08214 Lab 3: Lighting and Material Properties

Welcome to the lighting and material properties lab. To begin with in this lab we set up a test scene to perform lighting in. This should help reaffirm some of what you know about matrices working together. Then we'll develop a lighting model in shaders that includes model material, model topography and different types of light and light sources. There's a lot to learn in this lab, but take your time and make sure you understand each step as we go. In this lab you'll implement lighting in shader code for directional and positional lights, modelling diffuse, specular and ambient light, and material reflectivity properties.

If you have any problems (or spot a mistake) one way you can get help is on the forum thread for lab 3 here:

http://forums.net.dcs.hull.ac.uk/p/1944/6526.aspx

Before you begin this lab, you ***may*** need to add the following files to your solution, as well as setting them to copy to the output directory when the project is built:

Lab3/Shaders/fPassThrough.frag

Lab3/Shaders/vLighting.vert

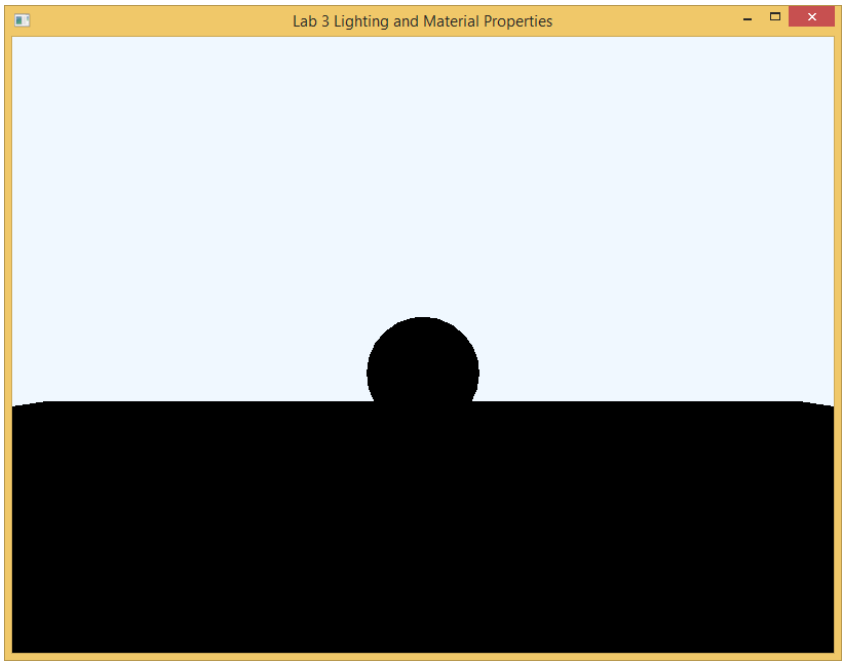
Utility/Models/cylinder.bin

Utility/Models/model.bin

Utility/Models/sphere.bin

Then in the Program.cs file change the m_CurrentLab value to Lab.L3

Once you have done all that your code should compile, and you should see a black window. In the OnLoad method change the clear colour to something other than black. You should then see something like this screenshot.



Commit your code to SVN with the message:

L3T1 Changed the clear colour to see some silhouettes

There is a lot of code in this lab. Let's take some time to recap and make sure we understand what it all does.

```
private int[] mVBO_IDs = new int[3];
private int[] mVAO_IDs = new int[2];
private ShaderUtility mShader;
private ModelUtility mSphereModelUtility;
private Matrix4 mView, mSphereModel, mGroundModel;
```

First we have some declarations. There are some integers to store handles to our VBOs and VAOs, a shader object, a model object and matrices for the view (camera position), a matrix for our sphere and a matrix for the ground. We're using a VAO for the ground and another for the sphere. We only have three VBOs because the ground doesn't have an index buffer. Instead we will just use DrawArray to render the ground.

In the OnLoad method we are doing the usually generating buffer, loading data into VBOs and setting state. You should be familiar with this code already. Code is provided to load a simple model from the disk into RAM. To keep things fast this is held in a binary format. This should work for most setups but there is a *small* chance that it won't if the system store bits in a different way. Fingers crossed!

One thing that is worth noting is how the matrices are set up and used.

```
mView = Matrix4.CreateTranslation(0, -1.5f, 0);
int uView = GL.GetUniformLocation(mShader.ShaderProgramID, "uView");
GL.UniformMatrix4(uView, true, ref mView);

mGroundModel = Matrix4.CreateTranslation(0, 0, -5f);
mSphereModel = Matrix4.CreateTranslation(0, 1, -5f);
```

Here the camera view is moved up by 1.5 units (or rather, the world is moved down), and the new value is set in the shader. The ground model (which is a flat, 20 by 20 square) is moved five units into the scene, and the sphere model is translated one unit up and five units into the scene. When we render, we're going to multiply the sphere model by the ground model, moving the sphere into ground space. This means the sphere will end up at (0, 1, -10) in world space. This is the basis of a crude form of scene graph. Look up scene graph if you are interested in finding out about hierarchical spaces.

In the on resize method we do make sure the projection looks good no matter how the screen is resized. Then in the OnRenderFrame method we do the drawing.

```
base.OnRenderFrame(e);
GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);

int uModel = GL.GetUniformLocation(mShader.ShaderProgramID, "uModel");
GL.UniformMatrix4(uModel, true, ref mGroundModel);

GL.BindVertexArray(mVAO_IDs[0]);
GL.DrawArrays(PrimitiveType.TriangleFan, 0, 4);

uModel = GL.GetUniformLocation(mShader.ShaderProgramID, "uModel");
Matrix4 m = mSphereModel * mGroundModel;
GL.UniformMatrix4(uModel, true, ref m);

GL.BindVertexArray(mVAO_IDs[1]);
GL.DrawElements(PrimitiveType.Triangles, mSphereModelUtility.Indices.Length,
DrawElementsType.UnsignedInt, 0);

GL.BindVertexArray(0);
this.SwapBuffers();
```

You should be getting pretty familiar with this sort of code by now. We clear colour and depth bits in the last frame , then we link the ground matrix to the shader, bind the ground array and draw it. Next we link the sphere matrix to the shader, bind the sphere array and draw that using a set of indices. Then we swap the buffers so that the back buffer (that we just drew onto) becomes the front buffer, on the screen, and the front buffer (that was on the screen) becomes the back buffer to draw the next frame onto. Take a brief look at the shaders. They are pretty standard at the moment.

OnUnload deletes the data we loaded on to the graphics card.

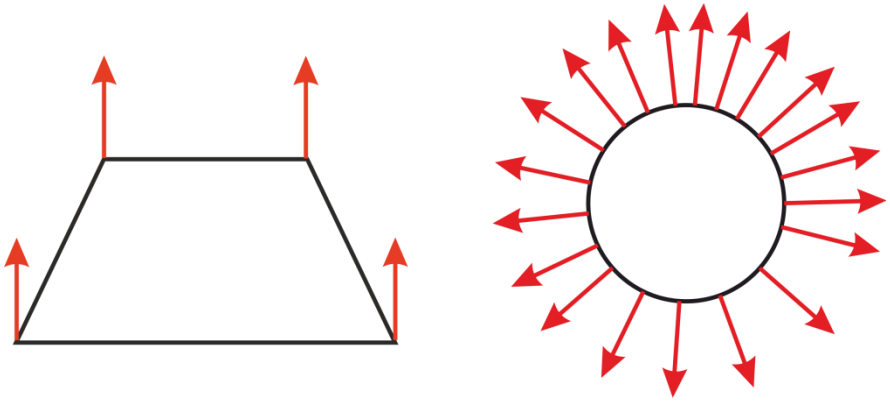
The last method to look at, because that is where we're going to do some work next, is the OnKeyPress method. Some camera movement has been added that enables you to move forward using the 'w' key

and rotate left using the ‘a’ key. Modify this code so that you can also move backwards using ‘s’ and rotate right using ‘d’.

Later, you’ll have to do update more data when you translate and rotate the camera. You might want to create separate methods for that now, and call them from the OnKeyPress function. It will save you time later! Once you’ve done that commit your code with the message

L3T2 Finished off first person style camera

If you were paying attention you will have noticed that for each vertex in the ground and the sphere we’re loading six floats, but we’re only using three. The last three floats are the normal vector. A normal vector is a vector that is perpendicular to a surface, which tells us about the curvature of that surface at that point. For the ground, the normal vectors all point straight up. For the sphere, the normal vectors all point straight out of the sphere, as shown in the diagram below. It is also important that normal vectors have a length of 1 unit.



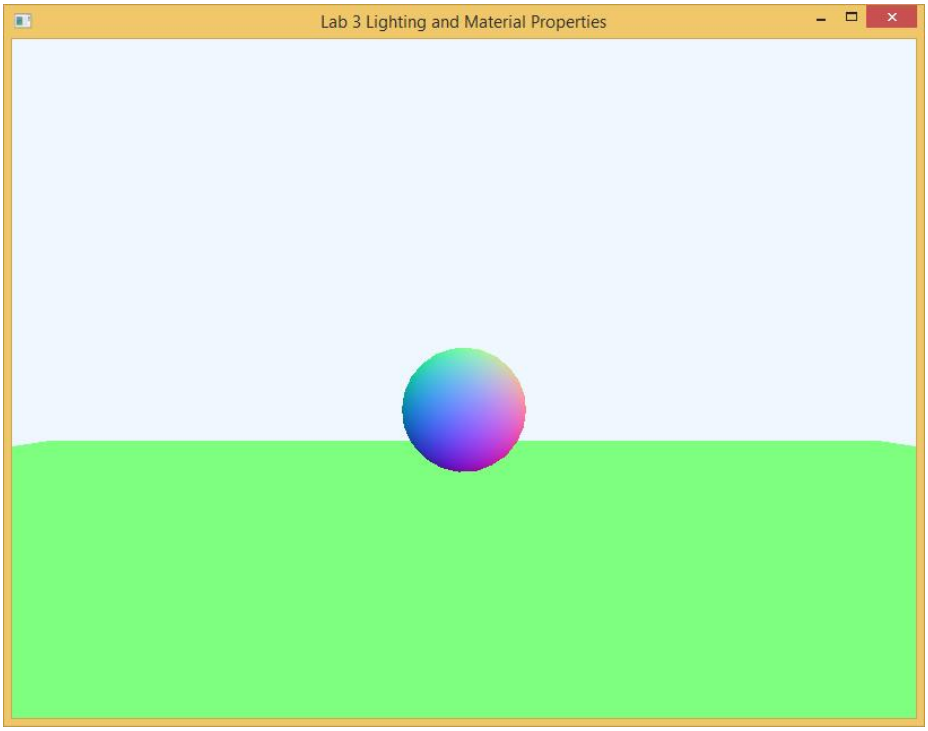
Let’s link our normal to our shader. For the time being we’ll visualize the normal vector as a colour. To do this we need to take into account that the components of normal vectors can be negative (they have values of -1 to 1), but it doesn’t make much sense for a colour to be negative. To compensate for this we will multiply the normal vector by a half (so the values will be between 0.5 and -0.5) and then we’ll add a half (so that the values will be between 0 and 1). Add a per vertex vec3 variable called vNormal, then change the lone that sets the output colour that is sent to the fragment shader to be

```
oColour = vec4(vNormal * 0.5 + 0.5, 1);
```

Notice that in glsl a number of shortcuts have been implemented that allow us to treat a vector value a bit like a float. vNormal is a vec3. When we multiply this by a scalar, the scalar is applied to all three of its components. Similarly, adding a float adds that value to all three components. This takes advantage of a convenient optimization called swizzling. Also, doing a multiply then an add is typically faster than the other way round.

Back in the OpenTK window, in the OnLoad method you need to link that per vertex vNormal attribute that you just created in the shader to the data we loaded into the buffer. We covered this in the last lab. You need to get the id of the normal location from the shader, and then set the vertex attribute for the VBO data. You also need to enable the appropriate attribute array. You need to do this for both VAOs. When you set the vertex attribute pointer you need to set the "normalized" parameter to true. This will ensure that the normal stays unit length as it moves through the pipeline.

Hint: For every vertex we are loading 6 float; position x, position y, position x, normal x, normal y, normal z. Remember that when you set the offset the value needs to be in bytes!



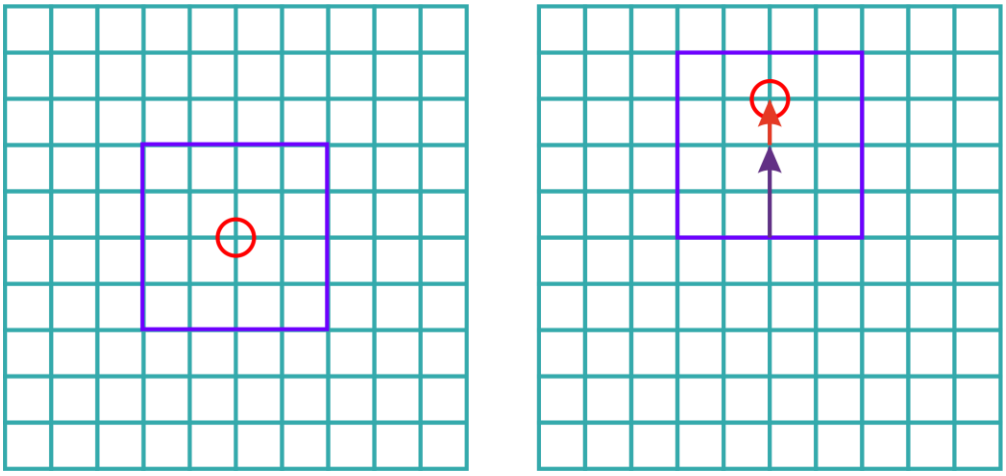
When you have managed to visualize the normal commit your code to SVN.

L3T3 Added per vertex attribute for the normal and linked VBO data to it.

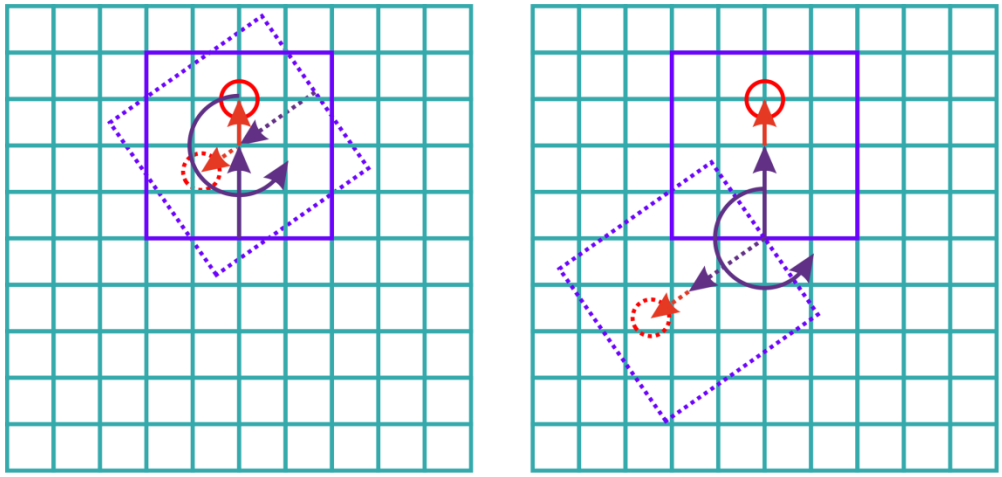
Before we start calculating any lighting it would be useful if we could rotate the ground around within the scene so that we can see the back of an object and check that it is being lit properly. In the KeyPress method make z and x rotate the ground and the sphere together. If you have a quick go at this you might try doing something like:

```
mGroundModel = mGroundModel * Matrix4.CreateRotationY(-0.025f);
```

This doesn't work (although it might look like it initially). We've seen this problem before when we were translating and rotating squares – the order matters! The diagrams below should give some insight into what is happening. Note the diagrams are for illustration purposes and are not drawn to scale.



In the diagram the turquoise grid represents “world space”. When they are first created, both the centres of the sphere and the ground square are at 0,0,0 in model space. Then we translated the ground 5 units in the z direction, and the sphere 5 units (in ground space) in the z direction.



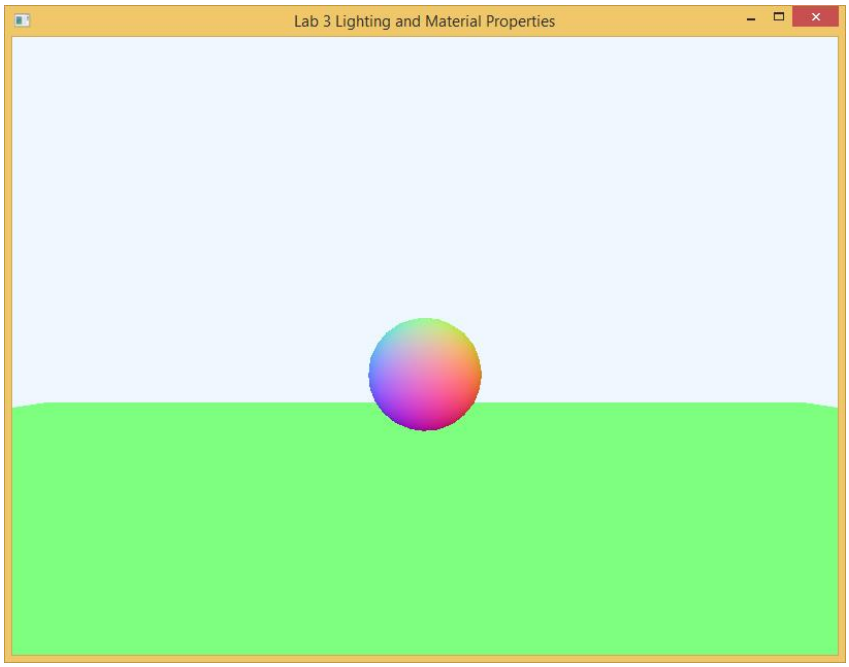
Now, when we try to rotate the ground, what we want to happen is shown on the left; we want the ground and the sphere to rotate around the centre of the ground. What actually happens is shown on the right. We rotate around the centre of the world. In order to achieve what we want we need to build a matrix that moves the square back to the centre of the world, rotates it, and then moves it back again. OpenTK’s Matrix4 class provides functionality for extracting a translation from a matrix.

```
Vector3 t = mGroundModel.ExtractTranslation();
Matrix4 translation = Matrix4.CreateTranslation(t);
Matrix4 inverseTranslation = Matrix4.CreateTranslation(-t);
mGroundModel = mGroundModel * inverseTranslation * Matrix4.CreateRotationY(-0.025f) *
translation;
```

Once you’ve done that for both directions Commit to SVN.

L3T4 Can rotate the ground and sphere in ground space (around the middle of the ground)

Next add some functionality to rotate the sphere in sphere space using the ‘c’ and ‘v’ keys (It should rotate, but it’s position shouldn’t change). By rotating the sphere you should be able to see the colours (provided by the normal data) changing. Here’s a screen shot of the back of the sphere.

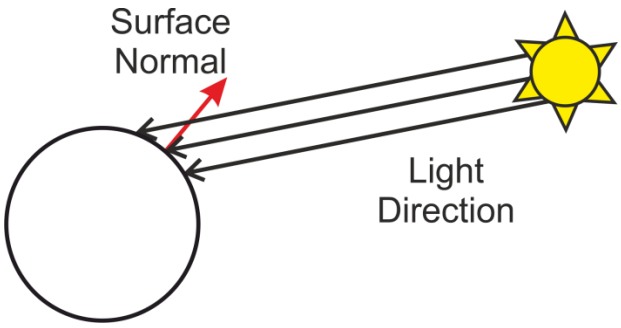


When you’ve done that, commit your code to SVN.

L3T5 Can rotate the sphere in sphere space (around the middle of the sphere)

Rotating the sphere like this is fine, but if you think carefully about if we were using the normal data to perform lighting calculations it shouldn’t have changed. The curvature of the surface I’m looking at hasn’t changed, but the colour (representing the normal) has changed. We’ll need to fix that later.

Let’s add a light and use it together with the normal to calculate the light at each vertex. To begin with, we’re going to add a directional light.



This means that we won’t take into account the lights position, because it is considered so far away, that the direction from any point in the scene is the same. This sort of light is ideal for modelling light sources that are very far away, like the sun. Initially we’re going to calculate diffuse light. Diffuse light is

proportional to the angle between the normal vector and the direction from the surface to the light source. Add a uniform vec3 variable called uLightDirection to the vertex shader, and then change the line the sets the output colour to be:

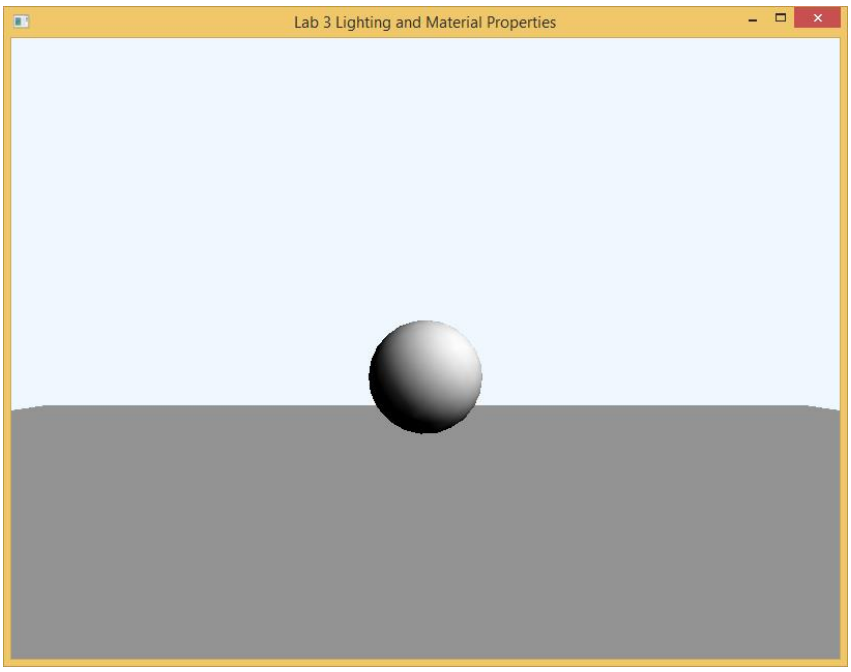
```
oColour = vec4(vec3(max(dot(vNormal, -uLightDirection), 0)), 1);
```

GLSL's built in dot function returns a float. The built in max function returns the maximum of the float from the dot product and zero. That will be used for all three values of the vector to be the same giving us a grey colour. The alpha channel is set to 1.

In the OnLoad method set the uniform light direction to be -1, -1, -1. This sets the direction of the light to be over your right shoulder.

```
int uLightDirectionLocation = GL.GetUniformLocation(mShader.ShaderProgramID,
"uLightDirection");
Vector3 normalisedLightDirection, lightDirection = new Vector3(-1, -1, -1);
Vector3.Normalize(ref lightDirection, out normalisedLightDirection);
GL.Uniform3(uLightDirectionLocation, normalisedLightDirection);
```

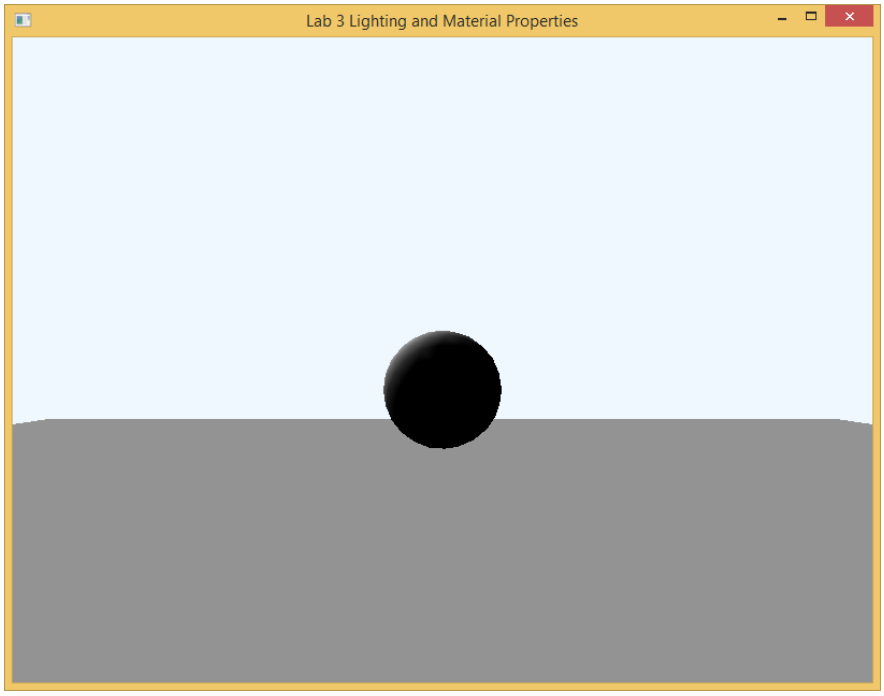
Run the code, you should see something like this.



Commit your code to SVN

L3T6 Made a directional light in the vertex shader

If you press ‘c’ or ‘v’ to rotate your sphere, the light will appear to move, but we know it’s not moving. The normal at the back of the sphere are being rotated, but we are not taking this into account in our calculations. You’ll also notice that if you move to the other side of the sphere the light will appear to move with you. In our calculations we already apply the model view projection matrix to our vertices. This calculation happens in world space, so we only need to apply the model view matrices to the normal vectors. Also, a normal is a direction. It doesn’t have a position part, so we only need to apply the rotation to the calculation.



The rotation part of a four by four matrix is the top left three by three cells. The final thing to consider is that our model view space may have scaled our model. If the model has been scaled in a non-uniform way, (i.e more in one direction than another) the curvature of our surface should be affected by the scaling. To compensate for these factors we need to multiply the normal vectors by the inverse transpose rotation of the model view matrix.

```
vec3 inverseTransposeNormal = normalize(vNormal * mat3(transpose(inverse(uModel * uView))));
oColour = vec4(vec3(max(dot(inverseTransposeNormal, -uLightDirection), 0)), 1);
```

If you rotate the sphere now you should be able to see that the sphere is rotating (by carefully looking at the edges) but the shading shouldn’t be changing.

Commit to SVN with the message

L3T7 Adjusted normal vectors by multiplying by the inverse transpose of the model view matrix

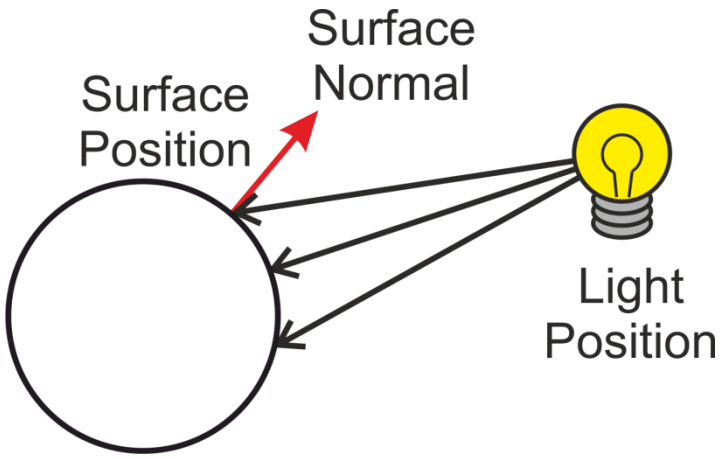
If you move the camera to the other side of the sphere you'll notice that the light still appears to have moved with you. That is because we are not taking into account the view. Remember, the view matrix moves the whole world around the camera. Like the normal, with the directional light the view only needs the rotation portion applied to it. Change the vertex shader so that the light direction is adjusted if the view is rotated.

```
vec3 inverseTransposeNormal = normalize(vNormal * mat3(transpose(inverse(uModel * uView))));
vec3 lightDir = normalize(-uLightDirection * mat3(uView));
oColour = vec4(vec3(max(dot(inverseTransposeNormal, lightDir), 0)), 1);
```

Check that you can move the camera and see the dark side of the sphere, and then commit to SVN.

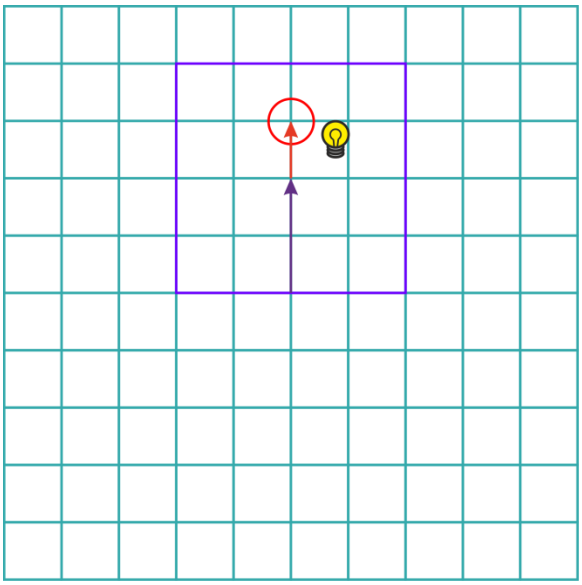
L3T8 Adjusted directional light for a rotated view matrix

The next light source that we're going to model is a point light source. This has a position in the world, which means that the direction from the surface to the light changes depending on the surface.



Now for each vertex calculation we need to work out what the light direction is relative to the vertex we are calculating.

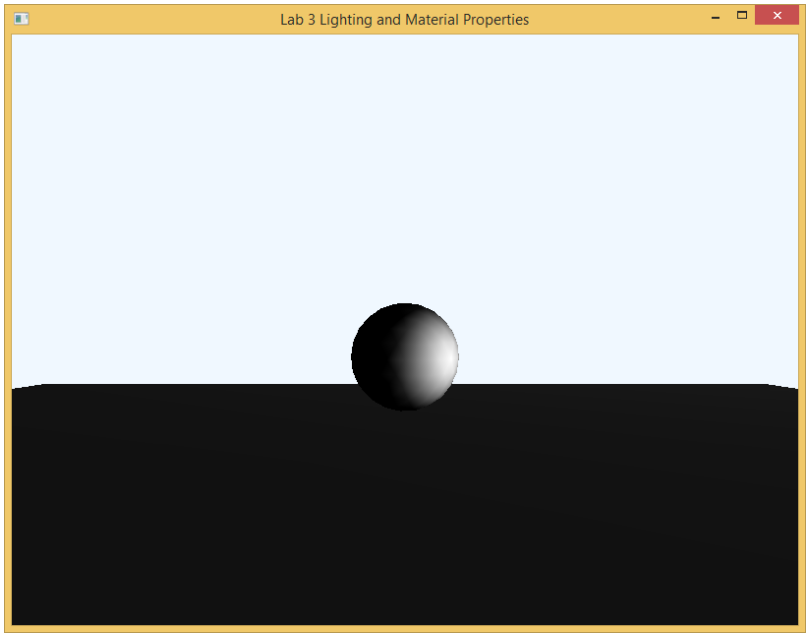
First change the uniform vec3 uLightDirection in the vertex shader to be called uLightPosition. Then in the OnLoad function set that value to be at world space position [2, 1, -8.5] This is a little bit in front and just to the right of the sphere as shown on the diagram below. There is also no need to normalize the light position. A direction should be unit length, but normalizing the light position will move it somewhere we don't want it to be.



Next in the shader after you have calculated the inverse transpose of the normal replace the rest of the method with the following code:

```
vec4 surfacePosition = vec4(vPosition, 1) * uModel * uView;
vec4 lightPosition = vec4(uLightPosition, 1) * uView;
vec4 lightDir = normalize(lightPosition - surfacePosition);
oColour = vec4(vec3(max(dot(vec4(inverseTransposeNormal, 1), lightDir), 0)), 1);
```

Take a minute to read it. What we're doing is calculating the surface position in world space, and the light position in world space, then we're calculating the vector from the surface position to the light position and making that a unit length. Finally we're doing the dot product of the inverse transpose normal and the light direction. Again the max function returns the maximum of the dot product and zero, to eliminate any "negative light". It might sound a little complicated, but it's only a little different to what we did for the directional light.



When you get this working submit to SVN with the message.

L3T9 Implemented positional light

Note the poor shading on the ground. That is because the colour value is calculated at the vertices, which are a long way away from the light. The colours that are calculated are then interpolated across all the fragments in the triangle and the whole triangle appears dark. There are two ways to address this. One is add more vertices to our ground, and the other is to do the lighting calculations in the fragment shader. Let's do the latter.

In the Lab 3 folder, right click on the Shaders folder and add a new item. Select text file and call it fLighting.frag. Do the same again and create a vertex shader called vPassThrough.vert.

In the vertex shader we need pass out the inverse transpose normal (oNormal) and the world space position (oSurfacePosition). To do that we still need to know about the model view and projection matrices, but everything else we can do in the fragment shader.

```
#version 330

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProjection;

in vec3 vPosition;
in vec3 vNormal;

out vec4 oNormal;
out vec4 oSurfacePosition;

void main()
{
        gl_Position = vec4(vPosition, 1) * uModel * uView * uProjection;
        oSurfacePosition = vec4(vPosition, 1) * uModel * uView;
        oNormal = vec4(normalize(vNormal * mat3(transpose(inverse(uModel * uView)))), 1);
}
```

In the fragment shader, we just need to do the same calculation that we did before to find out the light direction, and from that the fragment colour.

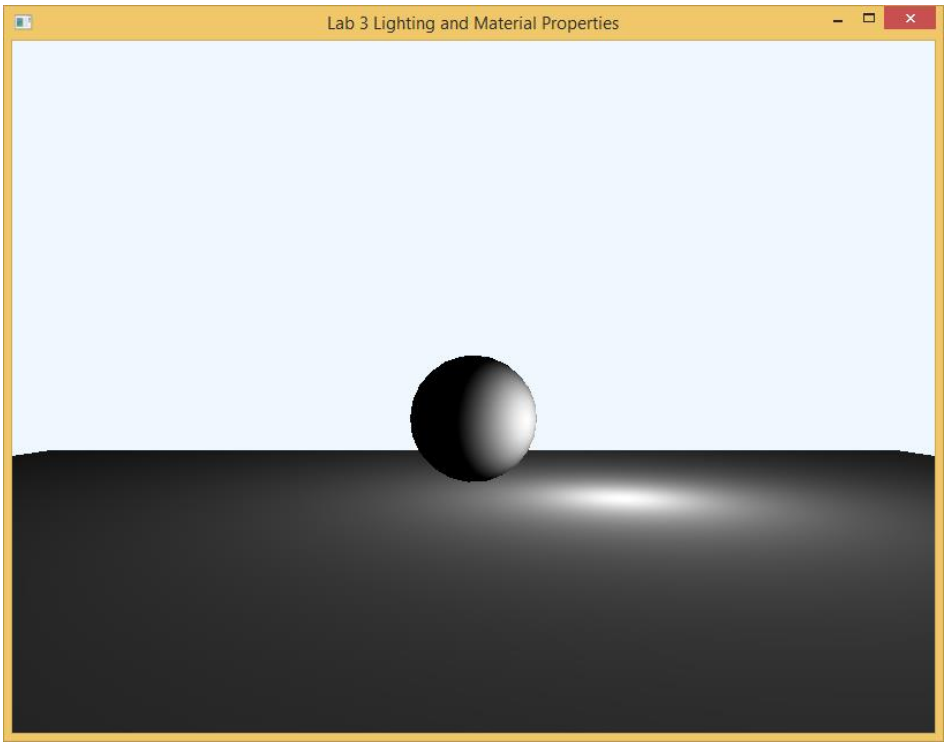
```
#version 330

uniform vec4 uLightPosition;

in vec4 oNormal;
in vec4 oSurfacePosition;

out vec4 FragColour;

void main()
{
        vec4 lightDir = normalize(uLightPosition - oSurfacePosition);
        float diffuseFactor = max(dot(oNormal, lightDir), 0);
        FragColour = vec4(vec3(diffuseFactor), 1);
}
```

Note that we're no longer changing the light position to the world view. Instead, whenever we move the camera we'll have to change the uLightPosition from the main program to compensate. This is where refactoring those camera movements could come in handy – wherever you move the camera update the new light position in camera space. OpenTK can help you transform a vector by a matrix using the Vector4.Transform method.

```
Vector4 lightPosition = Vector4.Transform(new Vector4(2, 1, -8.5f, 1), mView);
```

You still need to pass this value to the shader, whist we need to adapt our old code to pass in a vec4 for the light position, instead of a vec3.

Finally don't forget to load the new vertex and fragment shader into the shader object, instead of the old ones, and to set the new vertex and fragment shader to copy to the output directory when the solution is built.

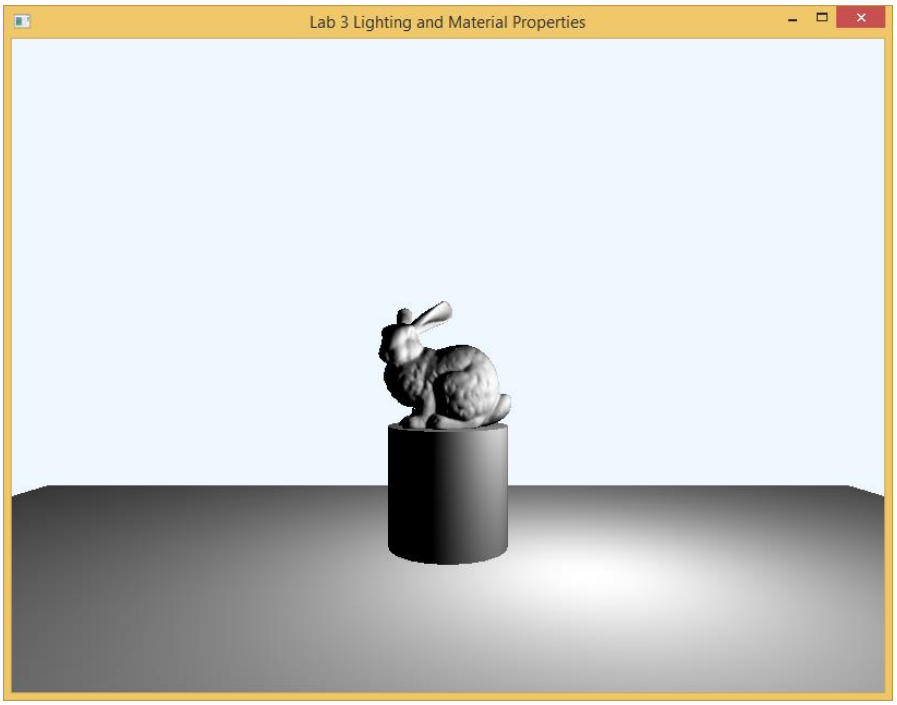


Notice the nice new floor! Don't be fooled into thinking that the light on the floor is behind the light on the sphere. This is an optical illusion due to the lack of shadows. Remember, the sphere is **touching** the floor, and not hovering over it. Make sure that your light stays in the right place when you move the camera, and when you rotate the sphere. Is *should* appear to stay stationary if you move the ground under it. Commit your code to SVN with the message

L3T10 Moved point lighting into the fragment shader

Looking at a simple sphere is a bit boring. Instead, there is a model.bin file you can use with a surprise model. Mine was the Stanford bunny, but there's a good chance that you'll have something different. There's also a cylinder.bin model which we can use as a pedestal. Load the bunny and the cylinder (you'll have to do quite a lot of work setting up new VBOs and VAOs as well as rendering, but you've done that a couple of times before now. If you get stuck check back in some previous labs, or ask for help!).

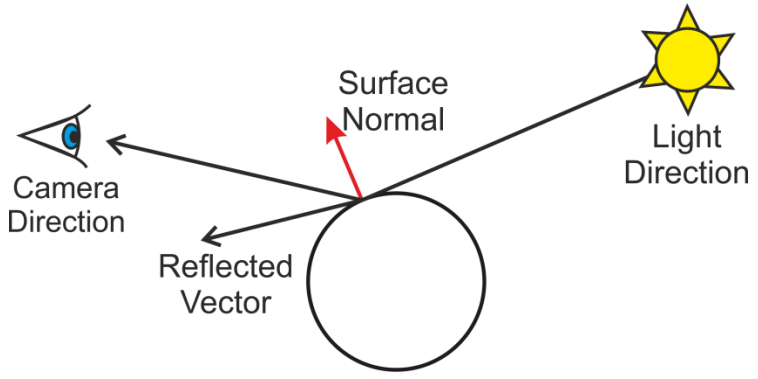
Credit for the models goes to the Stanford 3D Scanning Repository or the Czech Technical University. Please be aware that several of these models are scanned from artifacts that have religious and cultural significance. Please treat these models with respect.



Once you've managed to load your model, and your cylinder, and positioned/rotated your model into place, and got your camera into a decent place, and your light (quite a lot of tinkering) commit your code to SVN filling in the model. One method is to translate the model in cylinder space and apply both matrices when you render.

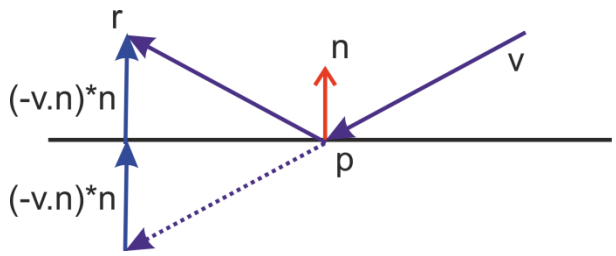
L3T11 Loaded my model – My model is a < insert model here! >

So far we've only considered a type of light-surface interaction called diffuse light. In the classical lighting models there are two other types of light-surface interactions. Ambient light, which is a constant "background" light that doesn't come from anywhere in particular, and specular light, which takes into account both the light direction and the eye position relative to the surface position. See the diagram below.



In the fragment shader we need to know what the camera position is in world space. Create a uniform vec4 uEyePosition. uEyePosition should be the camera position in world space. Set it from the main program and remember to update it whenever the camera is moved.

The reflected vector can be calculated using the cosine rule. See the diagram below.



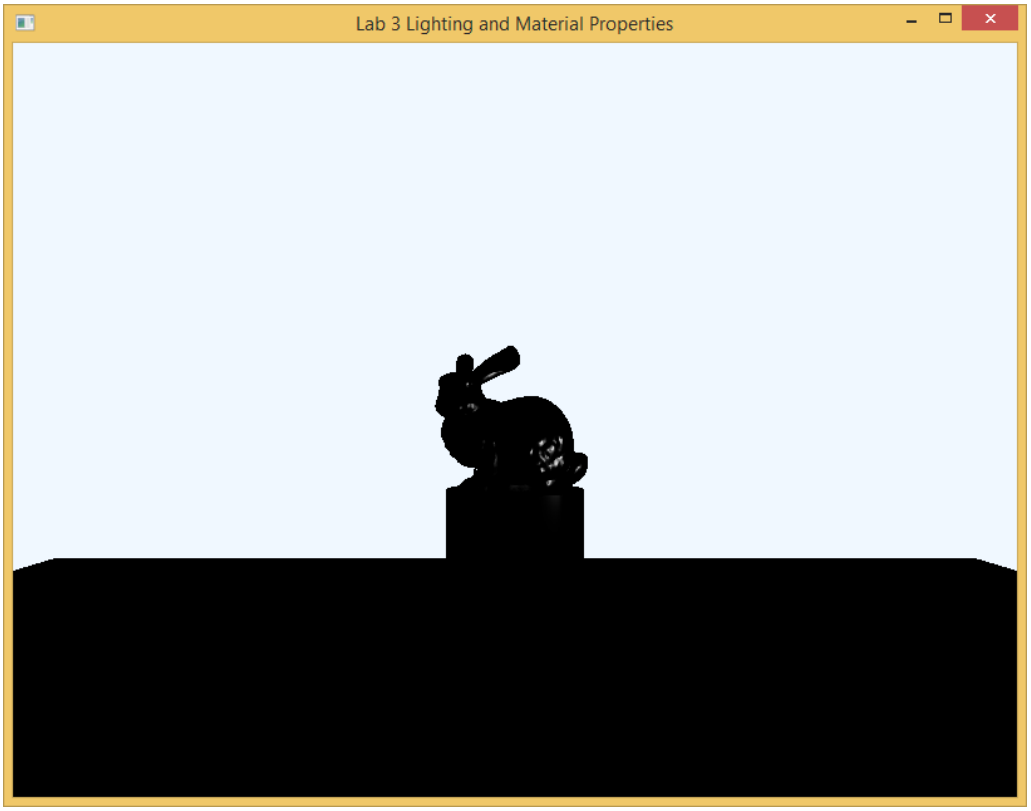
The reflection of incoming vector v is v + 2 * (-v.n) * n (using the cosine rule to calculate the adjacent side of the triangle formed by v and n). This is useful to know, because we can use it later when we calculate physics, but GLSL also provides a built in reflect method for us.

The specular component of light is calculated using the dot product of the reflected vector r, and the vector from the surface to the eye. This is increased to some power (that models shininess).

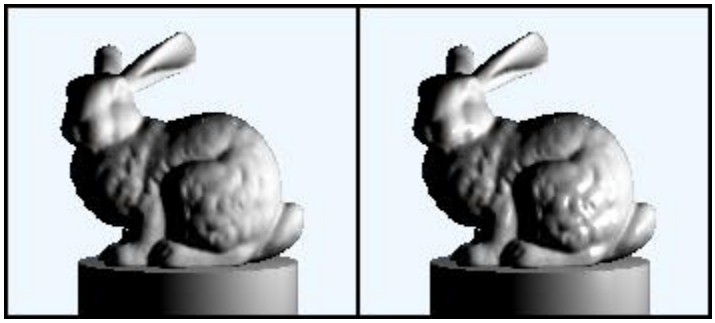
```
vec4 lightDir = normalize(uLightPosition - oSurfacePosition);
vec4 eyeDirection = normalize(uEyePosition - oSurfacePosition);
vec4 reflectedVector = reflect(-lightDir, oNormal);

float specularFactor = pow(max(dot( reflectedVector, eyeDirection), 0.0), 30);
```

In the fragment shader the specular factor is calculated as shown above. Note that this is really just an expression of the diagram showing the reflected vector. Also note the built in functions provided by GLSL. Dot returns the dot product of two vectors, max returns the maximum of two numbers and pow increase the its first parameter to the power of its second parameter. In this case, that second parameter is the shininess, and it has been set to 30.

If you set the program to just use the specular calculation to colour our pixels you can get a good idea of that the specular lighting gives us. It gives us the shiny, specular highlights on object as shown below.



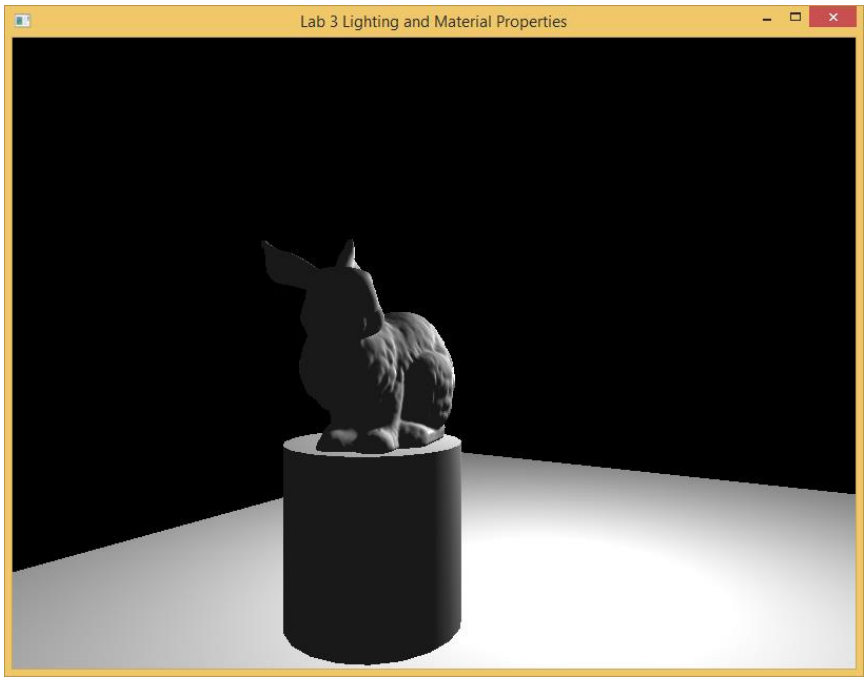
When the model is finished all the lighting terms should be added together. In the diagram below, on the left only the diffuse term is being used, and on the right only the diffuse and specular terms are added together.



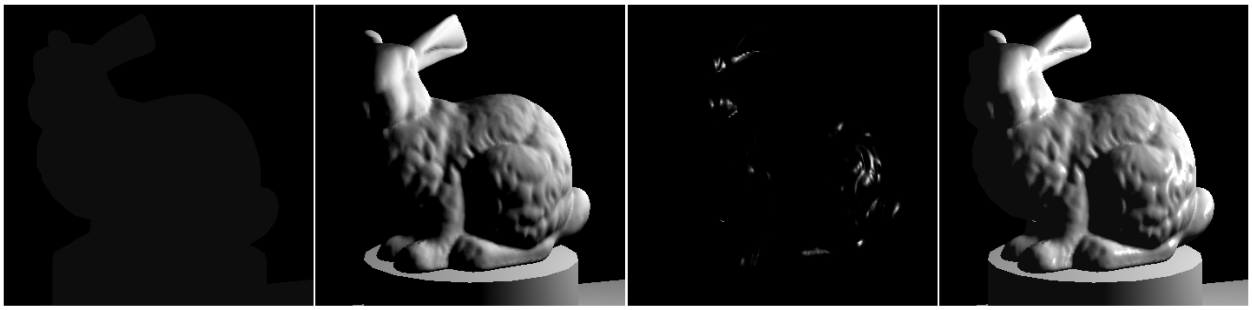
Once you've got your specular term working commit to SVN.

L3T12 Added specular light to lighting model

The final term (although usually the first term to be introduced) is the ambient term. This is a constant term that ensures that some light gets everywhere. In the diagram below the ambient term has been set relatively high (also, the clear colour has been changed for better contrast.



The diagram below shows the three different components, as well as the combined model.



Commit your code to SVN.

L3T13 Added ambient light to the model – now have ambient, diffuse and specular light

So far, we've developed a lighting model that can handle ambient, diffuse and specular light surface interactions, but everything we've made is grey. Next we'll add some colour using coloured light and material properties. It's not as complicated as you might think. For every type of light surface interaction we've modelled so far, what we can do is specify a colour to represent the colour of the incoming light, and another colour to represent the reflectivity of the surface. For each term in our lighting equation we multiple the calculated term with the appropriate light colour and reflectivity colour.

To keep this simple we'll add a struct to our shader for our light, and another for our material property.

In the fragment shader remove the uLightPosition variable, and outside of the main function add the following code.

```
struct LightProperties {
        vec4 Position;
        vec3 AmbientLight;
        vec3 DiffuseLight;
        vec3 SpecularLight;
};

uniform LightProperties uLight;

struct MaterialProperties {
        vec3 AmbientReflectivity;
        vec3 DiffuseReflectivity;
        vec3 SpecularReflectivity;
        float Shininess;
};

uniform MaterialProperties uMaterial;
```

This defines two structs, one to deal with light properties, and one to deal with material properties. We also create a uniform variable for each, which must be set from the main program. Do that next.

To set a uniform variable that is a struct is similar to setting a normal uniform variable – here are some examples to get you going.

```
int uLightPositionLocation = GL.GetUniformLocation(mShader.ShaderProgramID,
"uLight.Position");
Vector4 lightPosition = new Vector4(2, 4, -8.5f, 1);
lightPosition = Vector4.Transform(lightPosition, mView);
GL.Uniform4(uLightPositionLocation, lightPosition);

int uAmbientLightLocation = GL.GetUniformLocation(mShader.ShaderProgramID,
"uLight.AmbientLight");
Vector3 colour = new Vector3(1.0f, 1.0f, 1.0f);
GL.Uniform3(uAmbientLightLocation, colour);
```

You need to do this for all elements of the light struct and the material struct. You will also need to remember to change the way you update the eye position when you move the camera. Here, we are setting the material property in the OnLoad function, but you may want to do that between draw calls, so you can use different materials for the ground, the cylinder and the model. You may also want to save yourself some effort and create a method to help you do this.

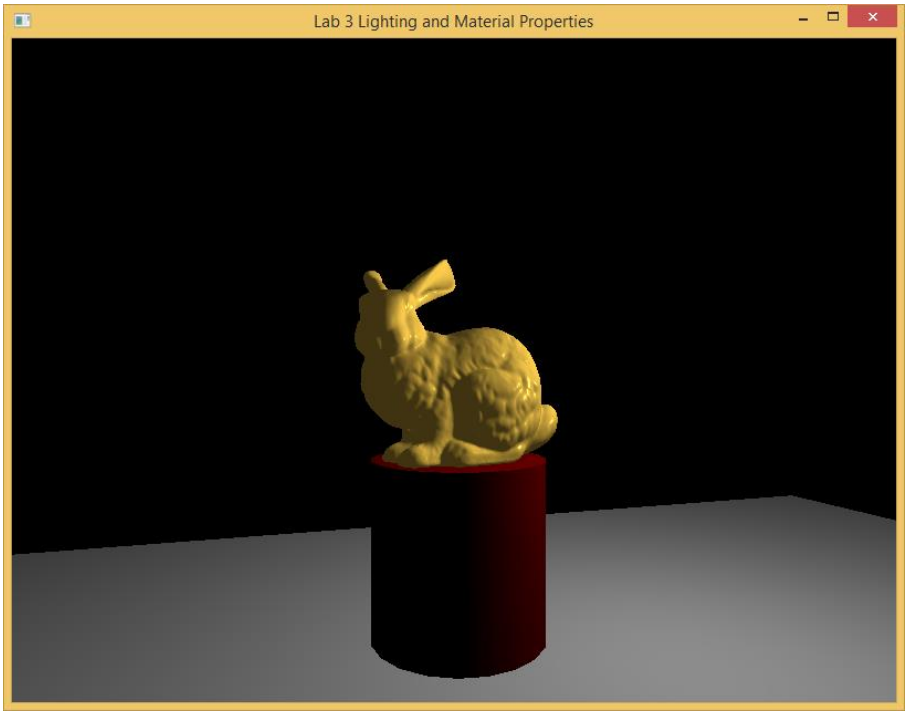
In the fragment shader we need to make a minor adjustment to the specularFactor so that it uses the material's shininess property, but the only other change is to the line that calculates the frag colour. We need to adjust it to scale all the terms by the appropriate colours depending on the light and the material. You also need to update the light position to use the position from the instance of the struct.

```
float specularFactor = pow(max(dot( reflectedVector, eyeDirection), 0.0),
uMaterial.Shininess);

FragColour = vec4(uLight.AmbientLight * uMaterial.AmbientReflectivity +
        uLight.DiffuseLight * uMaterial.DiffuseReflectivity * diffuseFactor +
        uLight.SpecularLight * uMaterial.SpecularReflectivity * specularFactor,  1);
```

Note how the uLight and uMaterial variables are being used with the calculated factors.

There is a table of realistic looking material properties here:

http://devernay.free.fr/cours/opengl/materials.html



Check out my gold bunny on a red plastic cylinder, with a white rubber ground. Once you've built your scene, commit your code to SVN with the following (completed) commit message.

L3T14 I made a <material> <model> on a <material> cylinder, with a <material> ground

What if you have more than one light source though? Next we're going to create a scene with three lights. I've changed my material s to white rubber, and I'm going to add a red, a green and a blue light around the model.

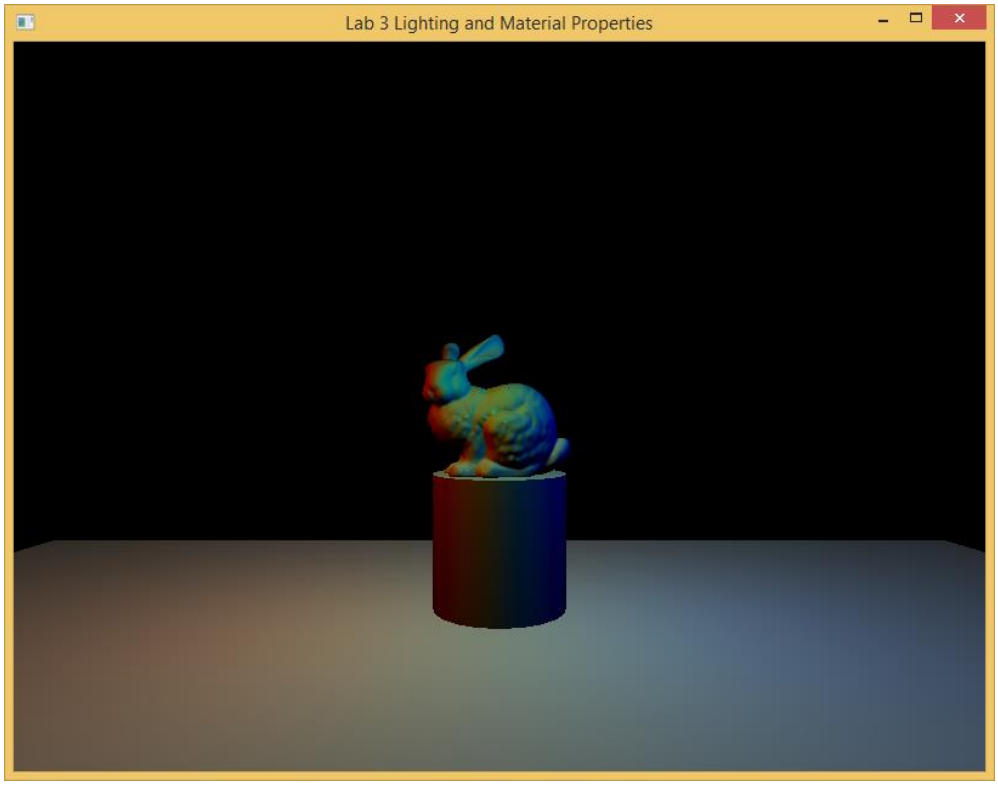
It's easier than you might imagine. The fragment shader supports arrays. In the fragment shader change the instance of the light struct to be an array. Then you can set each of the light's properties from the main program like this. You will need to do this for each property for each light!

```
int uAmbientLightLocation = GL.GetUniformLocation(mShader.ShaderProgramID,
"uLight[0].AmbientLight");
Vector3 colour = new Vector3(0.1f, 0.0f, 0.0f);
GL.Uniform3(uAmbientLightLocation, colour);
```

Finally, in the main body of the fragment shader you need to get light components for each light. Note that with the exception of the eyeDirection everything in the shader is dependent on the light.

```
for(int i = 0; i < 3; ++i)
{
        vec4 lightDir = normalize(uLight[i].Position - oSurfacePosition);
        vec4 reflectedVector = reflect(-lightDir, oNormal);
        float diffuseFactor = max(dot(oNormal, lightDir), 0);
        float specularFactor = pow(max(dot( reflectedVector, eyeDirection), 0.0),
uMaterial.Shininess);
        FragColour = FragColour + vec4(uLight[i].AmbientLight *
uMaterial.AmbientReflectivity + uLight[i].DiffuseLight * uMaterial.DiffuseReflectivity *
diffuseFactor + uLight[i].SpecularLight * uMaterial.SpecularReflectivity * specularFactor,
1);
}
```

As you can imagine this is quite a hit on performance. When this is compiled the for loop *should* be unrolled, but just in case you might want to unroll it manually, if you know that there are only going to be three lights. (If you don't and the number of lights varies then you're probably looking at an even **bigger** hit on performance!).

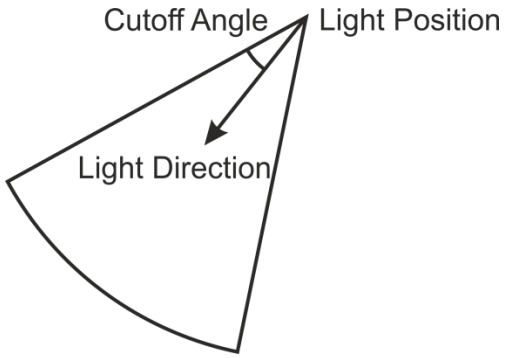


Once you've managed use multiple lights commit your code to SVN

L3T15 Calculating colour fragments from three lights at a time

The final tasks in this lab are optional extras. Although they should help solidify your understanding, so they are worthwhile and will help you with your assessment.

By now you should see the massive flexibility that programmable shaders give us. The first task is to use what you've learnt about shaders to convert our lights into spotlights. Here's a diagram to help you visualize what you might need:



You'll need to add some new variables to your light sources (and populate them with values). You'll also need to apply some trigonometry to find out if the angle between the vector for the light and the surface fragment is less that the cut off angle.

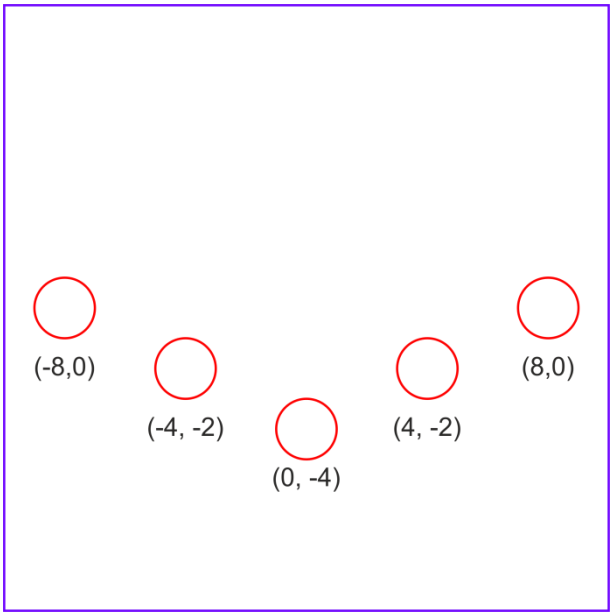
Another enhancements for any of the lighting we modelled so far is to add attenuation (less light reaching surfaces that are far away from the source).

If you complete this task commit your code to SVN with the (completed) message

L3T16 I made spotlights! I did it by…

As I said before, I used a bunny for my model. You might have a different model. There are five models in total. Collect all the models from your course mates, and load them into one scene. You can use them to practice arranging different models in 3D space.

I suggest an arrangement that looks something like this from above.



You don't need to do this to progress to the next lab – it's just for fun (and to check your understanding). When you've collected all the models commit your code to SVN.

L3T17 I collected all the models and rendered them in one scene. My favourite is the <favourite model>!

If you like, you can share what you have made on the forum. There is a thread here for screenshots of your collection of models here:

http://forums.net.dcs.hull.ac.uk/p/1945/6527.aspx

Summary

In this lab we've covered a lot of material. We've practiced rotating and translating models. We've learnt about normal vectors, and how we need to treat them differently when we rotate, scale and translate them, and we've built a shader based lighting model with different types of lights and material properties, and we're starting to see how powerful programmable shaders are.