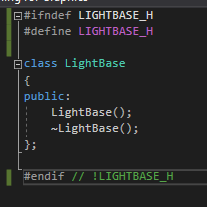
# Lights

## Disclaimer

This is the first time I have written lighting in OpenGl, the document jumps around a bit and as such I might have missed a variable or forgotten something. If I have, please contact me and I’ll fix it ASAP.

## Creating the light base class

Start by crating a new class for our light, I’m calling mine “lightbase” as I might want to extend this in to other light types in the future.



We need to add a transform, as our camera needs a position in space to occupy, so add a transform object to the light.

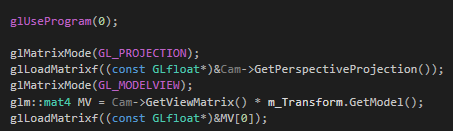


For debug reasons, I want to be able to see where my light is, so I’m going to define a Draw function and use it to draw some debug lines (a gizmo) at the position of the light.

So, define a Draw function the take a camera as a parameter, we’ll be needing the perspective matrix from it, so that our debug geometry looks right.



In the cpp, in the draw function, enter the following code.

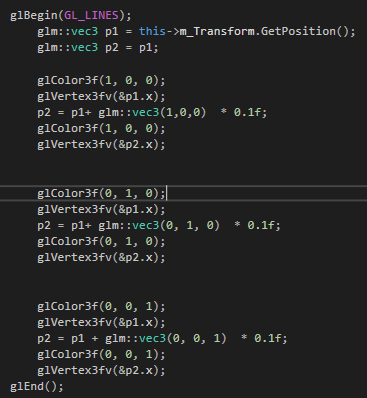


This code is, technically, for openGL 1 onwards. So, we don’t want to use this all the time, it’s slow.

OpenGL 1 and 2 were built around a state machine architecture. That means that to make it work, we must change some known variables and tell openGL to draw. The changing of the variables will change the state of openGL and thus we will see a different result on the screen.

* The first line tells openGL not to use a shader program, we may still be referencing the last one we called.
* The 2nd line tells openGL that we want to work with the variable for projection. We’re about to send some date to it.
* The 3rd line, is telling openGL to load our perspective projection in to the matrix we activated in the line above.
* The 4th line is activating the model view matrix
* The 5th line is multiplying our separate model and view matrices together
* And the 6th line is sending the combined model and view over to the activated model view matric from line 4.

Under that we will add the following code.



glBegin is how we tell old openGL that we want it to do something.

We then create 2 new vec3’s. one for the centre position of our object and one that we will reuse for the end point for each of our gizmo lines.

glColor() takes 3 parameters, R G and B glVertex3fv takes one parameters, a point in space. This will be our starting position for our first line.

We then set p2 = p1 + vec3(1,0,0) \*0.1f. this starts a point at our objects centre and off sets it by one in x, we then multiply it by 0.1 to scale it down.

Again, we call glColor and glVertex3fv and pass it p2. This is the end of our first line. openGL will draw us a line to connect these 2 points.

We do the same for the y and z directions and call glEnd() to tell openGL that were finished.

Include the lightbase in the main cpp and create a new light object.





In the main loop, call the lights draw function.



Run your code and you should see the following:



This is a simple gizmo, to tell us where the light is, currently, it is located at 0,0,0.

## Lighting objects

Now that we know where our light is, we need to start on the hard part, using the light to light and shade our objects.

First thing we ‘re going to do I add a vec3 M\_Color to our lightbase.h and set it to 1,1,1 in the constructor.





And that’s our light done!... what?.... you thought there was more to it?

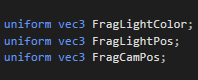
Well your right! But the rest happens in the shader.

### Shader update

In order to light our objects, we need to know the location of the light, the colour of the light and… well… that’s it. The actual shading of our objects happens on a per object basis, in the shader code.

So we now need to update our shader to take a vec3 for the light position, a vec3 for our camera position and a vec3 for the colour.

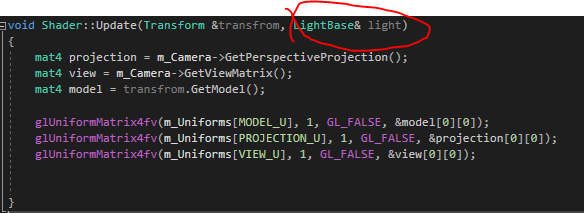
Define these 3 uniforms in the frag shader file:



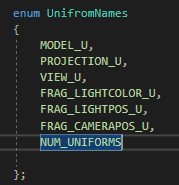
uniforms don’t have to start in the vertex file, we can define them in the fragment file as well. openGL will find the one we need. I have started mine off with the word Frag so that I know where I should be looking for them.

Now we need to get some data over to these new uniform variables. So, lets alter our shader update function to accept a light reference.

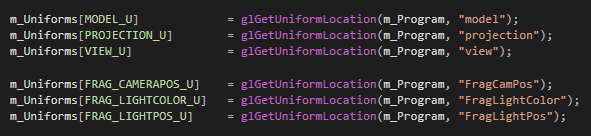




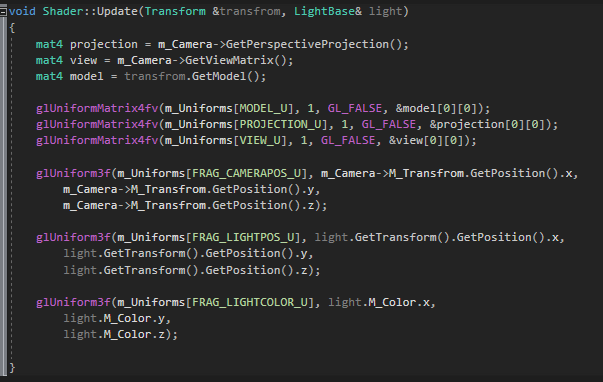
Now we are going to add 3 new uniform variables to the uniformNames enum:



In the shader constructor, we will use these to find our new frag uniforms.



And now we need to update them in the update function.



Ok, this should get our data over to our shader (don’t forget to update the call to shader.update in the main loop).

### Calculating vertex normal

In order to calculate the secularity of a surface, we need to know the normal of the surface.

The normal of a surface is a vector, perpendicular to the surface its self.

To calculate this value, we will need to use the following:

Making the following vectors:

The normal of the surface, defines above, is:

P2

P3

P1

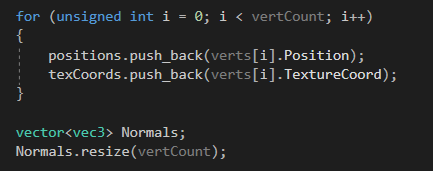
OR

We can cheat and have glm calculate the normal for us.

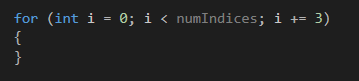
We can use the function triangleNormal(), which takes 3 vector (to define the surface).

We will need a normal for each vertex in our mesh

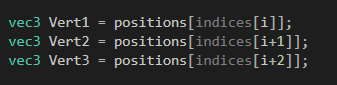
So go to your mesh constructor and add the following, under the for loop that pushes positions and texture coordinates in to their own loop



Now define the following loop:

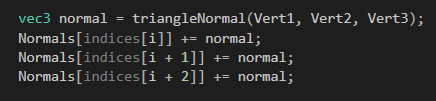


In this loop, define 3 vec3’s. we’ll need these to hold references to the verts we want to use in our normal calculations.



Notice that I’m counting up in 3 in the loop and then using ‘I’ ‘i+1’ and ‘i+2’ to get values from the index buffer and pass those index values directly in to the positions vector to get the vert information. Since we are working with planes, or faces, of a mesh, we need to grab the data that we use to define those faces. That would be 3 vertices.

Now we need to calculate the normal and set the normal vec3, which corresponds the vertex we want to ship it with.



Don’t forget to include the appropriate header file.

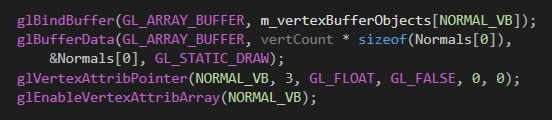


Now we need to send it to the vertex shader.

Add to the following to the VertexBuffers enum:



Bind the buffer in to mesh constructor:



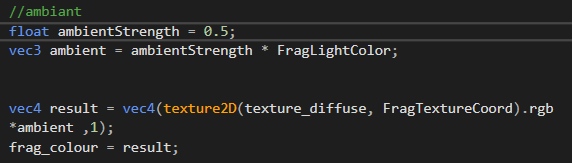
And add the following to the vertex shader, just under the ‘in’ for the texture coordinates:



Ok, we have normal data in our vertex shader.

### Ambient lighting

In the fragment shader, add the following:



If you run the program now, you should see something like this.



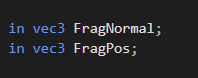
Change the ambient strength to 0.1. we only want a little bit of ambient light.

We have not yet defined a variable to allow use to control the ambient strength from the light, or even globally in the program. I’m not going to do this as its trivial at this point, but you can if you want to.

### Diffuse

Let’s start in the fragment shader.

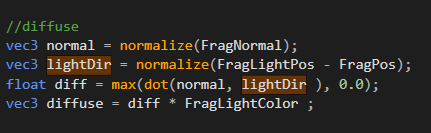
We need two in vec3’s to get data from the vertex shader.



FragNormal is going to be used to send the surface normal from the vertex shader to the fragment shader.

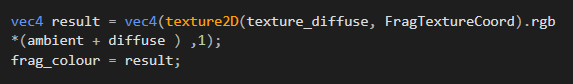
FragPos is going to be the position of this fragment in model space (think or it as, “which fragment, on the surface”.

The code for calculating diffuse light is as followed:

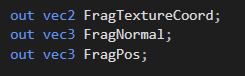


Note that our code, uses the max of Dot Normal, light and 0. This is to prevent the Dot product from return -values and messing up out lighting colour.

Now we have the diffuse light, we need to add it to the result colour.



Now we need to get the FragNormal and FragPos data from the vertex shader in to the fragment shader. Go to the vertex shader. Add 2 new out vec3’s under the FragTextureCoords



In the vertex main, set them like so:



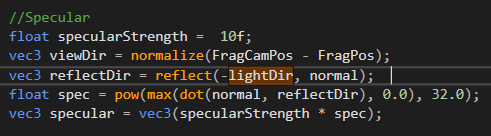
FragPos is the position on the surface and FragNormal is just the surface normal.

If you run the code now, you should see the following.



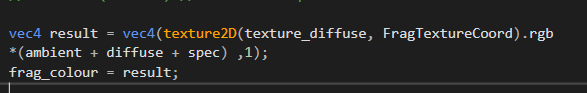
### Specular

Calculating the specular is easy, now that we have all the data we need from the ambient and diffuse.



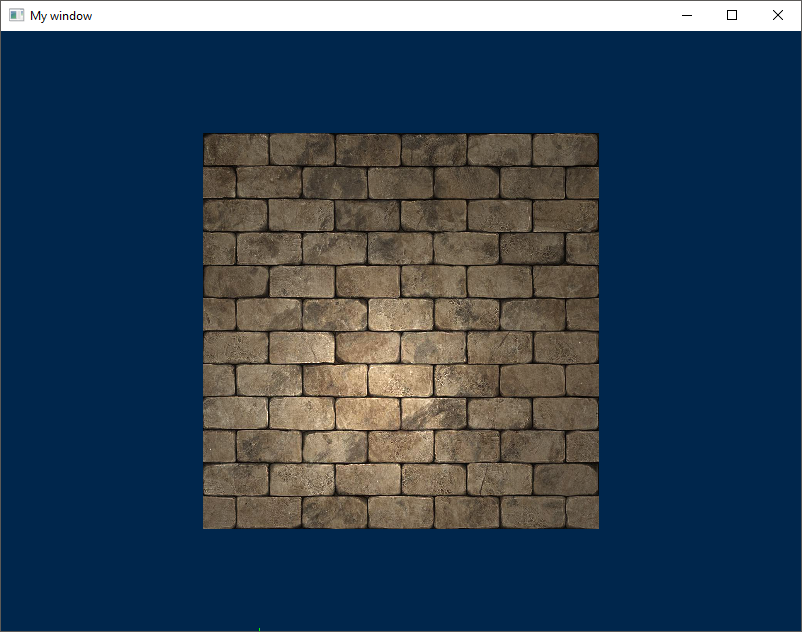
Note the negation of the lightDir. The reflect function expects the light direction to be FROM the light to the surface. Positive lightDir is from the surface to the light so we must negate it.

We combine it with the ambient and diffuse in the same way.



## Conclusion

The result should look something like this.



## Further work

Can you implement multiple lights?

The equations in the slides should give you some idea of how to do this.