

open gl 3d renderer

By Bradley Pearce

Games Programming Level I: Programming Graphic For Games Assignment 1 2016

Contents

**Page 1 – Project Overview**

**Page 1 – Research**

**Page 3 – Implementation**

**Page 4 – Analysis and Conclusion**

**Page 5 – References**

**USER GUIDE:**

**To load a texture for use in the renderer, simply replace the current string in the constructor of main’s texture object and change it to: “./res/(filename.(filetype))”**

**To load a new mesh into our scene, simply create a new Mesh object in main and as a parameter, insert “./res/(filename.obj)”**

**To change the rate of spin on the model, increase the value we add to counter each frame. Note that this should be a small value so the model doesn’t spin to rapidly, the default is 0.16.**

**Then just run the program and it should appear on screen.**

## Project overview

The purpose of this project is to read OBJ files and to render the corresponding 3D object in real time, using the data provided by the file loaded. The API we will be utilizing is OpenGL which handles sending data to the graphics card and display our graphics to the screen. In addition to these requirements, there is room to experiment with the features of OpenGL which include but not limited to:

* lighting effects and multiple types of light
* advanced material effects
* texturing
* normal mapping
* image based lighting (incl. HDR)
* interactive GUI
* Deferred Rendering
* Shadow Mapping
* Tessellation
* Modelling Tools

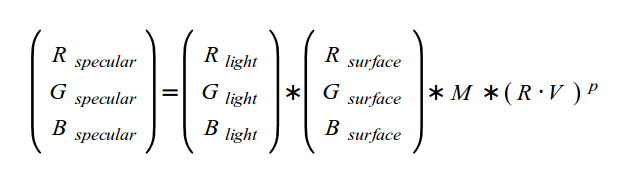
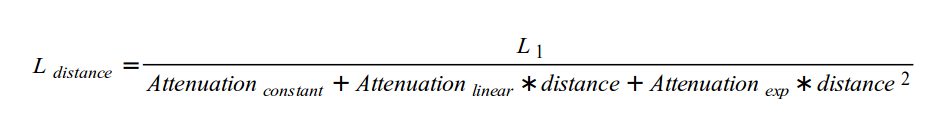
This report will detail the development of this application, along with research into mathematical solutions for the project requirements along with any extensions I can add to the project before the imposed deadline.

## Research

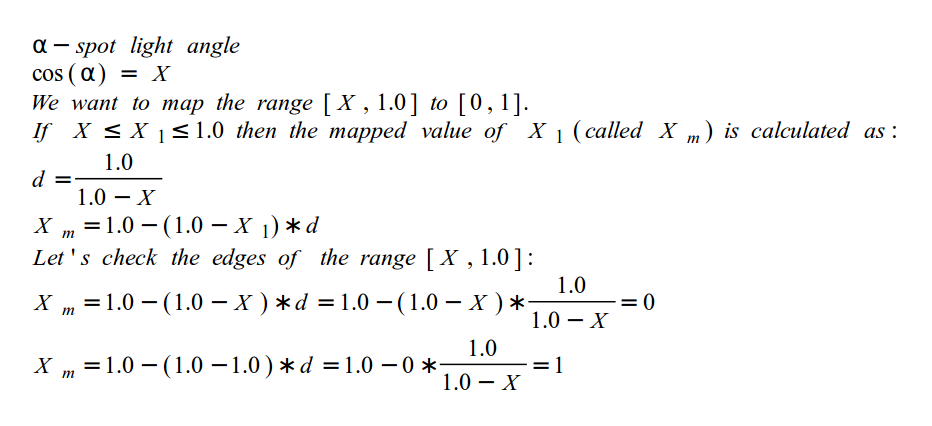
The challenges of this project we approached by research mathematical solutions which are reliable and ideally provide the most efficient solutions the problems laid out in the specification. Lots of research went into making this project a reality however for the purposes of the report I will highlight only the most challenging and notable research.

To accelerate my production, I relied on various educational resources found on line. The best solutions I applied for the purposed of my project were The Benny Box (source: <https://www.youtube.com/watch?v=ftiKrP3gW3k&list=PLEETnX-uPtBXT9T-hD0Bj31DSnwio-ywh>) and OpenGL step by step (source: <http://ogldev.atspace.co.uk/>). The Benny box’s implementation focuses on neatly arranging the solution into various modular classes, individually handling our requirements. Overall, the solution provided excellent support for creating the base requirements of the project. Then using the further material provided by ogldev.atspace.co.uk, I was able to further extend the features of the implementation by building on top of the foundation the Benny Box’s code provided.

A particularly difficult technical problem encountered was creating a blinn-phong shading model which supported specular highlights. In order to facilitate this, we needed to input an incoming light and multiply this by our surface colour, multiplied by our specular intensity and then multiplied by the cosine of the angle between the reflected light and the eye (to the power of specular power). Using these calculations in our fragment shaders were an efficient solution to creating more realistic looking materials which reflect light.



Another problem encountered was applying attenuation effects for our spot and point lights. Point lights don’t appear realistic if their light extends infinitely, in order to have the light’s colour dim over time is to divide the resulting light colour (taking into account our material attributes for our models) by our overall attenuation. The above diagram demonstrates three different types of attenuation, constant (where the light doesn’t drop off), linear (the attenuation factor multiplied by the distance from the vertex’s world position) and exponential (the attenuation factor multiplied by the distance from our vertex’s world squared). For our project we created variables to represent all three of our attenuation types and we apply an overall attenuation to our point light’s colour.

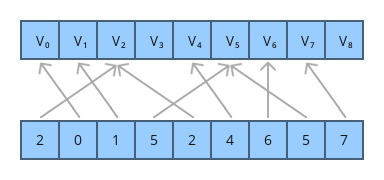


For spot lights we need to represent a cone of light using a cutoff angle and a direction. Using the cosine of the angle between vertex world position and our light’s direction. Then comparing this to our cutoff angle. If the cosine is smaller it means the angle between the light’s direction and the vertex world position is outside the spot lights circle of influence and set the colour contribution to zero. If it’s larger than our cutoff angle, we calculate the colour using our point calculation (for an attenuation effect) and plug our colour value multiplied by the ratio of the cosine of the angle divided by the ration of our light’s cutoff angle. Giving us a cone of light.

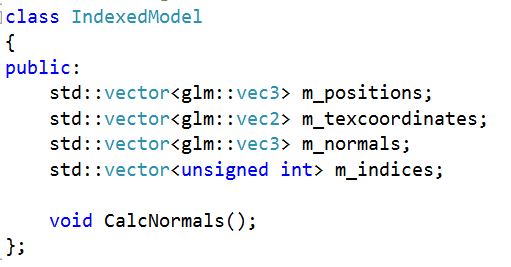
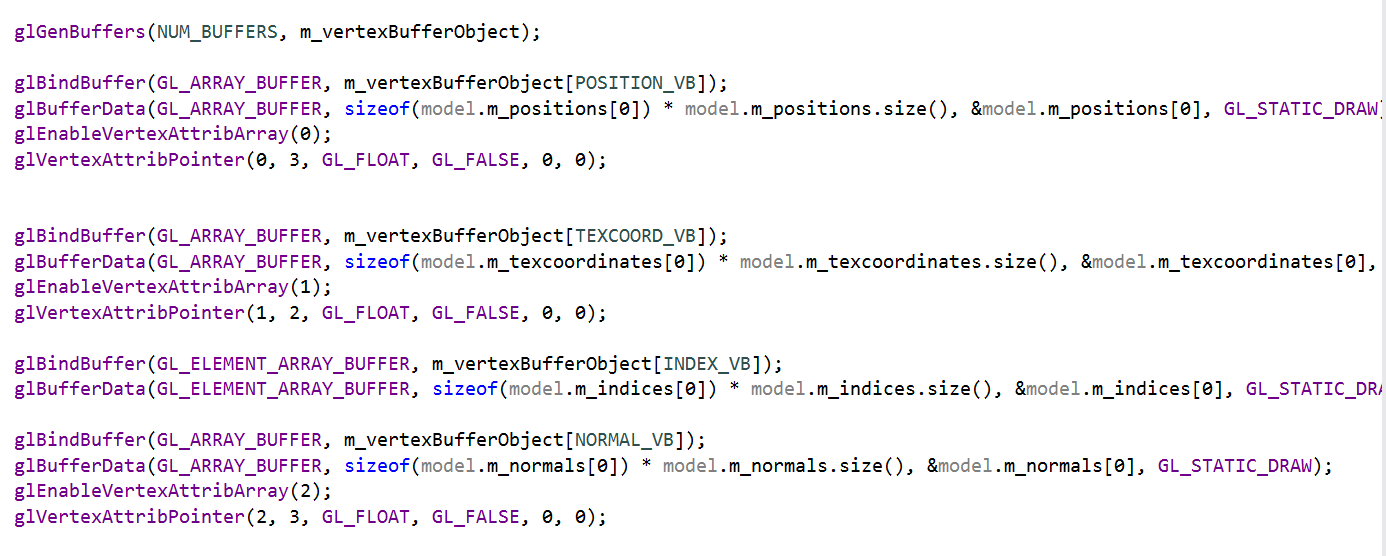
Overall, the solutions provided by these preconceived equations were essential for getting the effects I wanted for my project.

## implementation

For my project’s implementation, I chose to use the combination of the SDL and Glew libraries for our OpenGL implementation as these were the most efficient for creating a window, event handling, passing data from Ram to the GPU and compiling vertex and fragment shaders. For our vectors and matrices, we make use of the GLM library which is excellent for our model view projection calculation which is essential for presenting 3D data in a realistic way.



The OBJ loader and our Mesh class are designed to parse data in an OBJ file, resulting in the vectors for our vertex, uv and normal attributes which we pipe to our GPU for use in our shaders. One of the issues with loading a model from an OBJ file is the vertex data, including normals, provided results in multiple copies of vertices that share the same edge. In order to avoid this and ultimately render our models as efficiently as possible. To avoid this, we use an indexed model to load our faces so we can use the same vertex rather than a copy of the same vertex. To explain how this works, we load in our indicies from our OBJ file, denoted by ‘F’ and contain them in a vector of ints. The indices match the location of the vertices in the vertex buffer. In our implementation, we use the class Indexed Model to contain our positions, texturecoordinates, normals and indices. In order to utilize our indexed model and correctly produce a model, we use our model’s attributes (vertices, uvs ect.) as an array buffer to our buffer objects and for our indices we use an ‘element array buffer’ which parses the buffer. Then when we call Drawelements, we pass in our indices and this function when correctly draws our model, without the needless copies of vertices and normals.



To enable our lighting effects, we needed to predefine variables and pass them into our shaders by utilizing the uniforms type in GLSL. Uniforms remain a constant across an entire draw call, so we have to pass these in to our shader. In order to do this in our shader we need to get the uniform location (in our shader, usually found using its variable name in the shader) and then apply the value we want to assign it to by using the gluniform function glew provides. In effect, we use the getuniformlocation function to assign a GLuint to our uniform location and then use a variable we define in our shader class to assign our uniform the value of the variable we pass in our glUniform functions. This works for float values, vectors and matrices, this is heavily relied upon in our fragment shader to achieve high quality per pixel lighting.

In main we define all the required objects we require such as our display, camera, shader, texture and mesh classes then we loop only if our window isn’t closed and simply bind our shaders, update them and draw our meshes. This simple structure allows us to easily add new objects to our code without great effort and keeps our code in main readable. Overall, comparing the final project to the original design specification I think my design decisions were very successful.

## ANAlysis & conclusion

Overall, the success of my project has been due to my research and careful selection of solutions which ultimately meant my application met the requirements of the project with additional features as extensions. The application unfortunately does not support input mapping, nor does it incorporate multiple shader programs to add variance to our models. This could be fixed by creating a GameObject class which is composed of a mesh, a transform and a shader which would add variation to each of our object in our scene however the deadline imposed did not allow for this extension. The features it does have however are executed efficiently and with high quality.

## References

Shreiner, D; (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL Version 4.3*. 8th ed. -: Addison-Wesley Professional. 984.

TheBennyBox. (2014). *Intro to Modern OpenGL .* Available: https://www.youtube.com/watch?v=ftiKrP3gW3k&list=PLEETnX-uPtBXT9T-hD0Bj31DSnwio-ywh. Last accessed 07/01/2016.

Meiri, E. (2015). *OpenGL step by step.* Available: http://ogldev.atspace.co.uk/. Last accessed 07/01/2016.