George Hanks P17173176

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

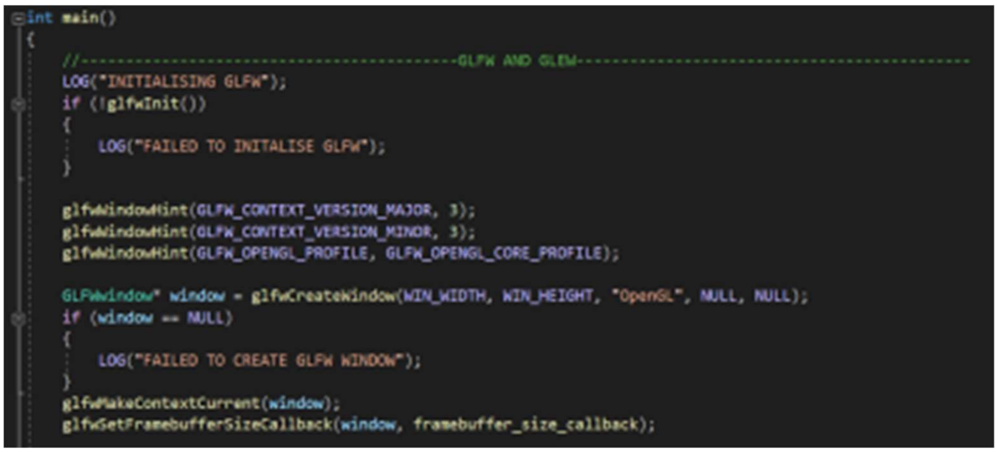
The primary resources I used for this project was the OpenGL video series by ‘The Cherno’ on YouTube which covers the basics of building a 2D renderer, and ‘LearnOpenGL.com’ by ‘Joey de Vries’, this website was especially helpful when trying to understand some of the advanced shader techniques and the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

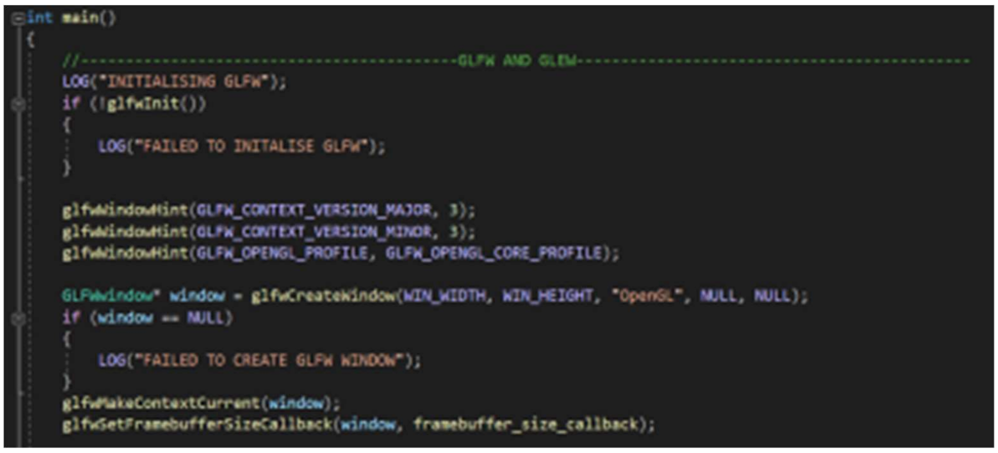
In order to be able to render anything I needed to open a window, luckily this is made very easy with ‘GLFW’, I simply needed to instantiate the GLFW library, tell it what version of OpenGL I want to use and then create the window using the appropriate function, passing parameters about the window as 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.



```
int main()
{
    //-----------------------------------------GLFW AND GLEW-----------------------------------------
    LOG("INITIALISING GLFW");
    if (!glfwInit())
    {
        LOG("FAILED TO INITALISE GLFW");
    }

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* window = glfwCreateWindow(WIN_WIDTH, WIN_HEIGHT, "OpenGL", NULL, NULL);
    if (window == NULL)
    {
        LOG("FAILED TO CREATE GLFW WINDOW");
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

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.



```
int main()
{
    //-----------------------------------------GLFW AND GLEW-----------------------------------------
    LOG("INITIALISING GLFW");
    if (!glfwInit())
    {
        LOG("FAILED TO INITALISE GLFW");
    }

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* window = glfwCreateWindow(WIN_WIDTH, WIN_HEIGHT, "OpenGL", NULL, NULL);
    if (window == NULL)
    {
        LOG("FAILED TO CREATE GLFW WINDOW");
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

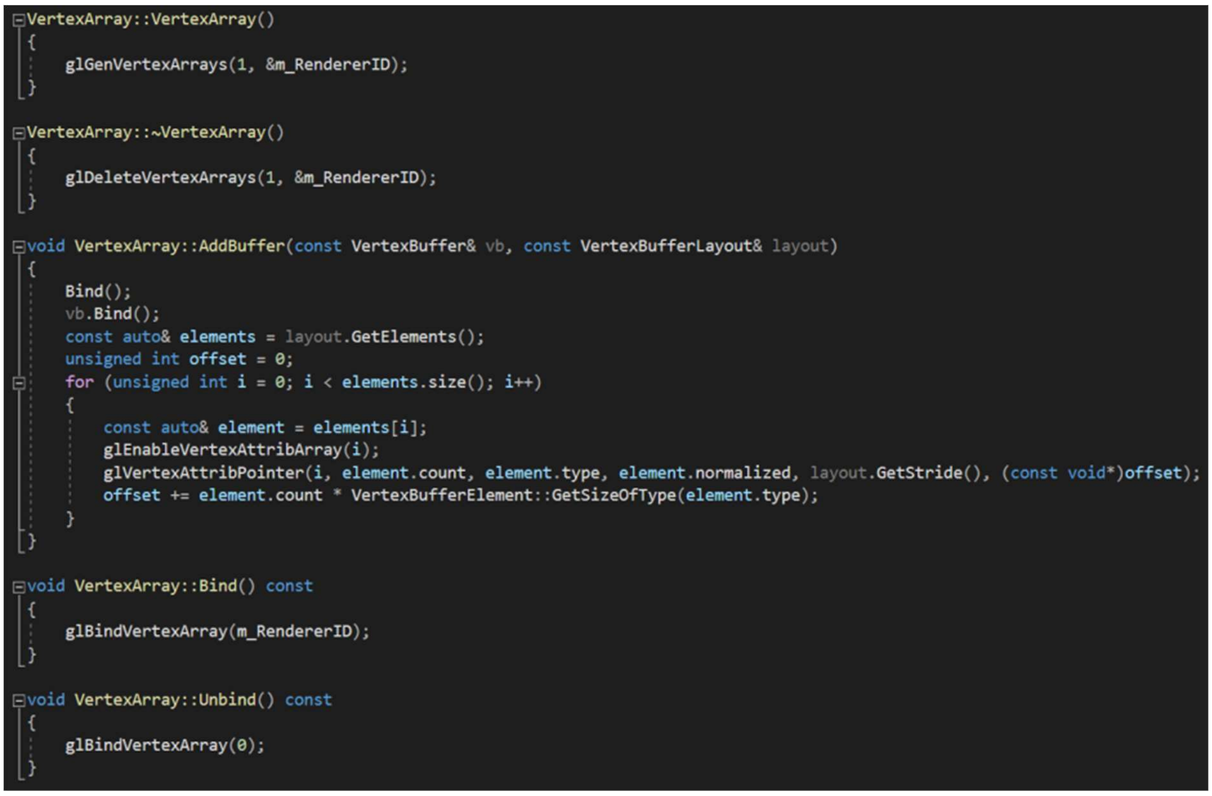
Rendering a Quad

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 on the GPU, 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. There is a bind and unbind function for when I want to use a Vertex Array.



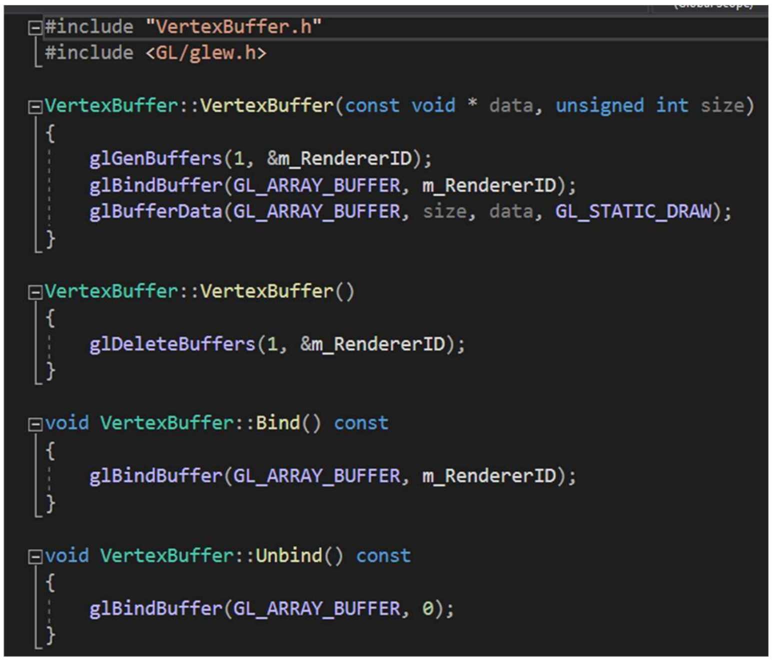
```
VertexArray::VertexArray()
{
    glGenVertexArrays(1, &m_RendererID);
}

VertexArray::~VertexArray()
{
    glDeleteVertexArrays(1, &m_RendererID);
}

void VertexArray::AddBuffer(const VertexBuffer& vb, const VertexBufferLayout& layout)
{
    Bind();
    vb.Bind();
    const auto& elements = layout.GetElements();
    unsigned int offset = 0;
    for (unsigned int i = 0; i < elements.size(); i++)
    {
        const auto& element = elements[i];
        glEnableVertexAttribArray(i);
        glVertexAttribPointer(i, element.count, element.type, element.normalized, layout.GetStride(), (const void*)offset);
        offset += element.count * VertexBufferElement::GetSizeOfType(element.type);
    }
}

void VertexArray::Bind() const
{
    glBindVertexArray(m_RendererID);
}

void VertexArray::Unbind() const
{
    glBindVertexArray(0);
}
```

George Hanks P17173176

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. The Vertex Buffer is eventually passed into the shader program and will be used to render an object.

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 buffer.

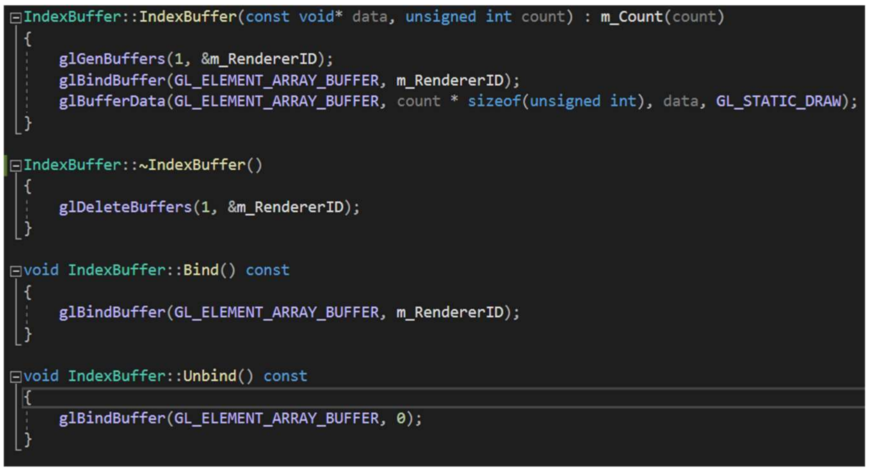


```
#include "VertexBuffer.h"
#include <GL/glew.h>

VertexBuffer::VertexBuffer(const void * data, unsigned int size)
{
    glGenBuffers(1, &m_RendererID);
    glBindBuffer(GL_ARRAY_BUFFER, m_RendererID);
    glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
}

VertexBuffer::VertexBuffer()
{
    glDeleteBuffers(1, &m_RendererID);
}

void VertexBuffer::Bind() const
{
    glBindBuffer(GL_ARRAY_BUFFER, m_RendererID);
}

void VertexBuffer::Unbind() const
{
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

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.



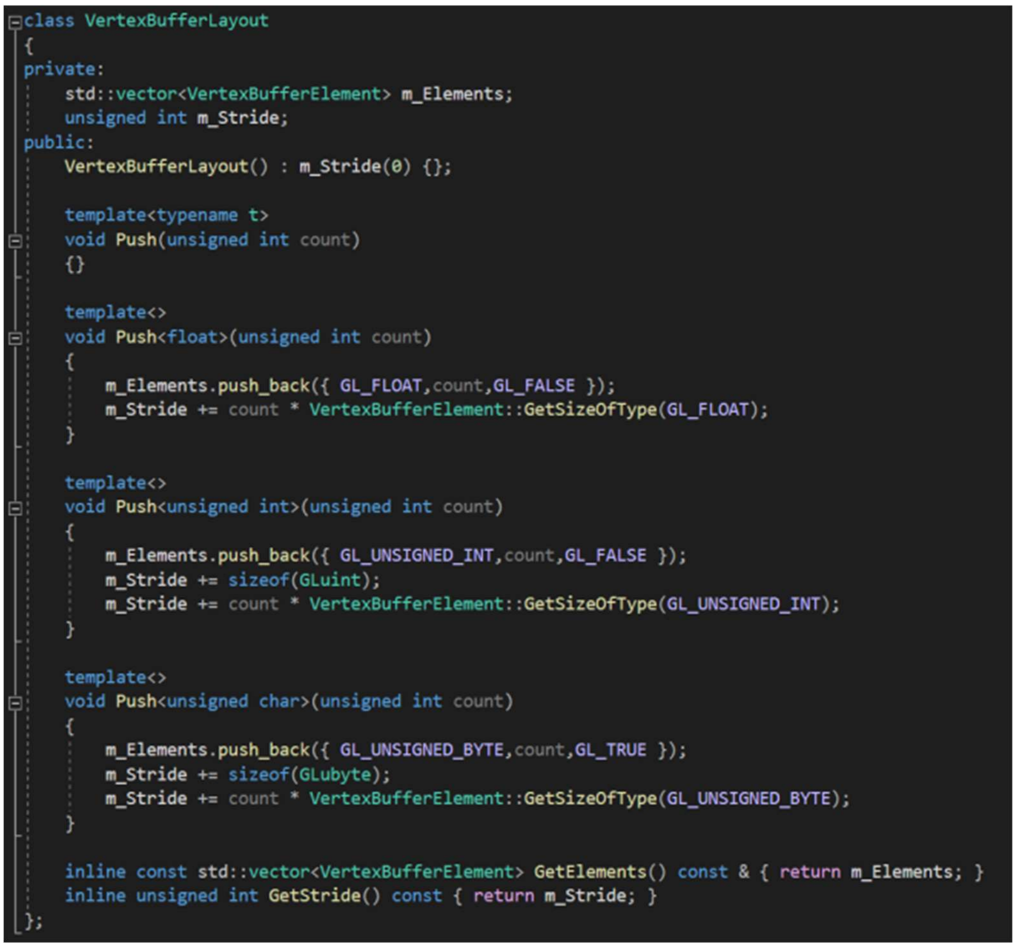
```
IndexBuffer::IndexBuffer(const void* data, unsigned int count) : m_Count(count)
{
    glGenBuffers(1, &m_RendererID);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererID);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, count * sizeof(unsigned int), data, GL_STATIC_DRAW);
}

IndexBuffer::~IndexBuffer()
{
    glDeleteBuffers(1, &m_RendererID);
}

void IndexBuffer::Bind() const
{
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererID);
}

void IndexBuffer::Unbind() const
{
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
```

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.



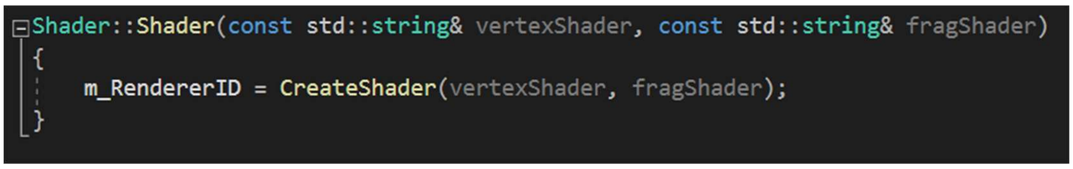
```cpp
class VertexBufferLayout
{
private:
    std::vector<VertexBufferElement> m_Elements;
    unsigned int m_Stride;
public:
    VertexBufferLayout() : m_Stride(0) {};

    template<typename t>
    void Push(unsigned int count)
    {}

    template<>
    void Push<float>(unsigned int count)
    {
        m_Elements.push_back({ GL_FLOAT,count,GL_FALSE });
        m_Stride += count * VertexBufferElement::GetSizeOfType(GL_FLOAT);
    }

    template<>
    void Push<unsigned int>(unsigned int count)
    {
        m_Elements.push_back({ GL_UNSIGNED_INT,count,GL_FALSE });
        m_Stride += sizeof(GLuint);
        m_Stride += count * VertexBufferElement::GetSizeOfType(GL_UNSIGNED_INT);
    }

    template<>
    void Push<unsigned char>(unsigned int count)
    {
        m_Elements.push_back({ GL_UNSIGNED_BYTE,count,GL_TRUE });
        m_Stride += sizeof(GLubyte);
        m_Stride += count * VertexBufferElement::GetSizeOfType(GL_UNSIGNED_BYTE);
    }

    inline const std::vector<VertexBufferElement> GetElements() const & { return m_Elements; }
    inline unsigned int GetStride() const { return m_Stride; }
};
```

Rendering a Quad – Shaders

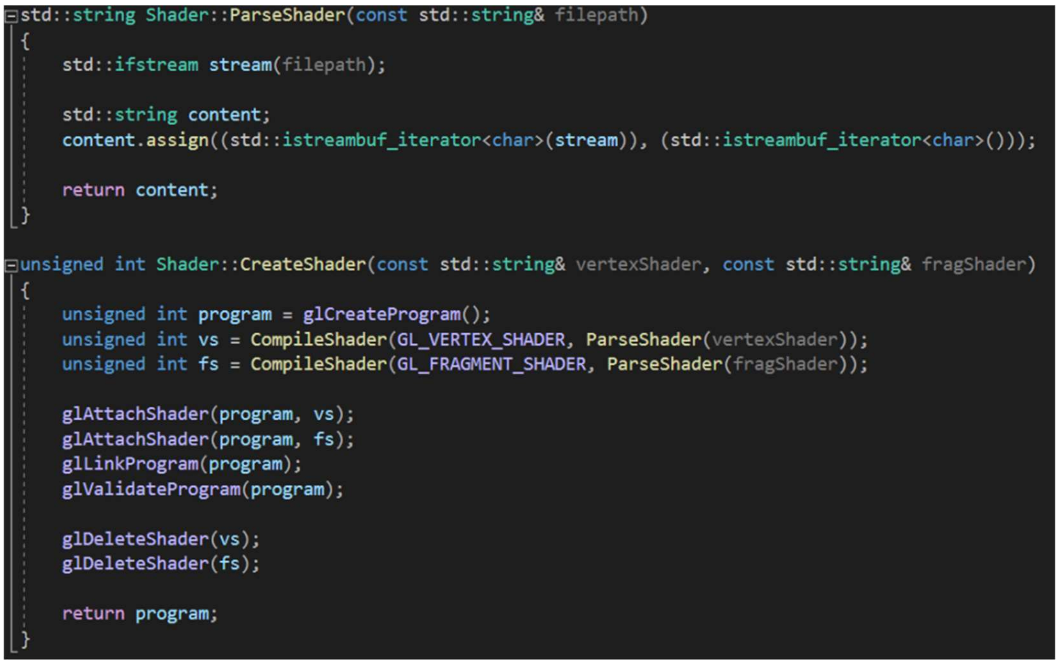
The final thing needed in order to render a quad is a shader program, a shader program calculates where pixels should be rendered on the screen using fragments and then calculates the colour of the fragment based on inputs. These inputs will be either vertex data or uniforms, vertex data changes for each vertex and uniforms stay the same for the entire program. 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.

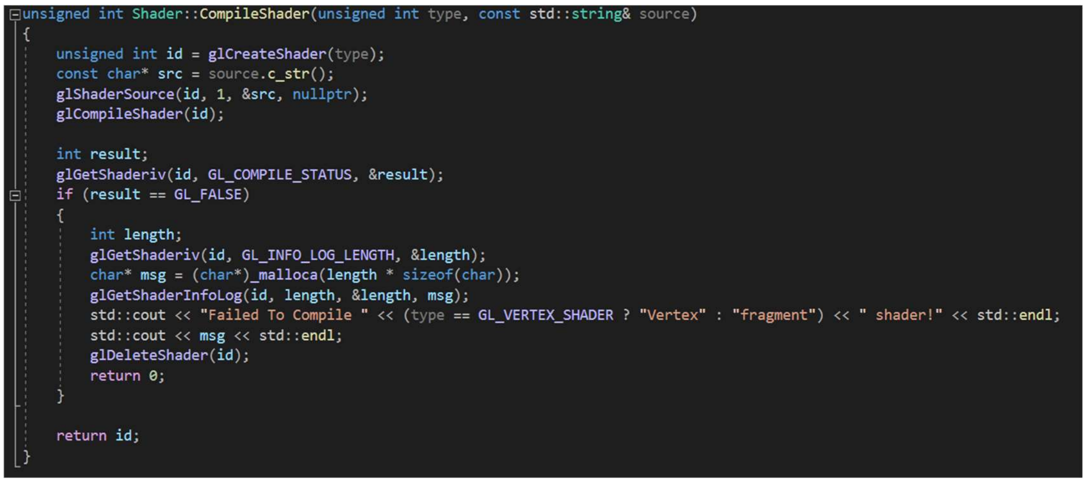
This then runs the create shader function which creates a shader program suing the OpenGL calls, storing the id of this object as an integer, then it will parse and compile both shaders, checking for any errors. It will then attach the shaders to the shader program and validate it, as well as cleaning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



```cpp
Shader::Shader(const std::string& vertexShader, const std::string& fragShader)
{
    m_RendererID = CreateShader(vertexShader, fragShader);
}
```

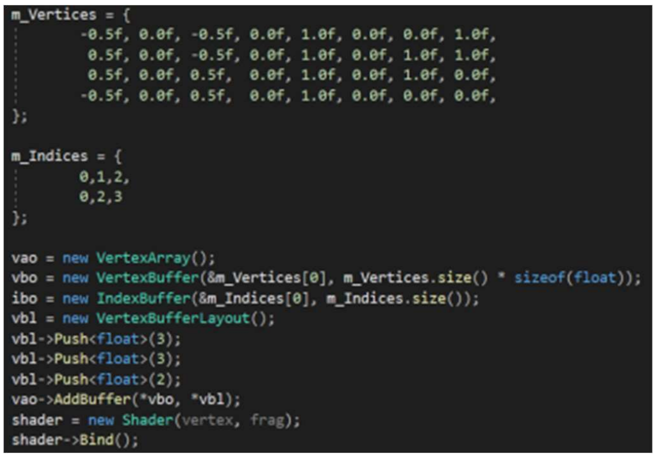


```cpp
std::string Shader::ParseShader(const std::string& filepath)
{
    std::ifstream stream(filepath);

    std::string content;
    content.assign((std::istreambuf_iterator<char>(stream)), (std::istreambuf_iterator<char>()));

    return content;
}

unsigned int Shader::CreateShader(const std::string& vertexShader, const std::string& fragShader)
{
    unsigned int program = glCreateProgram();
    unsigned int vs = CompileShader(GL_VERTEX_SHADER, ParseShader(vertexShader));
    unsigned int fs = CompileShader(GL_FRAGMENT_SHADER, ParseShader(fragShader));

    glAttachShader(program, vs);
    glAttachShader(program, fs);
    glLinkProgram(program);
    glValidateProgram(program);

    glDeleteShader(vs);
    glDeleteShader(fs);

    return program;
}
```

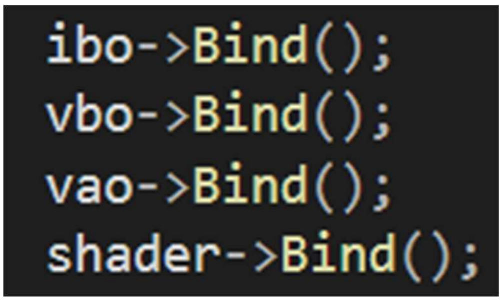


```cpp
unsigned int Shader::CompileShader(unsigned int type, const std::string& source)
{
    unsigned int id = glCreateShader(type);
    const char* src = source.c_str();
    glShaderSource(id, 1, &src, nullptr);
    glCompileShader(id);

    int result;
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE)
    {
        int length;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
        char* msg = (char*)_malloca(length * sizeof(char));
        glGetShaderInfoLog(id, length, &length, msg);
        std::cout << "Failed To Compile " << (type == GL_VERTEX_SHADER ? "Vertex" : "fragment") << " shader!" << std::endl;
        std::cout << msg << std::endl;
        glDeleteShader(id);
        return 0;
    }

    return id;
}
```

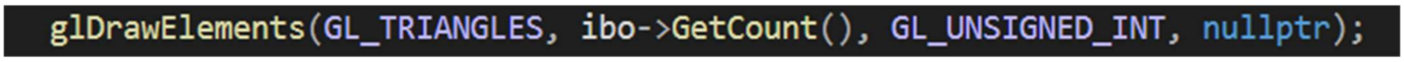
Finally, to render a quad I put these objects together inside the main function of my software, load in the relevant data, bind the objects, and issue a draw call.



```cpp
m_Vertices = {
        -0.5f, 0.0f, -0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f,
         0.5f, 0.0f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,
         0.5f, 0.0f, 0.5f,  0.0f, 1.0f, 0.0f, 1.0f, 0.0f,
        -0.5f, 0.0f, 0.5f,  0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
};

m_Indices = {
        0,1,2,
        0,2,3
};

vao = new VertexArray();
vbo = new VertexBuffer(&m_Vertices[0], m_Vertices.size() * sizeof(float));
ibo = new IndexBuffer(&m_Indices[0], m_Indices.size());
vbl = new VertexBufferLayout();
vbl->Push<float>(3);
vbl->Push<float>(3);
vbl->Push<float>(2);
vao->AddBuffer(*vbo, *vbl);
shader = new Shader(vertex, frag);
shader->Bind();
```



```cpp
ibo->Bind();
vbo->Bind();
vao->Bind();
shader->Bind();
```



```cpp
glDrawElements(GL_TRIANGLES, ibo->GetCount(), GL_UNSIGNED_INT, nullptr);
```

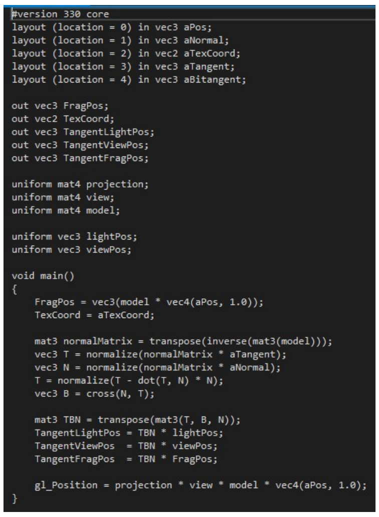
George Hanks P17173176

Normal Mapping

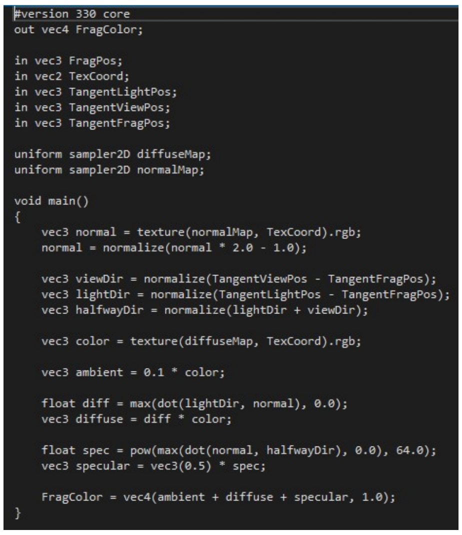
Normal mapping is an advanced shading technique that will alter the colour of pixels to make it appear that there is more detail in the mesh than there actually is, it does this by altering the normal of a fragment using a normal map, the RGB channels of a normal map are used to create a normal vector for the fragment, this then alters the diffuse and specular calculations, creating highlights and shadows.

Normal mapping requires that we have per fragment normals, to do this I passed a normal map (a texture with per fragment data) into the fragment shader using a 'Sampler2D' uniform, I then create a normal vector by converting the RGB values of a point on the texture into XYZ coordinates, the point on the texture is determined by the texture coordinates of the fragment. This normal vector then replaces the existing vertex normal in the lighting calculations.

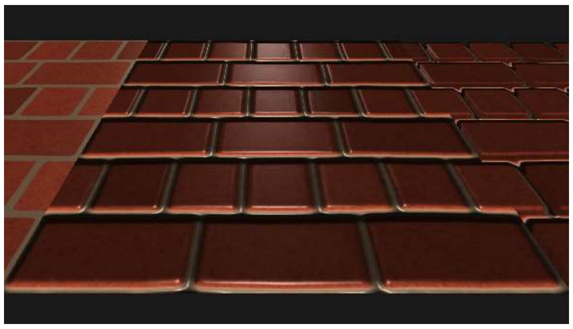
However, this normal is in tangent space, it does not consider the rotation of the model, so in order to get an accurate render, we need to transform the other data in the shader into tangent space. To do this I needed tangent and bitangent normals, this is usually handled by the model loader and included with the vertex data, but I manually created my own vertexes, so I had to calculate my own normals.



```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;
layout (location = 3) in vec3 aTangent;
layout (location = 4) in vec3 aBitangent;

out vec3 FragPos;
out vec2 TexCoord;
out vec3 TangentLightPos;
out vec3 TangentViewPos;
out vec3 TangentFragPos;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

uniform vec3 lightPos;
uniform vec3 viewPos;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    TexCoord = aTexCoord;

    mat3 normalMatrix = transpose(inverse(mat3(model)));
    vec3 T = normalize(normalMatrix * aTangent);
    vec3 N = normalize(normalMatrix * aNormal);
    T = normalize(T - dot(T, N) * N);
    vec3 B = cross(N, T);

    mat3 TBN = transpose(mat3(T, B, N));
    TangentLightPos = TBN * lightPos;
    TangentViewPos  = TBN * viewPos;
    TangentFragPos  = TBN * FragPos;

    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```



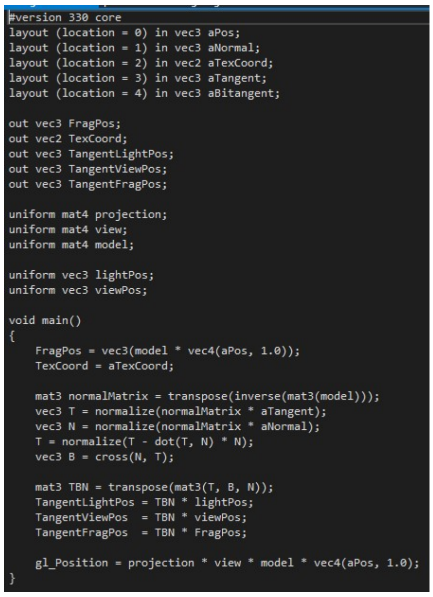
```
#version 330 core
out vec4 FragColor;

in vec3 FragPos;
in vec2 TexCoord;
in vec3 TangentLightPos;
in vec3 TangentViewPos;
in vec3 TangentFragPos;

uniform sampler2D diffuseMap;
uniform sampler2D normalMap;

void main()
{
    vec3 normal = texture(normalMap, TexCoord).rgb;
    normal = normalize(normal * 2.0 - 1.0);

    vec3 viewDir = normalize(TangentViewPos - TangentFragPos);
    vec3 lightDir = normalize(TangentLightPos - TangentFragPos);
    vec3 halfwayDir = normalize(lightDir + viewDir);

    vec3 color = texture(diffuseMap, TexCoord).rgb;

    vec3 ambient = 0.1 * color;

    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * color;

    float spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = vec3(0.5) * spec;

    FragColor = vec4(ambient + diffuse + specular, 1.0);
}
```



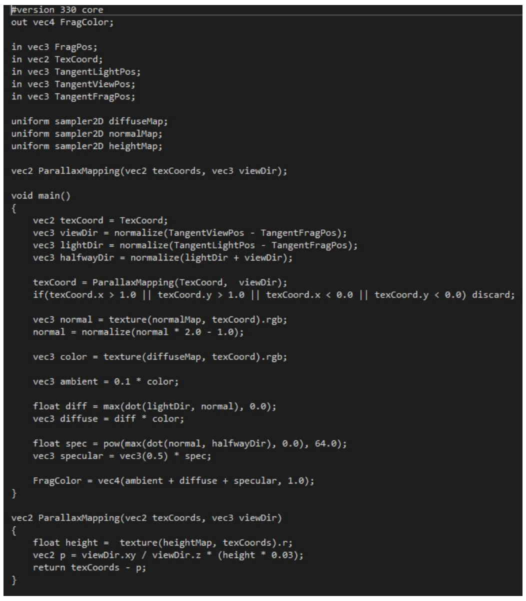
Parallax Mapping

Parallax mapping is like normal mapping in its goal to imitate more detail in a mesh however it achieves this goal in a different way. Parallax mapping will alter the texture coordinates of a fragment, using a displacement map / height map to create the illusion of depth. Despite the 3D look parallax mapping does not require any extra geometry which makes is a very cost-effective technique.

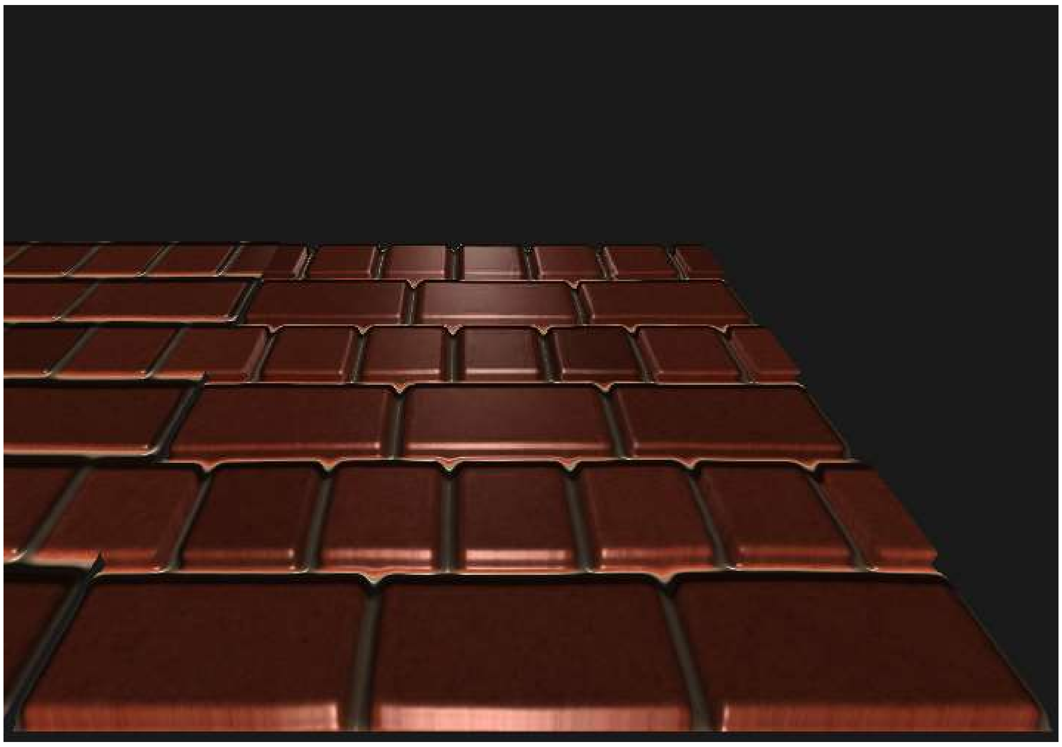
To achieve parallax mapping I took my existing normal mapping shader and introduced a height map into the fragment shader using a Sample2D uniform. I would then pass the texture coordinates and the view direction vector into a function which would output a new texture coordinate. This function uses the height map along with a calculation to determine the new offset texture coordinates. This texture coordinate then replaces the original for the rest of the fragment shader.



```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;
layout (location = 3) in vec3 aTangent;
layout (location = 4) in vec3 aBitangent;

out vec3 FragPos;
out vec2 TexCoord;
out vec3 TangentLightPos;
out vec3 TangentViewPos;
out vec3 TangentFragPos;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

uniform vec3 lightPos;
uniform vec3 viewPos;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    TexCoord = aTexCoord;

    mat3 normalMatrix = transpose(inverse(mat3(model)));
    vec3 T = normalize(normalMatrix * aTangent);
    vec3 N = normalize(normalMatrix * aNormal);
    T = normalize(T - dot(T, N) * N);
    vec3 B = cross(N, T);

    mat3 TBN = transpose(mat3(T, B, N));
    TangentLightPos = TBN * lightPos;
    TangentViewPos  = TBN * viewPos;
    TangentFragPos  = TBN * FragPos;

    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```



```
#version 330 core
out vec4 FragColor;

in vec3 FragPos;
in vec2 TexCoord;
in vec3 TangentLightPos;
in vec3 TangentViewPos;
in vec3 TangentFragPos;

uniform sampler2D diffuseMap;
uniform sampler2D normalMap;
uniform sampler2D heightMap;

vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir);

void main()
{
    vec2 texCoord = TexCoord;
    vec3 viewDir = normalize(TangentViewPos - TangentFragPos);
    vec3 lightDir = normalize(TangentLightPos - TangentFragPos);
    vec3 halfwayDir = normalize(lightDir + viewDir);

    texCoord = ParallaxMapping(TexCoord,  viewDir);
    if(texCoord.x > 1.0 || texCoord.y > 1.0 || texCoord.x < 0.0 || texCoord.y < 0.0) discard;

    vec3 normal = texture(normalMap, texCoord).rgb;
    normal = normalize(normal * 2.0 - 1.0);

    vec3 color = texture(diffuseMap, texCoord).rgb;

    vec3 ambient = 0.1 * color;

    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * color;

    float spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = vec3(0.5) * spec;

    FragColor = vec4(ambient + diffuse + specular, 1.0);
}

vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)
{
    float height =  texture(heightMap, texCoords).r;
    vec2 p = viewDir.xy / viewDir.z * (height * 0.03);
    return texCoords - p;
}
```



Reflection

Overall, I am very happing with how the project went, I have learnt a great deal about the rendering pipeline, and shaders. I think the software I have produced is an excellent representation of my practical skills with object-oriented C++ and OpenGL.

However, if I were to have more time to complete the project, I would have like to introduce model loading into the project, this would have increased the variety of scenarios I could have demonstrated, but at that point the project becomes more about building a fully featured 3D rendered rather than exploring advanced shading techniques.