Graphics Programming

M3I622944

Iain Cotter

S1507974

***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.***

*Signature:*

**Iain Cotter**

Table of Contents

[1 - Background Information 3](#_Toc514252801)

[1.1 - Overview 3](#_Toc514252802)

[1.2 - User Inputs 3](#_Toc514252803)

[1.2.1 – Controls 3](#_Toc514252804)

[1.2.2 – Invisible 4](#_Toc514252805)

[1.2.3 – Visible Normals 4](#_Toc514252806)

[2 - Initialising Shaders 5](#_Toc514252807)

[2.1 – Loading Shader Program 5](#_Toc514252808)

[2.1.1 – Call to Initialise Shader 5](#_Toc514252809)

[2.1.2 – Creation and Compilation 5](#_Toc514252810)

[2.1.3 – Creating Individual Shader 6](#_Toc514252811)

[2.1.4 – Shader Loading 6](#_Toc514252812)

[2.2 – Using Shader in the Scene 6](#_Toc514252813)

[2.2.1 – Bind Shader 6](#_Toc514252814)

[2.2.2 – Set Shader Values 7](#_Toc514252815)

[2.2.3 – Set Transform Uniforms 7](#_Toc514252816)

[2.2.4 – Drawing to Scene 8](#_Toc514252817)

[3 – Shader Effects 8](#_Toc514252818)

[3.1 – Skybox 8](#_Toc514252819)

[3.1.1 – Set Cubemap Texture 8](#_Toc514252820)

[3.1.2 – Set Cubemap Vertices 9](#_Toc514252821)

[3.1.3 – Skybox Shader 9](#_Toc514252822)

[3.2 – Environment Mapping 10](#_Toc514252823)

[3.2.1 – Reflection Shader 10](#_Toc514252824)

[3.2.2 – Refraction Shader 11](#_Toc514252825)

[3.3 - Overlay Shader 12](#_Toc514252826)

[3.4 - Geometry Shaders 13](#_Toc514252827)

[3.4.1 – Explosion Shader 13](#_Toc514252828)

[3.4.2 – Visible Normals Shader 14](#_Toc514252829)

[3.5 - Fragment Shaders 15](#_Toc514252830)

[3.5.1 – Toon Rim Shader 15](#_Toc514252831)

[3.5.2 – Fog Toon Rim Shader 16](#_Toc514252832)

[1 - Appendex 17](#_Toc514252833)

# 1 - Background Information

## 1.1 - Overview

The scene consists of four models: plane and 3 birds (See figure 1). 

Figure 1 - Scene

The Birds travel towards either engine and if they successfully reach any engine then a life is depleted from the bottom left corner (the Hud in figure 2). If health drops below 1 then game loop will quit – there is a check in console purely for debugging purposes to stop inputs of keys affecting code when suddenly quit.



Figure 2 - health

## 1.2 - User Inputs

### 1.2.1 – Controls

The user manipulates planes transform rotation to avoid incoming birds.

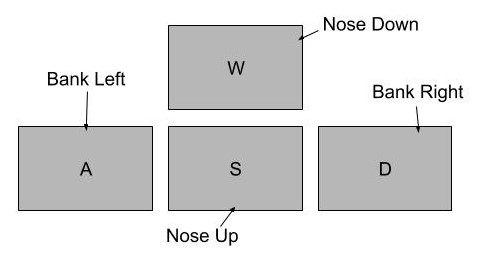


Figure 3 - Controls

The plane controls work off the standard ‘wasd’ format. The vertical controls have been inversed.

### 1.2.2 – Invisible

The user can use spacebar to change normal shader (figure 4) to a shader that calculates how light passes through object and how to texture mesh accordingly by considering environment map (see section 3.2) .

Figure 4 – Rim Toon (Left)

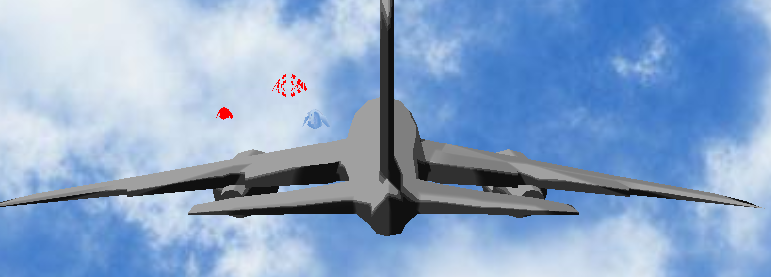


Figure 5 – Refract (right)

This has a mechanical feature as while ‘invisible’ the birds that spawn will not travel towards the plane engines. As target is set to random position behind the plane (figure 6).

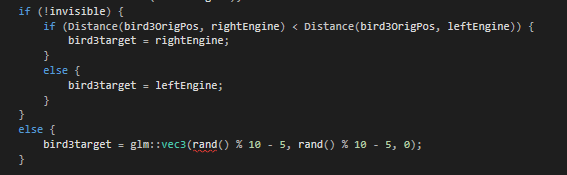


Figure 6 – Bird spawn

### 1.2.3 – Visible Normals

The final input is purely cosmetic. The reflection shader uses the same environment mapping method as seen in figure , once the ‘Left Shift’ button is pressed the normals will be visible.



Figure 9 – Normals Visible

Figure 8 - Reflect

# 2 - Initialising Shaders

## 2.1 – Loading Shader Program

### 2.1.1 – Call to Initialise Shader

In this project there are two methods for loading shader: one loads both vertex and fragment shader based on their filename, the other loads vertex, fragment and geometry shader.



Figure 10 – shader programs being Initialised in TheGame.cpp

The code (Figure 2.1) runs the functions within “Shader.cpp” as seen in figure 2.2 (“InitialiseShader” does not load a geometry shader that is only difference).

### 2.1.2 – Creation and Compilation

Figure 11 – Shader program being loaded and created

The specific shader within the shader program are created using “ShaderCreation” (figure 2.3). That creates shader loads data as a string value based on what is returned from “ShaderLoad” (figure 2.4). It also gets memory location for the uniform transform in shader program – so it can be set later.

### 2.1.3 – Creating Individual Shader

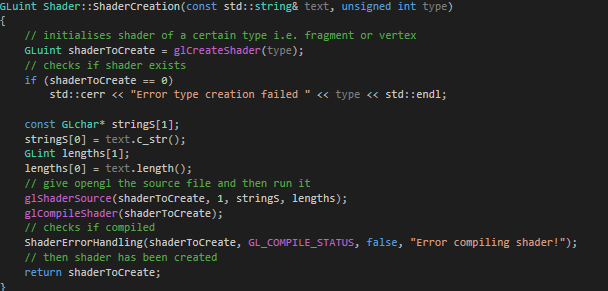


Figure 12 – Individual Shader being created

Creates a shader of specified type (Vertex, Fragment and Geometry are used in this project). Gets the data read by “LoadShader” and compiles the new shader. Finally returning the created shader as a ‘GLuint’ (see section 2.1.2).

### 2.1.4 – Shader Loading

Figure 13 – Reading shader file

This reads the shader file line by line and adds it to overall string ‘output’ and once finished return the outputted string as data to feed “ShaderCreation” (see section 2.1.3).

## 2.2 – Using Shader in the Scene

### 2.2.1 – Bind Shader



Figure 14 – Shader to use

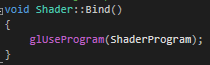
This method call is to the shader of this instance (different shaders need separate calls), that tells openGL to use that shader program (i.e. draw model using that shader effect).

Figure 15 – State to OpenGl what shader to use

### 2.2.2 – Set Shader Values

After binding certain uniform values need to be inserted into shader program for shader toi execute correctly.

Figure 16 – Call in TheGame.cpp

A shader needs set of inputs that in this project have been set as uniforms.

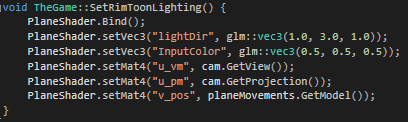


Figure 17 – Call to set Uniform

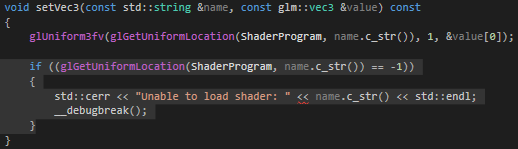
The Setting of value runs through “Shader.h” (figure 2.8) – different functions for different types of data: floats, vectors, matrices etc. Values can be set at run time which gives somewhat realistic effects.

Figure 18 – Sets Uniform or returns break point

### 2.2.3 – Set Transform Uniforms

In some of the shaders a transform uniform is required to update the change in model position visually (other shaders take in model view projection matrix, or some variant, directly as uniform set in section 2.2.2 to calculate the position on screen). The models transform is passed to “Shader.cpp”.

Figure 19 – Update transform in shader call

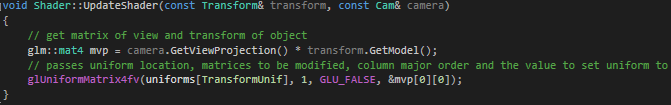


Figure 20 – updating transform in shader

The code above (figure 20) merges model matrix (position, rotation and scale) with view projection matrix to create Model View Projection matrix that is then passed into shader. This positions model onscreen depending on model position, rotation and scale and applies the perspective squishing onto 2D screen.

### 2.2.4 – Drawing to Scene

Once texture has been bound and values set successfully then model is ready to be drawn to scene.

Method call to “mesh.cpp”. That draws the mesh as a series of triangle using the bound shader to apply effect.

Figure 21 – Draw Model

# 3 – Shader Effects

## 3.1 – Skybox

### 3.1.1 – Set Cubemap Texture

The first step in creating a skybox is loading and setting texture to a cube. This is done in the initialise method, so it is run once and avoids system crashing.



Figure 22 – Call to Set Cubemap Texture

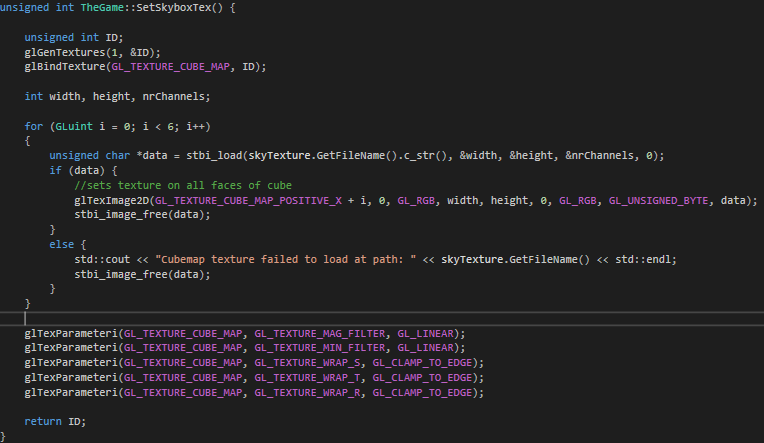


Figure 23 – Set Cubemap texture

The code above generates and binds texture to location. The texture is then loaded as a char (this is done in for loop because ordinarily there would be 6 sides to texture however this does not apply to this scene), each face of cube is assigned a texture. The textures are wrapped to faces to stop stretching or tiling of texture.

### 3.1.2 – Set Cubemap Vertices

After cubemap has been assigned texture then faces of the cubemap are made using a set of vertices accessible in “TheGame.h”.

Figure 26 - Skybox

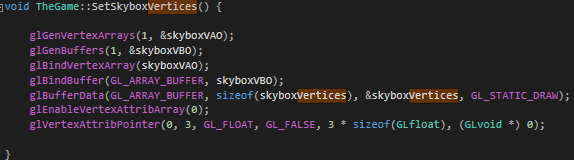
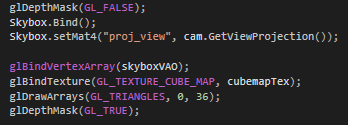


Figure 24 – Set vertices of the cubemap

### 3.1.3 – Skybox Shader

The shader loading, and binding uses the steps in sections 2.1.1 and 2.2.1.

Figure 25 – Draw skybox



Depth mask is false so that skybox will always be behind other models. Shader takes in view projection matrix. The skybox is drawn with texture using triangle to render.

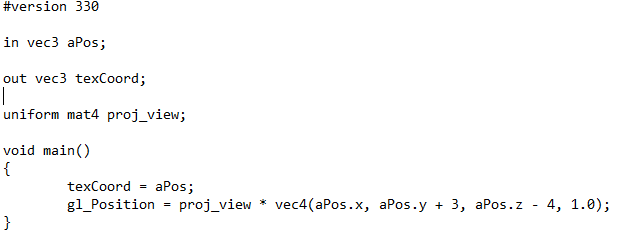


Figure 25 – Vertex Shader

The position of each is offset to put the skybox in screen view and not off in distance.

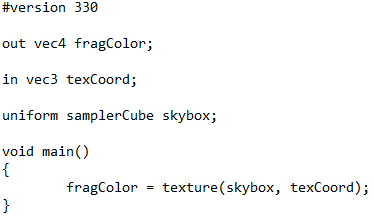


Figure 26 – Fragment Shader

Each fragment is given color value based on texture passed in, texture coordinate relates to point on texture and color is whatever that points color is.

## 3.2 – Environment Mapping

Now that cubemap has been used other effects come easily as using environment mapping techniques: Reflection and refraction specifically.

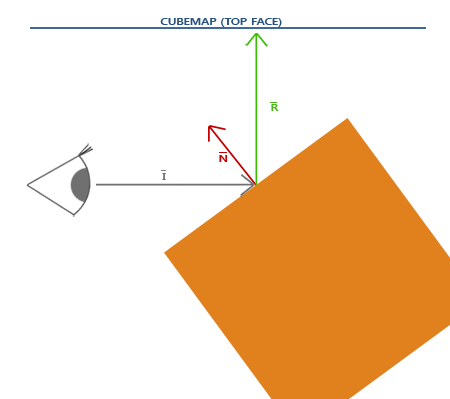


Figure 27 - Reflectance

The principle behind reflection is that using normal of surface and angle of incidence, perfect reflector can be calculated. Where perfect reflector meets cubemap that is point to display on object.

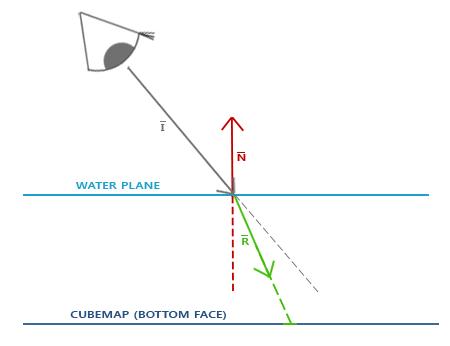


Figure 28 - Refractance

Refraction is similar to reflection it uses the surface normal and angle of incidence to calculate refractor as it passes through an object. That point where refractor meets cubemap is point to display object.

### 3.2.1 – Reflection Shader



Figure 29 – Reflection Shader

The shader loading, and binding uses the steps in sections 2.1.1 and 2.2.1.

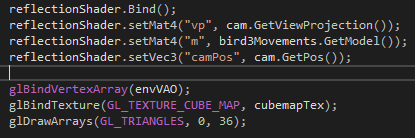


Figure 30 – Set uniforms and draw the vertices with texture

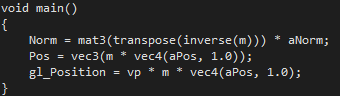


Figure 31 – vertex Shader

‘Pos’ is passed through to fragment to calculate view direction. Then vertex position is placed based on Model View Projection Matrix and initial position of vertex.

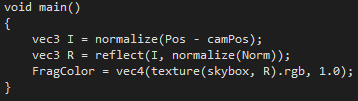


Figure 32- fragment Shader

The ‘Pos’ of model allow openGL to calculate incidence, with this using ‘Norm’ find reflector.

The color can then be found by finding texture colour at texture coordinate of reflector (vector3).

### 3.2.2 – Refraction Shader

Figure 33 - Refractance



The shader loading, and binding uses the steps in sections 2.1.1 and 2.2.1.

The refractance shader is very similar to section 3.2.1 the key difference being fragment shader takes in ratio of material (figure 35 has ratio of a diamond), GLSL uses its own refract method to pass refractor through material. The color output is cubemap texture and texture coordinate of refractor vector 3.

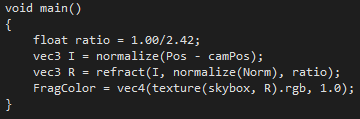


Figure 35 – Fragment Shader

## 3.3 - Overlay Shader



Figure 36 – Hud at bottom left of screen

The shader loading, and binding uses the steps in sections 2.1.1 and 2.2.1.

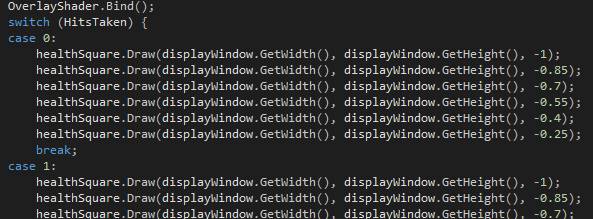
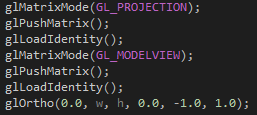


Figure 37- draw the quads

There are 6 quads drawn to screen, each applied offset on x so don’t draw on top of one another. In order to render a 2D overlay on a 3D scene. The 3D scene should be drawn, the projection will need to be swapped to 2D (saving 3D projection), then drawing 2D element and finally loading 3D saved projection.

Figure 38 – Switching to 2D projection

The projection is switched from 3D to 2D while saving 3D projection data.

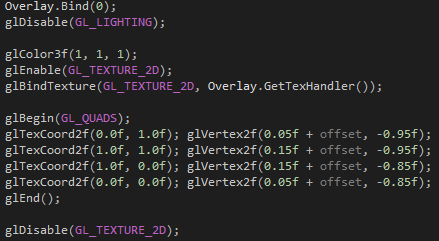


Figure 39 – Draw Quad Method

This draws a quad at lower left of screen

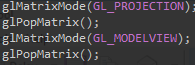


Figure 40 – Switching back

## 3.4 - Geometry Shaders

When initialising geometry shader it makes use of “LoadThreeShaders” from section 2.1.1.

### 3.4.1 – Explosion Shader



Figure 41

The shader loading, and binding uses the steps in sections 2.1.1 and 2.2.1.

The explosion uses the fog rim toon vertex and fragment shaders in section 3.5.2.

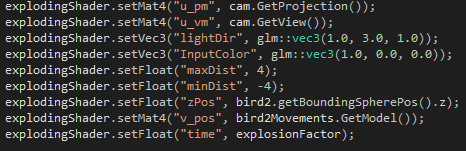


Figure 42 – Pass to geometry shader

This is float value to pass to the geometry shader which is counter value that is ever increasing.

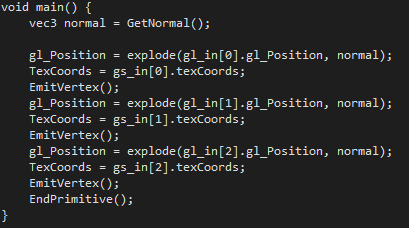


Figure 43 – change positions of vertices

The normal (perpendicular direction of the primitive) is calculate in figure 44. Then each vertex position is changed along the normal (figure 45).

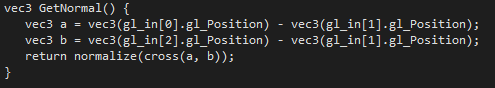


Figure 44 – Normal of Primitive

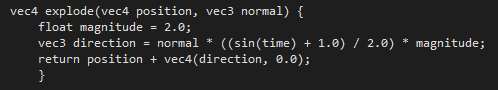


Figure 45 – Position

The Position changes as ‘explosionFactor’ counter that is linked to ‘Time’ uniform increases the sin calculation makes the model explode in and out.

### 3.4.2 – Visible Normals Shader



Figure 46 – Visible Normals

The shader loading, and binding uses the steps in sections 2.1.1 and 2.2.1.

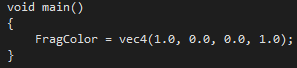


Figure 48 – Set colour of normals

The colour has been set to red to match other models.

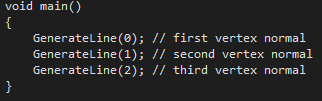


Figure 49 – Create Lines

The lines are created at each vertex position in by vertex shader.

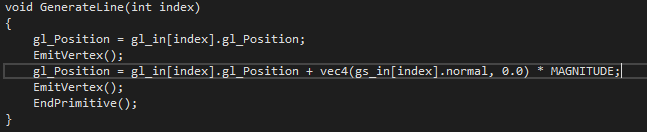


Figure 50 – Generation of line

This starts the line at original position passed in by vertex shader, ends at point along the normal of vertex times by scalar value (‘MAGNITUDE’ is 0.4).

## 3.5 - Fragment Shaders

### 3.5.1 – Toon Rim Shader



Figure 51 – The plane Banking left



Figure 52 - The plane Banking right

The shader loading, and binding uses the steps in sections 2.1.1 and 2.2.1.

As the user enters the inputs (see section 1.2.1) then the ‘v\_pos’ changes then the toon rim intensity changes with it. This gives the effect that plane is turning into the suns direction.

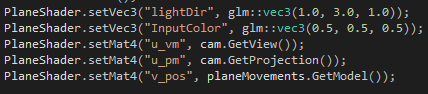


Figure 53 – Set Uniform Values

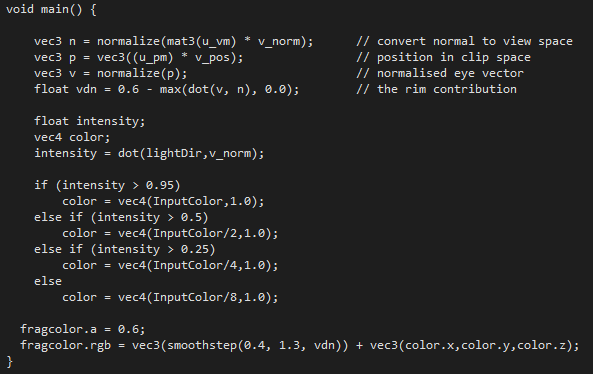


Figure 55 – Fragment shader

This calculates input colour based on the dot product between the light direction set as uniform

### 3.5.2 – Fog Toon Rim Shader

Figure 56 - Fog

The shader loading, and binding uses the steps in sections 2.1.1 and 2.2.1.

The detail has left the model but will become more visible as model gets closer.

Fragment shader is different from rim toon as it calculates fog ‘factor’ as well and mixes the colours (figure 59).

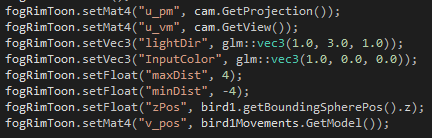


Figure 57 – New Uniform values

New values ’maxDist’ and ‘minDist’ for calculating the ‘factor’.

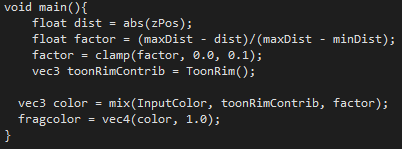


Figure 59 – fragment Shader

# 1 - Appendex

Vries, J. d., 2014. *Cubemaps.* [Online]   
Available at: https://learnopengl.com/Advanced-OpenGL/Cubemaps  
[Accessed 16 May 2018].

Vries, J. d., 2014. *Geometry Shader.* [Online]   
Available at: https://learnopengl.com/Advanced-OpenGL/Geometry-Shader  
[Accessed 16 May 2018].