# Graphics Programming Coursework

Name: Steven Smith

Matriculation Number: S1315451

Course: BSc/BSc (Hons) Computer Games (Software Development)

Module: Graphics Programming

Disclaimer:-

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

*Steven Smith*

Contents:

[1.0 Code Description 3](#_Toc481451385)

[1.1 Set-up 3](#_Toc481451386)

[1.2 Display Class 3](#_Toc481451387)

[1.3 Shader Class 4](#_Toc481451388)

[1.4 Shaders 6](#_Toc481451389)

[1.4.1 Vertex Shader 6](#_Toc481451390)

[1.4.2 Fragment Shader 6](#_Toc481451391)

[1.5 Mesh Class 7](#_Toc481451392)

[1.6 Texture Class 8](#_Toc481451393)

[1.7 transform.h 9](#_Toc481451394)

[1.8 camera.h 10](#_Toc481451395)

[1.9 main.cpp 10](#_Toc481451396)

[2.0 External scripts 11](#_Toc481451397)

[3.0 Figures 11](#_Toc481451398)

[4.0 References 13](#_Toc481451399)

[5.0 Bibliography 13](#_Toc481451400)

# 1.0 Code Description

# 1.1 Set-up

For setting up the project, a few software libraries were downloaded to provide useful functions when creating the code for the project. The first software library added was GLEW, this provides all the OpenGL functions that are supported by the GPU. GLM was downloaded as it provides the means for working out complex sums with built-in functions which saves programming time. The last library added was SDL2, this provides tools which help with the cross-platforming of window creation, etc. These libraries were downloaded and added to the project to help with certain functions and to save time when programming. These libraries were downloaded and the files added into two folders “include” and “lib” respectively. The files in the project were then finally linked up with Visual Studio.

# 1.2 Display Class

The display class handles the creation of the window. The Display constructor takes in two integers which determine the width of the window and the height of the window, a string is also passed in which will determine the name of the created window. To save the data that is stored in the display, SDL2 is used to create a variable SDL\_Window which is stored as a pointer to save memory, this is called m\_Window. An SDL\_GLContext called m\_glContext is created to store the context data that will be accessed later. A boolean is also created to ensure the window is open, this bool is called m\_isClosed.

The Display constructor first initialises SDL so it can be used for drawing the window, at this point SDL\_INIT\_EVERYTHING is passed in to the SDL\_Init() function to ensure everything needed will be initialised. After this a list of attributes is set using SDL\_GL\_SetAttribute(), these attributes that are passed in help with drawing colour to the screen and how much data a single pixel should be allocated etc. The window is then created using the SDL\_CreateWindow function, this function takes in a window name, where the window should be created, the windows width and height and any flags that should be used, SDL\_WINDOW\_OPENGL is added to the flags to ensure SDL knows that OpenGL is being used to draw the window. After this the context from SDL is saved onto m\_glContext using the SDL\_GL\_CreateContext() function and passing in the window (m\_Window). After this, glew is then initialised using glewInit() and a quick error check is performed to ensure glew initialised correctly by checking the status of the initialisation. After this, m\_isClosed is set to false as the window is not closed as it has now been constructed, a few OpenGL functions are used after this to enable depth tests and face culling to help with the drawing of objects on the SDL window.

The methods in this class include the Clear method, the SwapBuffers method and the IsClosed method. The Clear method returns void and takes in four floats that represent a red colour value, a green colour value, a blue colour value and the alpha value. The Clear method is used to clear the colour that exists on the screen and to clear the colour buffer and the depth buffer that is enabled in the constructor. The IsClosed method returns a bool and takes in void, this is only used to return the status of m\_isClosed to tell if the window is closed or not. The SwapBuffers method takes in and returns void, it is used to swap the windows as a buffer window is being used, it also checks if the SDL window has been closed by checking all SDL events and if any are equal to SDL\_QUIT then m\_isClosed is set equal to true.

The display class destructor (~Display()) method is used to clear any data that was created or saved to a variable from the constructor class, this includes deleting the context that was created (m\_glContext), destroying the window (m\_Window) and finally quitting out of SDL.

# 1.3 Shader Class

The shader class handles retrieving the shader files from resources and then compiling and running them, it also allows us to bind the shaders created to an object/mesh that exists within the project to ensure this is what the GPU is using to draw the object/mesh. A GLuint was created to save data on the current program, two GLuint arrays were also created to hold the number of shaders (m\_shaders) and the number of uniforms (m\_uniforms). The number of shaders is set to two as this program includes a vertex shader and fragment shader, this set-up allows for other shader types to be loaded such as other vertex/fragment shaders or a geometry shader.

The constructor for this class takes in a string, this string is used to find where the external shaders are in the project files. The first thing done is to create space for the program, this is done by setting m\_program equal to the glCreateProgram() function, after this the two shaders that are in the resource folder (res) are loaded up, OpenGL is told the shader type and they are sent to the m\_shaders array. A for loop is then used to attach all loaded shaders to the program using the glAttachShader() function and passing in the program (m\_program) and the shader (m\_shader[i]). After this a few glBindAttribLocation functions are called, these tell the shader there it is mapping the attribute. After this, two error checks are carried out using glLinkProgram() and glValidateProgram() and passing in the current program (m\_program), these check for any errors in the linking of the shaders and check the validity of the shaders that are now attached to the program, the final thing handled in the constructor is to store the location of the transforms of objects passed through the shader and save them to the array m\_uniforms, this will allow us access to the transform of objects later on.

The two error checking functions are helper functions that were taken from a tutorial series on youtube[1], LoadShader() is used to load a text file from the hard drive and CheckShaderError() checks for errors in the shader and generates any error messages that arise from reading the shaders.

The BindShader() method is a simple method that takes in and returns void, it is used to bind any shaders created in main.cpp to the program using the glUseProgram() function and passing in the current program (m\_program). The Update method takes in a reference to a transform and a camera, this method is used to change the m\_uniforms matrix incase and models change their position. This is done by constructing a model matrix (model) by multiplying the projection matrix from the camera and the transform of the model together. This is saved as a mat4 called model. This model matrix is then used to update the m\_uniforms list by using the glUniformMatrix4fv() function. The last method in the class other than the destructor is the CreateShader method, this takes in the file name for the shader and the shader type and returns the completed shader, the first thing this method does is create the shader depending on the shader type it has been passed by using the glCreateShader function and passing in the shader type from earlier, this shader is saved to a GLuint called “shader”. A check is then made to ensure the shader has actually been loaded by ensuring that the variable shader does not equal zero, this ensures that the shader has been created correctly. After this is done the strings from the shader that has been loaded are saved and the length of the string is also saved, this string is then translated to C as this is the shader language of OpenGL, the length of this new translated string is then also saved. This new string is then sent to OpenGL using the glShaderSource() function, this has the shader, the number of source strings, the source strings and the length of the source strings passed into it. The function glCompileShader() is then used with the shader (shader) being passed into it, this then compiles the C strings to complete the building of the shader. An error check is then made to ensure the shader compiled correctly and if this passes the shader is returned.

The destructor for the shader class (~Shader) simply detaches any shaders attached to the program and deletes them by use of a for loop and glDetachShader and glDeleteShader functions. After this the overall shader program is deleted by use of the glDeleteProgram() function and passing in the created shader program (m\_program).

# 1.4 Shaders

# 1.4.1 Vertex Shader

The vertex shader (Figure 1) is set up to take in data using the attribute key word and save this for use within the shader, the three attributes are: position, texCoord and normal, these store the position of a vertex, the texture coordinate of a vertex and the normal of a vertex respectively. The varying key word allows data to be passed between the shaders, the texCoord0 and normal0 are created to be passed to the fragment shader. The uniform key word is used to ensure the transform matrix that comes after it is not changed as the shader program executes.

Within the main function, the position of a vertex is set to gl\_Position and is worked out by multiplying the position of the vertex with the transform matrix. The next line simply sets the texture coordinated to a varying value so they can be passed to the fragment shader, the last line does the same thing but swizzling is used to ensure the vec4 can be saved as a vec3.

# 1.4.2 Fragment Shader

The fragment shader (Figure 2) has a uniform sampler2D which is the diffuse of an object, the two following variables are the varying variables that are passed in from the vertex shader.

Within the fragment shaders main function, it determines the colour of a fragment of pixels which is saved as gl\_FragColor, this is worked out by getting the texture on the pixel fragment and applying it to the color saved. Lambert lighting is also added here by multiplying the colour of a fragment by the dot product of the negative direction of the light and the normal of the vertex the fragment is from. This whole sum is then clamped using the clamp() function to ensure the objects are never too dark that you cannot see them as it is my personal preference.

# 1.5 Mesh Class

Within the mesh class a vertex class (Vert) exists, this class is used to store information on any vertices and provide functionality needed for the mesh class. The vertex class stores the position, texture coordinates and normal of a given vertex, this is done by setting the variables with “glm::vec3 position;” etc. After the variables are set they are then added to the vertex class by passing the values into the Vert method and setting them to their respective values within the vertex class using the setters “this->position = position;” etc. for each variable that has been passed in. The last thing the vertex class holds is a getter method for each variable type, this is achieved with the code “inline glm::vec3\* GetPosition() { return &position; }”. A getter is constructed for each variable in the class once again.

The variables within the Mesh class include two GLuints, these are m\_vertArrayObject (VAO) and m\_vertArrayBuffer (VAB), the VAO is used to store mesh data on the GPU while the VAB is used to allow access and reference to each buffer created. The unsigned int “m\_drawCount” is used as a means to store the number of indices in a loaded mesh and tell OpenGL how much of the VAO should be drawn.

There are two mesh constructors as method overloading is used to allow for more methods of creating meshes, the first mesh constructor simply takes in a file name and is used to create a mesh for a model from a .obj file. It works by using a model loading method from the script “obj\_loader.h”, this script is talked about later. A model is created and the .obj data is added to the model by using the OBJModel(objModel).ToIndexedModel() function, this translates the information from the .obj file into something the mesh initialiser can understand, the model data is then passed into the InitModel method. The second mesh constructor takes in vertexes and the number of vertexes, it also takes in the indices and the number of indices. This constructor allows for manual drawing of vertices instead of loading an .obj file. This is done by passing in the required variables that are then stored on the model using two for loops and then the data is sent to the mesh initialisation method.

The InitMesh method is a method that returns void and takes in the model data using the “model” variable, at the start of this method the number of indices is saved from the model that is passed in onto the m\_drawCount integer. This is done by use of the model.indices.size() function which returns the number of indices on the model. After this the space required for the VAO is allocated by using the glGenVertexArrays() function and passing in the VAO, after this the glBindVertexArray() function is called and the VAO is passed in, this is to ensure the VAO is treated as a vertex array and will be affected by any changes that should affect a vertex array. After this the buffers that will be used to write data to are allocated with use of the glGenBuffers() function and passing in the NUM\_BUFFERS enumerator. The VAB is then passed into the glBindBuffer() function along with the buffer type of the VAB so OpenGL knows how to handle the VAB properly, this function ensures that any function that affects buffers will now affect the VAB. After this the buffers created have data written to them which is stored on the GPU by use of the glBufferData() function that once again takes in the buffer type and also takes in the number of model positions and the model positions array along with how to draw the data, I used GL\_STATIC\_DRAW as I wanted to only write data, not to fetch it and re-write it so using GL\_STATIC\_DRAW will ensure a high level of optimisation. Once all the data is written to the buffer the rest of the method, excluding the last line, handles how to access the data within the buffer and where each data type is stored within the buffer, this is done by enabling a vertex attribute array for each variable within the buffer using glEnableVertexAttribArray() and then using a pointer to point to where the data for this attribute array is stored. This is done by using the glVertexAttribPointer() and passing in the attribute array number and the length of the data, it then takes in the data type, the last three variables that are passed in are not relevant to the data type we are reading and are set to GL\_FALSE, 0 and 0 respectively. This is repeated for each data type that is stored on the buffer, in this case there is three data types, the texture coordinates, the normal and the indexes. The last line of the InitMesh method is to stop the VAO from being affected by vertex array functions and is done by simply passing a value of zero into the glBindVertexArray() function. The DrawMesh function is used to draw a mesh with the GPU, this method takes in and returns void. First, the VAO is bound to a vertex array again using the same function as before, then OpenGL is told how to draw the mesh using the glDrawElements() function and passing in the type of data to draw, the draw count and a data type with a value of zero being passed in lastly. After this the VAO is unbound using the previous method.

Finally the mesh destructor (~Mesh) simply deletes the VAO created in the constructor, this is done using the glDeleteVertexArrays() function and passing in the number of VAOs and the VAO to delete.

# 1.6 Texture Class

The texture class is used to load a texture and handles how to apply the texture to an object, also handles saving the texture for OpenGL to access. A GLuint is created to hold the texture data, this is called “m\_texture”. The texture class relies on functions from “stb\_image.h” which is an image loading script that is talked about later.

The texture class constructor (Texture) takes in a string that is used to find a file. Three integer variables are created at the start which are width, height and numComponents. The texture data is then loaded by using a function from the texture loading script called “stbi\_load()”, the file name is converted to C and passed in, then a reference to the width, height and number of components is passed in, finally it is passed the amount of components it requires. An error check is then performed to ensure a texture was loaded by ensuring imageData does not equal null. The texture is then created and OpenGL is told to treat the texture as a texture, this is done in a similar fashion to the VAO, first glGenTextures() is called and the number of textures and texture are passed in, after this glBindTexture() is called with the texture type and texture being passed in so OpenGL knows the texture type of the texture. After this glTexParameteri() is called twice to tell OpenGL how to handle if the object is out with the texture height or width. This is passed the texture type, if it is handling the height or width and how to handle the texture. After this glTexParameterf() is called to handle how the texture will be interpolated and extrapolated, this is again passed in the texture type and the filter on minification/magnification and how to handle this. Finally the texture is sent to the GPU using glTexImage2D() and passing in the texture type, the level, the internal format of the texture (in our case RGBA), the width and height of the texture, the border size, the format of the texture, the data type and the number of pixels. After the texture has been sent to the GPU it is then deleted using the stbi\_image\_free() function and passing in the image data (imageData).

The BindTexture method returns a void and takes in an unsigned integer specified as unit, first a check is performed to ensure the unit is within the range of allocation for OpenGL by using the assert() function and passing in a check to ensure unit is between zero and thirty one. After this the active texture unit is allocated using glActiveTexture and passed GL\_TEXTURE0 + unit, this is to allow the saving of one or more texture easily, after this the texture is bound to m\_texture by using the glBindTexture function and passing in the texture type and m\_texture.

The destructor for the texture class (~Texture) simply calls glDeleteTextures() and passes in the size of the texture and the address of the texture to delete the texture that is saved to m\_texture.

# 

# 1.7 transform.h

The transform header file is used to apply movement to any models that exist within the world space of the program, this is done using vectors as it is much easier to apply inputs to vectors than to matrices, the vectors are turned into matrices when necessary.

The constructor for this class is used to initialise the variables of the model that is being transformed, it is passed the position, rotation and scale of the model and default values are also set for each, once these values have been passed in they are set in the transform constructor to allow for the creation of getters and setters. The GetModel matrix is created to change the position, rotation and scale vectors into a matrix and multiply each together to return a model matrix. This is exactly what it does by changing each vector into its respective matrix and then first creating the full rotation matrix before using it to create the model matrix and returning this value. The getters simply grab the data that is stored and allow it to be manipulated passively while the setters allow a value to be passed in to instantly manipulate the respective vector. These functions combined allow any models transform to be manipulated during runtime.

# 1.8 camera.h

The camera header allows for the creation of a camera that will allow for several view types to be used, the perspective view type is used in this case. The constructor for the camera takes in a position which is a vec3 and four floats that are the field of view, aspect ratio and near and far clipping planes. When these variables are passed in they are then used to create a perspective camera using the glm::perspective() function and passing in the previous variables, the position of the camera is then set using the vec3 position and the camera is then told which way is forward and which way is up by using vec3s and saving them as m\_forward and m\_up. In this case (0,0,1) is the forward direction and (0,1,0) is the up direction.

The other method in the header is the GetViewProjection() function which is used to create the projection matrix and return it. This is done by multiplying the perspective matrix with the view matrix. This was done to make setting transforms easier as they would take in vec3s rather than confusing matrices and this method could be called whenever the projection matrix was needed.

# 

# 1.9 main.cpp

Finally, the main function uses all the previous classes and scripts to easily create the window and texture and render objects etc. Firstly the width and height are defined to allow them to change easily. After this the main method is called which takes in a void and returns a variable, firstly the display class constructor (Display) is called to create a display, the width, height and title of the window are passed in. After this the mesh constructor is called three times to create the three meshes, each time the location of the .obj file is passed in. The shaders are then created using the shader constructor and passing in the file locations. The texture constructor is then called and the location of the textures is passed in, this constructs the textures. Then the camera is created using the camera class constructor and passing in the cameras position, the FOV, the aspect ratio and the near and far clipping planes. The transforms are then created using the transform constructor, each transform will default to the default values given in “transform.h” and a float “counter” is constructed to manipulate the transforms.

Once this is finished a while loop is entered while the display is not closed by using the display.IsClosed() method. In this while loop the display is cleared using display.Clear() and passing in the rgba values. The counter is then manipulated and saved in different ways to allow for transform manipulation effects. After this the transform of each object is then set by using the getters and setters created in “transform.h”. The objects are then rendered in order using the shader methods created within the shader class. The final things are to swap the display and update it using the SwapBuffer() method and to increase the counter. This while loop is called once every frame.

After the while loop there is a return type of zero, this is because an integer must be returned from the main function. When the program is now run it displays the window seen in figure 3.

# 2.0 External scripts

The two scripts that were loaded in to help with the creation of the program were obj\_loader.h/.cpp and stb\_image.h/.cpp, you can find the links to these files in the references. The obj\_loader was used to help with the loading of objects from an .obj format. The stb\_image class was used to help with the loading of textures from several file types, all textures used in the program were, however, .png.

# 3.0 Figures

Figure 1:-

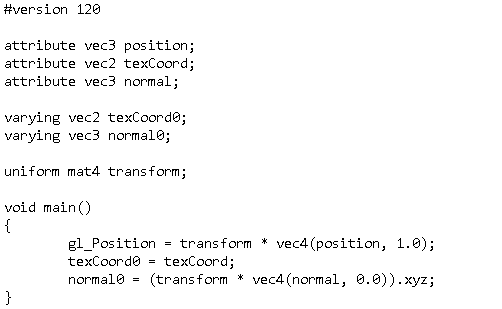


Figure 2:-

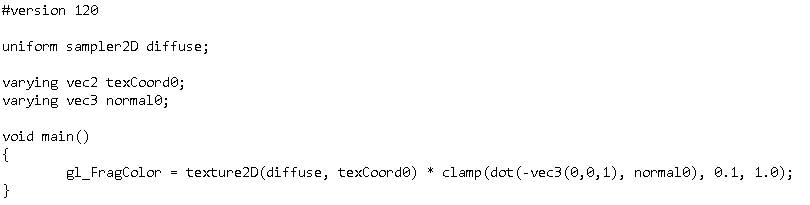
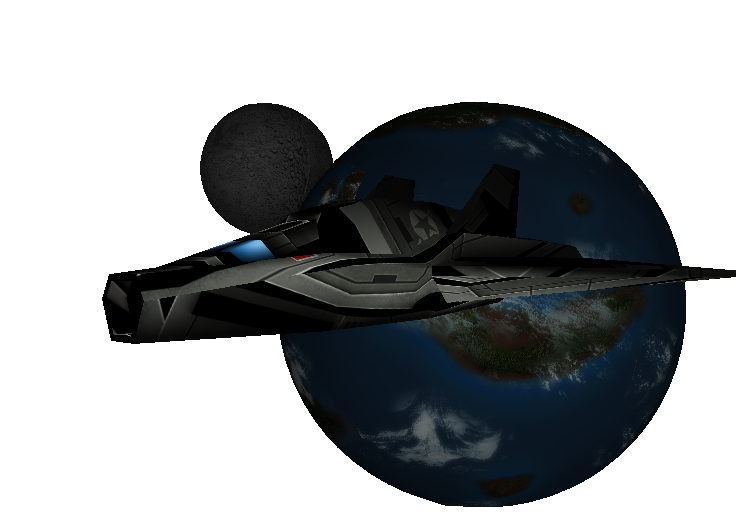


Figure 3:-



# 4.0 References

[1] - <https://www.youtube.com/watch?v=csKrVBWCItc>

# 5.0 Bibliography

Tutorials:

<https://www.youtube.com/watch?v=ftiKrP3gW3k&index=1&list=PLEETnX-uPtBXT9T-hD0Bj31DSnwio-ywh>

<https://www.youtube.com/watch?v=FxCC9Ces1Yg&list=PLSPw4ASQYyymu3PfG9gxywSPghnSMiOAW>

<https://www.youtube.com/watch?v=AvrjQtFBJvk>

obj\_loader and stb\_image:

<https://github.com/BennyQBD/ModernOpenGLTutorial/blob/master/obj_loader.h>

<https://github.com/BennyQBD/ModernOpenGLTutorial/blob/master/obj_loader.cpp>

<https://github.com/BennyQBD/ModernOpenGLTutorial/blob/master/stb_image.h>

<https://github.com/BennyQBD/ModernOpenGLTutorial/blob/master/stb_image.c>

.obj files and textures:

<https://www.turbosquid.com/FullPreview/Index.cfm/ID/797594>

<https://www.turbosquid.com/FullPreview/Index.cfm/ID/588767>

<https://www.turbosquid.com/FullPreview/Index.cfm/ID/553741>