IMAT3906 Shader Programming Report

**Introduction**

The primary focus of this project is to further my skills and knowledge in multiple areas, I wanted to learn the theory of rendering pipelines, in particular OpenGL and the practical side of implementing such as system, using object-oriented C++, to do this I would create an OpenGL 3D Renderer.

For my implementation I wanted to build a system from the ground up starting with an empty visual studio project, this way I would maximise the amount I could learn, and I would get to decide how the system would be designed, which would make it easier later on when the project gets more complicated as I would have a full understanding of my code.

For my project to be successful however it needs to showcase the power of advanced shaders and rendering effects, such as texture mapping, normal mapping, and parallax mapping. To achieve this, I wanted to showcase these effects side by side, so a viewer could easily see the differences and advantages of such effects.

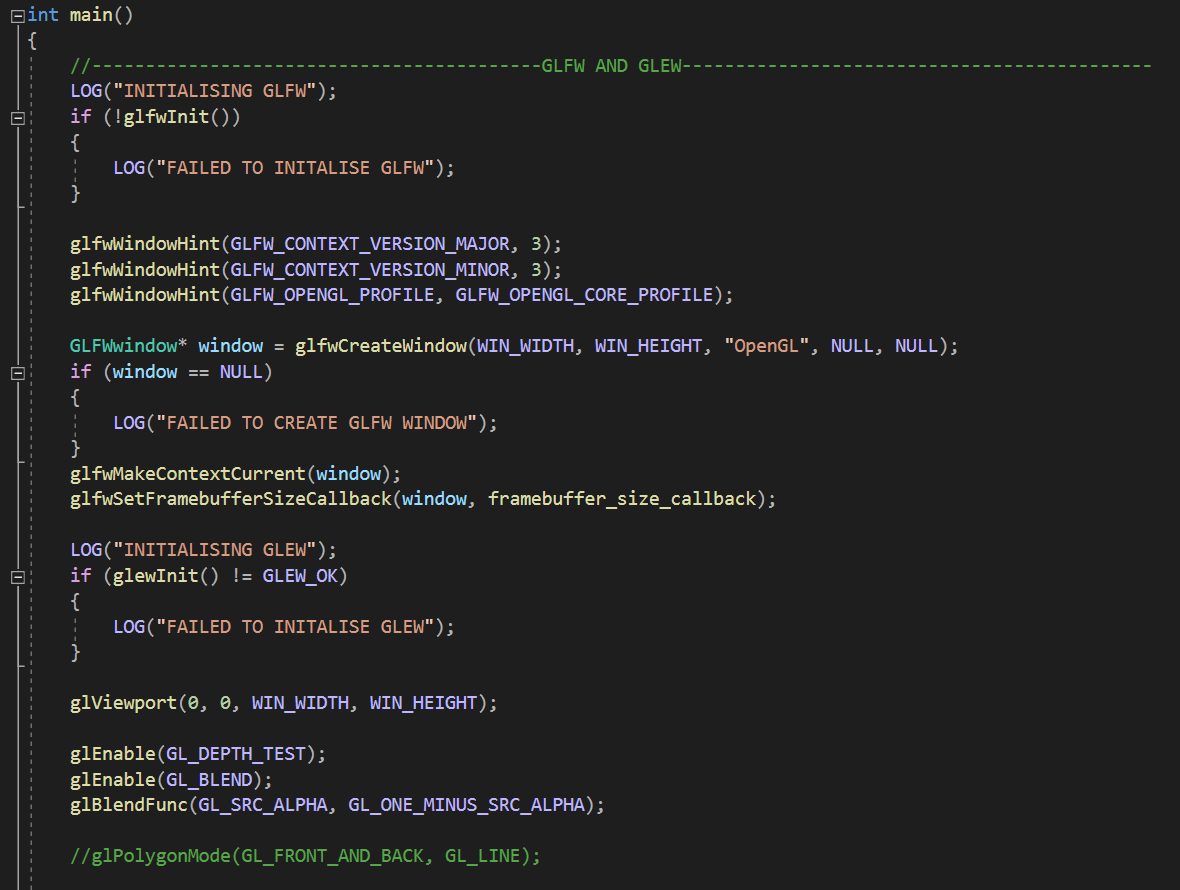
**Resources**

The primary resources I used for this project was the OpenGL video series by ‘The Cherno’ on YouTube which covers the basics of building a 2D renderer, and ‘LearnOpenGL.com’ by ‘Joey de Vries’, this website was especially helpful when trying to understand some of the advanced shader techniques and the issues I may come across.

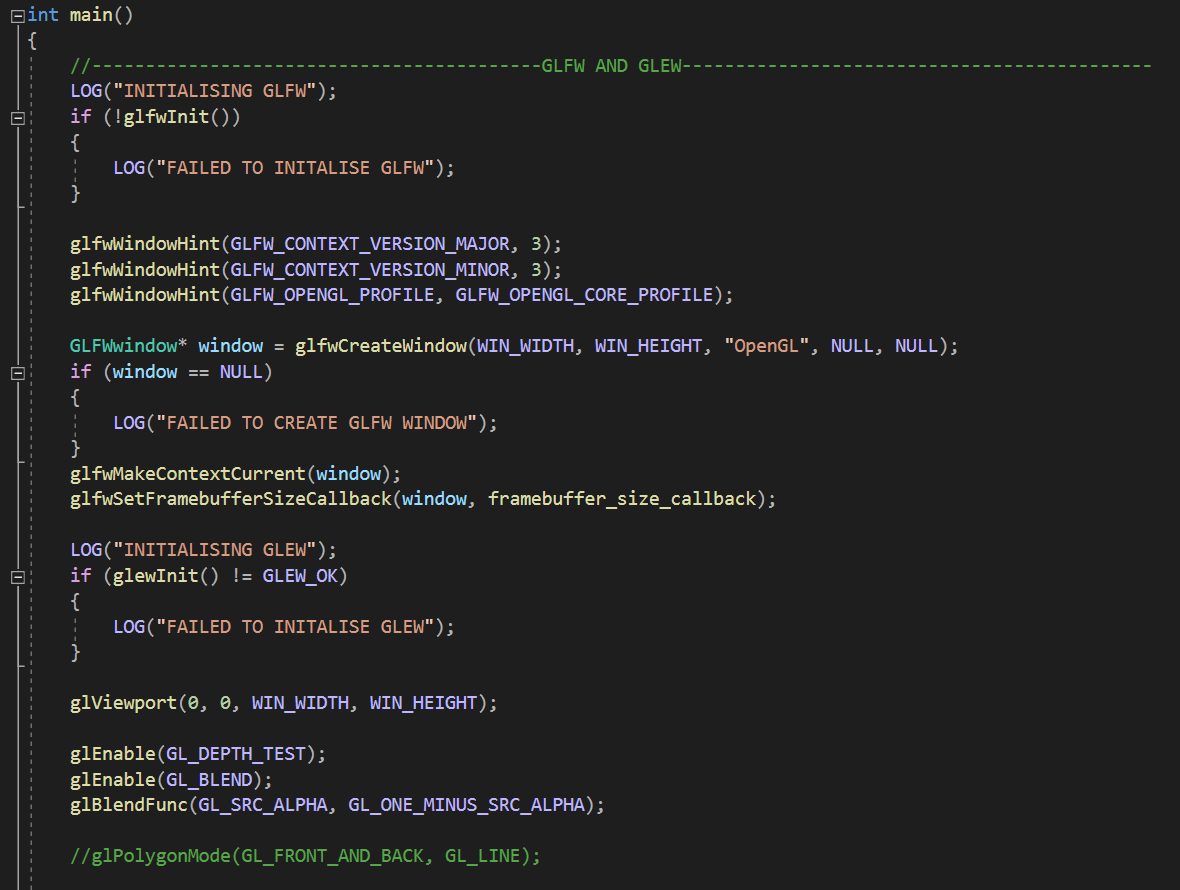
I also made use of several libraries for the project, libraries are resources that I can use in my software to achieve a goal, they can be very simple such as math libraries which contain lots of equations that would be quite tiresome to have to code myself, or even complete game engines containing many sub libraries. I linked all the libraries I used in my project statically, I did this over dynamic linking as it is much easier to do, and I do not need to worry about the bloat it can add to the project. If this where a larger project I would use dynamic linking.

The main two libraries I used where ‘GLFW’ which enables me to make graphics windows and ‘GLEW’ which allows me to access OpenGL function and communicate with the GPU. Other libraries of mention are ‘GLM’ which is a Vector and Matrix library and ‘STB’ which is an image loading library.

**Opening a Window**

In order to be able to render anything I needed to open a window, luckily this is made very easy with ‘GLFW’, I simply needed to instantiate the GLFW library, tell it what version of OpenGL I want to use and then create the window using the appropriate function, passing parameters about the window as well. I made use of macros for the window size so I can change them in one place for the entire project. After the window has been made, I can set it as the current render context, this tells OpenGL where to render the pixels.

**Accessing the GPU**

In order to access the GPU and therefore make use of its parallel performance I needed to set up GLEW, this is done very simply with an initialise function, I then tell it how big the viewport is using the window size macros I have set up. I then execute several render commands, ‘glEnable(GL\_DEPTH\_TEST)’ tells OpenGL to render closer items on top of others, and ‘glEnable(GL\_BLEND)’ this tells OpenGL to blend the alphas of pixels. ‘glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA)’ tells OpenGL how I want the alphas of pixels to be calculated.

**Rendering a Quad**

In order to render a quad, I need to send the relevant data to the GPU, this is done through objects, in my implementation I make use of 5 objects, these are Vertex Array Objects, Vertex Buffer Objects, Index Buffer Objects, Buffer Layouts, and Shaders. I abstracted these objects into classes to make my life easier when designing the software.

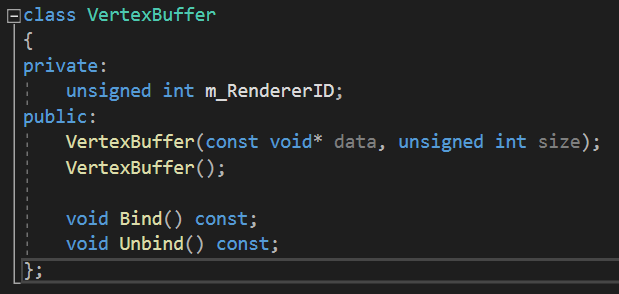
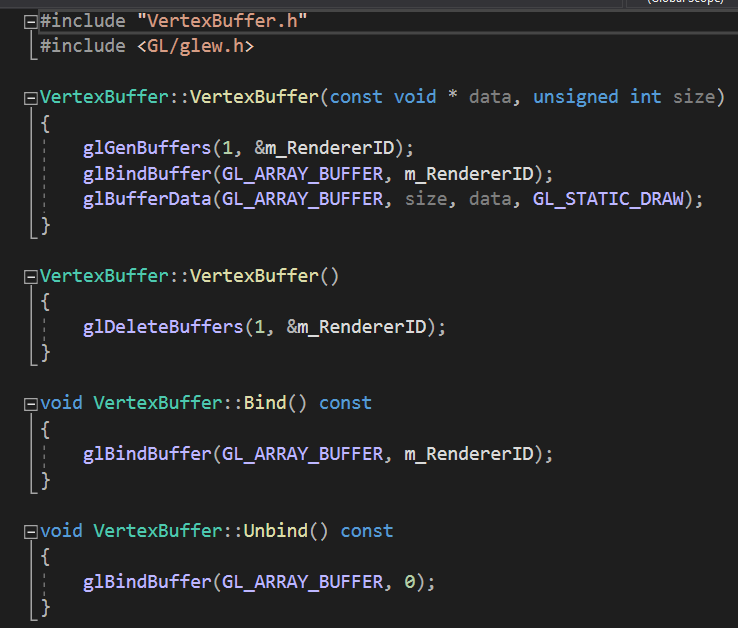
**Rendering a Quad – Vertex Array**

A Vertex array is an object that can store one or more Vertex Buffers, this data is store inside the high-speed memory of the GPU. They also tie a Buffer Layout to the Vertex Buffers inside the array, so they don’t have to be reinitialised every time a draw call happens.

In the constructor for my Vertex Array class I generate a vertex array object and then store its id, at the appropriate time I would pass a vertex buffer and a buffer layout to the vertex array using the ‘AddBuffer’ function, I then loop over the elements of the buffer layout and enable the vertex attributes, I calculate the offset and retrieve the stride at each iteration.

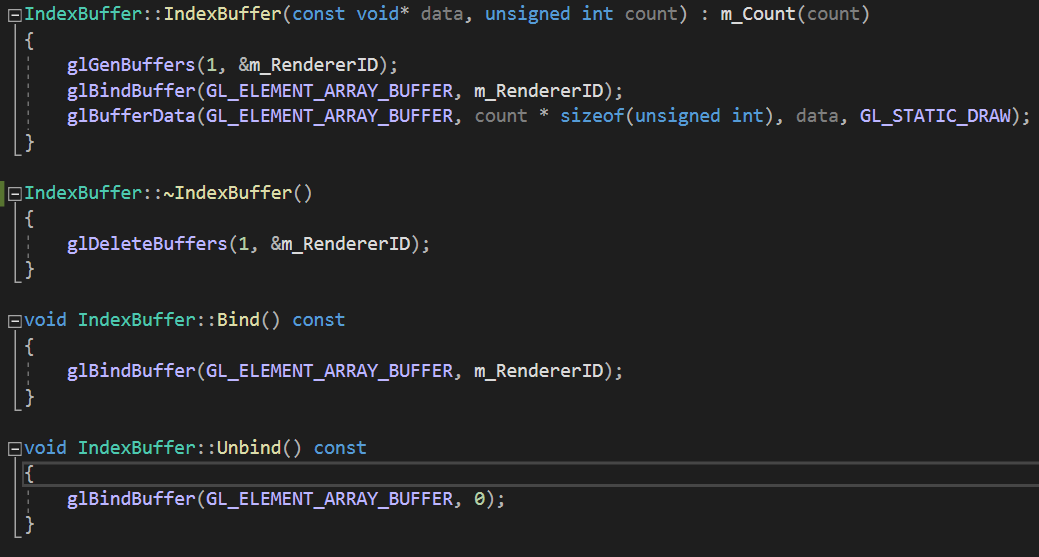
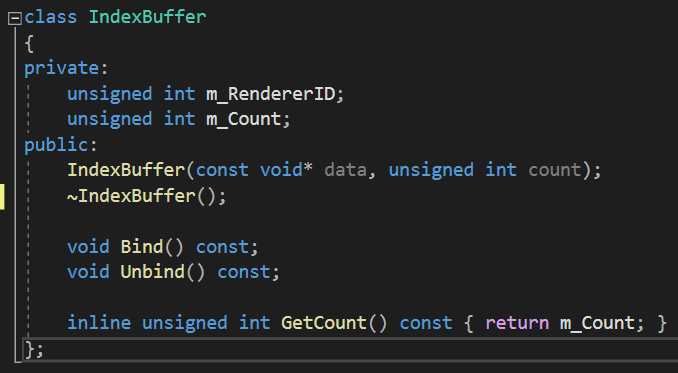
**Rendering a Quad – Vertex Buffer**

A Vertex buffer is an object that stores vertex information on the GPU, this information can be almost any data type, is usually integers and floats. A Vertex Buffer is usually accompanied by a Buffer Layout, they tell OpenGL the structure of the Vertex Buffer and how to handle the data.

In the constructer for my Vertex Buffer class I generate and bind a vertex buffer object using the OpenGL calls and then load the data that is passed into the constructer, this loads the vertex data onto the GPU memory, the amount of memory is the same as the size variable I pass into the constructor. I load this data with parameter of ‘GL\_STATIC\_DRAW’ as I won’t be changing the contents of the vertex buffer. I also implemented a bind and unbind function for when I want to make use of this vertex array.

**Rendering a Quad – Index Buffer**

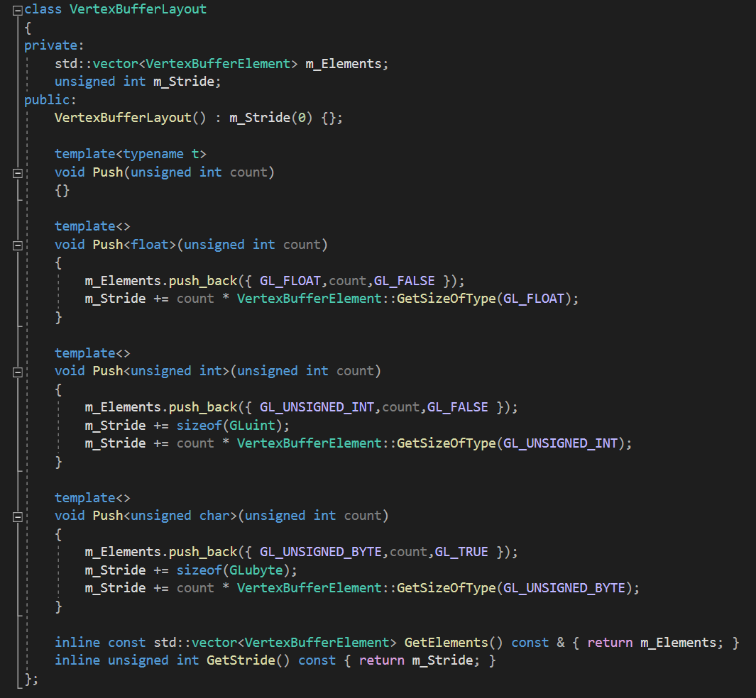
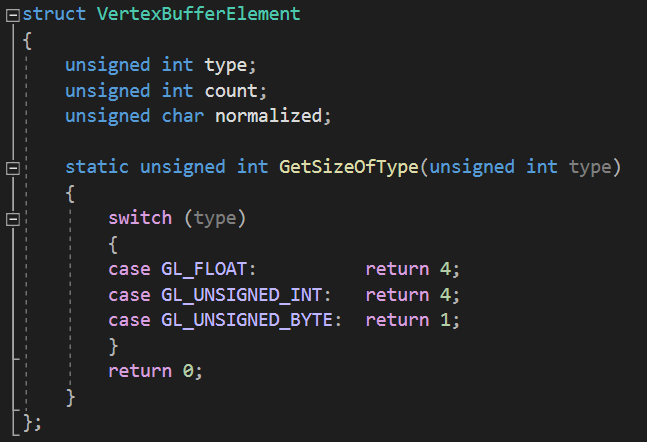
An Index Buffer is an object used to improve the efficiency of the render pipeline, it is very common for 3d models to share vertex data, therefore it makes sense to share vertex data instead of repeating it and wasting memory. It is usually a list of integers representing the index of a bit of vertex data.

My implementation of an Index Buffer class is very similar to my Vertex Buffer, this isn’t just by chance they are essentially the same thing just store a different type of data. One change I made is I pass the count in, rather than the size of the data into the constructer as I want to store the count for later use. Just like the vertex buffer the data is loaded statically, and I have a bind and unbind functions.

**Rendering a Quad – Buffer Layout**

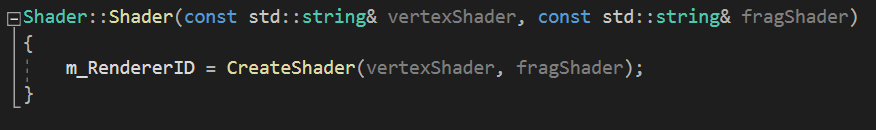
A buffer layout is an object that tells OpenGL what type of data is inside the vertex buffer, how many of each type of data there is, and the gap between the start of each piece of vertex data, known as the stride. There function is also tied closely to shader programs.

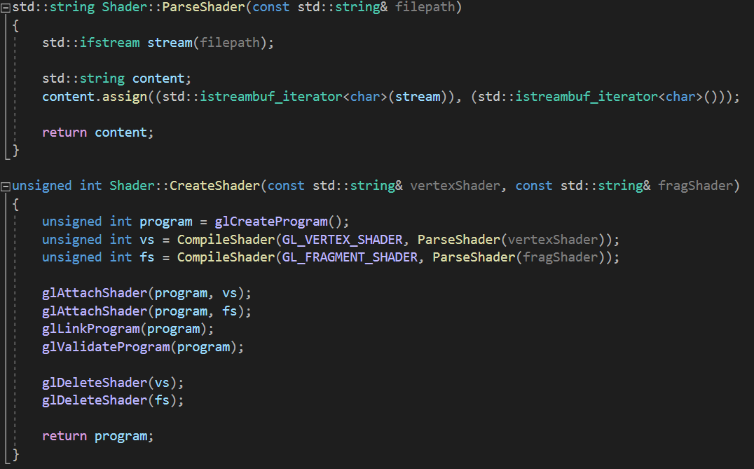
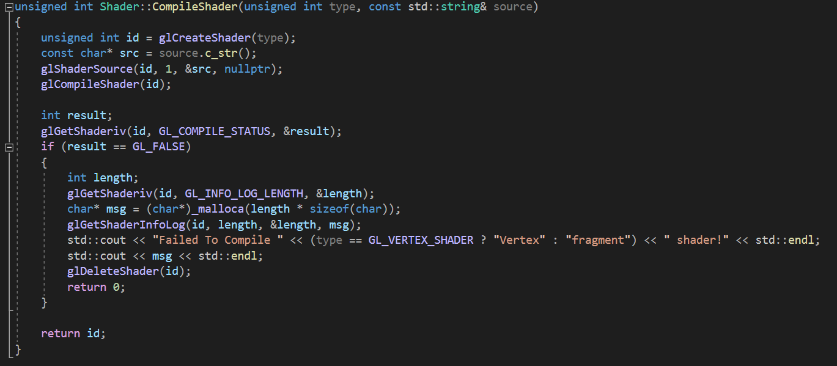
My Buffer Layout class begins with a struct with represents a Vertex Buffer Element, it holds the type of data the element is as an integer which is the same as the OpenGL data types, how many of this type of data there is, and if it is normalize or not. I also made a function to tell me the size of the data type, this is useful when calculating the stride and offset within buffer layouts.

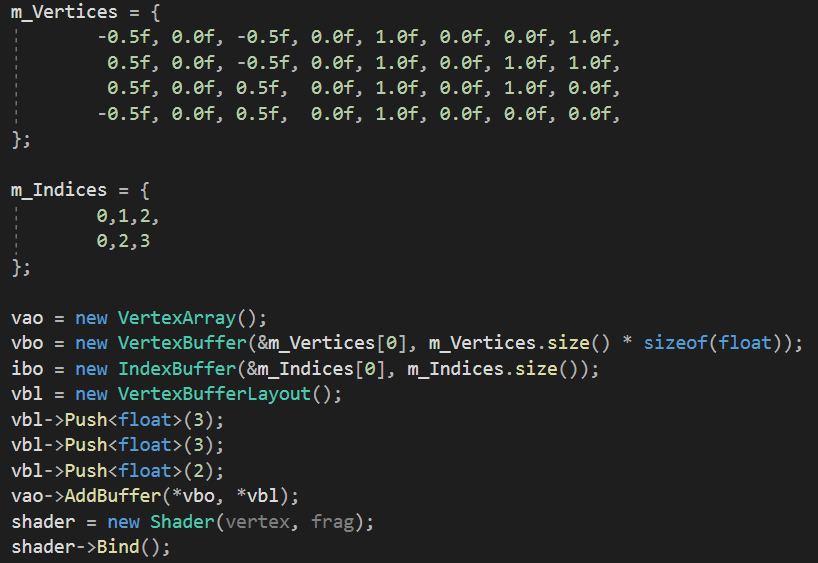
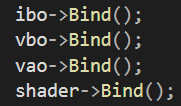
These Buffer Elements are held within a Buffer Layout which keeps track of all the elements and the current stride of a set of vertex data. I made use of C++ templates to build a more robust system for adding elements to the layout. I simply need to call the push function and give it a data type and count and the function handles the rest.

**Rendering a Quad – Shaders**

The final thing needed in order to render a quad is a shader program, a shader program calculates where pixels should be rendered on the screen using fragments and then calculates the colour of the fragment based on inputs. This process is usually split into two programs known as the vertex shader, which handles the creating of fragments from vertex positions and the fragment shader which handles the colouring of fragments.

My Shader class handles the parsing of shader files, the creation of the shader program, and setting uniforms within the shader. I first must pass in the file paths of the vertex and fragment shader I want to use.

This then runs the create shader function which creates a shader program suing the OpenGL calls, storing the id of this object as an integer, then it will parse and compile both shaders, checking for any errors. It will then attach the shaders to the shader program and validate it, as well as cleaning up any loose objects.

Finally, to render a quad I put all these objects together inside the main function of our software, loaded in the relevant data, bound all the objects, and issued a draw call.

**Reflection**

**Appendix**