3 Id; // diffuse light colour
uniform vec3 Is; // specular light colour
uniform vec3 LightDirection;

out vec4 FragColour;

void main() {

      // ensure normal and light direction are normalised
      vec3 N = normalize(vNormal);
      vec3 L = normalize(LightDirection);

      // calculate lambert term (negate light direction)
      float lambertTerm = max( 0, min( 1, dot( N, -L ) ) );

      // calculate view vector and reflection vector
      vec3 V = normalize(cameraPosition - vPosition.xyz);
      vec3 R = reflect( L, N );

      // calculate specular term
      float specularTerm = pow( max( 0, dot( R, V ) ), specularPower );

      // calculate each colour property
      vec3 ambient = Ia * Ka;
      vec3 diffuse = Id * Kd * lambertTerm;
      vec3 specular = Is * Ks * specularTerm;

      FragColour = vec4( ambient + diffuse + specular, 1);
}
```

And with all of that we should now have a complete Phong lighting shader that can work on any model that includes normal and has the material properties set:



What would we need to modify in our shaders to support adding textures?