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](#_Toc514215993)

[1.1 - Overview 3](#_Toc514215994)

[1.2 - User Inputs 3](#_Toc514215995)

[2 - Initialising Shaders 5](#_Toc514215996)

[2.1 – Loading Shader Program 5](#_Toc514215997)

[2.1.1 – Call to Initialise Shader 5](#_Toc514215998)

[2.1.2 – Creation and Compilation 5](#_Toc514215999)

[2.1.3 – Creating Individual Shader 6](#_Toc514216000)

[2.1.4 – Shader loading 6](#_Toc514216001)

[2.2 – Using Shader in the Scene 7](#_Toc514216002)

[2.2.1 – Bind Shader 7](#_Toc514216003)

[2.2.2 – Set Shader Values 7](#_Toc514216004)

[2.3 – Set Transform Uniforms 8](#_Toc514216005)

[3 – Shader Effects 8](#_Toc514216006)

[3.1 – Skybox 8](#_Toc514216007)

[3.1.1 – Set Cubemap Texture 8](#_Toc514216008)

[3.1.2 – Set Cubemap Vertices 9](#_Toc514216009)

[3.1.3 – Shader 9](#_Toc514216010)

# 1 - Background Information

## 1.1 - Overview

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

Figure 1.. - Scene

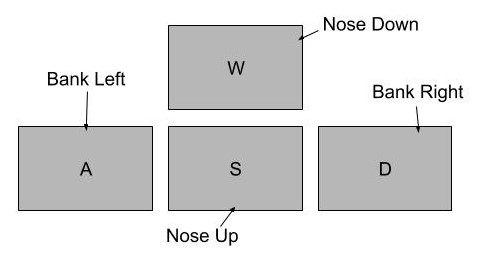
The Birds travel towards either engine and if they successfully reach any engine then a life is depleted from the bottom left corner (the Hud in figure 1-2). If health drops below 1 then game loop will quit – there is a check in console purely for debugging purposes to stop inputs of keys affecting code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



Figure 1.. - Health

## 1.2 - User Inputs

The user manipulates planes transform rotation to avoid incoming birds.

Figure 1.. - Controls

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

The user can use spacebar to change normal shader (figure 1-4) to a shader that calculates how light passes through object and how to texture mesh accordingly by considering environment map .

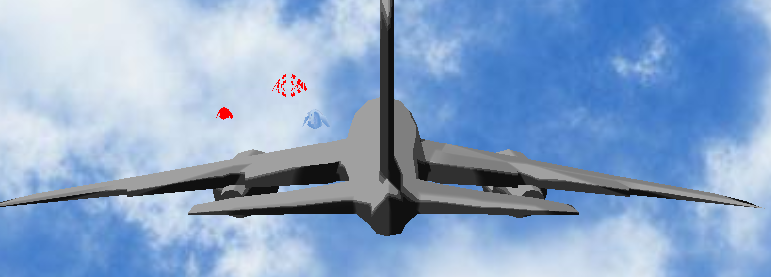
Figure 1.4 – Plane Normal Shader (left)

Figure 1.5 – Plane Refraction Shader (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 1-6).

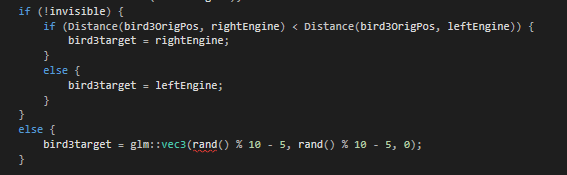


Figure 1.6 – Bird Spawn Snippet

The final input is purely cosmetic. The reflection shader uses the same environment mapping method as seen in figure 1-5, once the ‘Left Shift’ button is pressed the normals will be visible.

Figure 1.7 – Reflective Shader (Left)

Figure 1.8 – Visual Normal Shader (Right)

# 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 2.. – Shader Loading 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

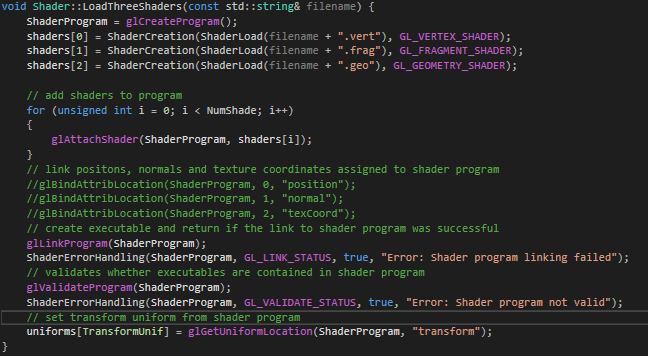


Figure 2.. – Shader program creation

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

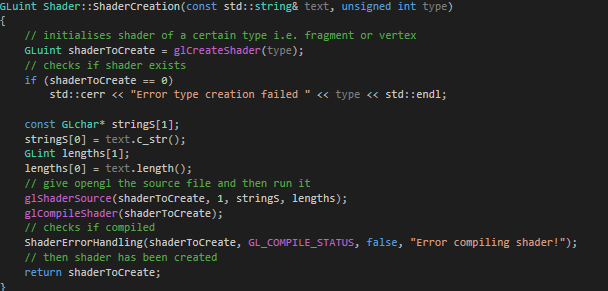


Figure 2.. – Shader Creation

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

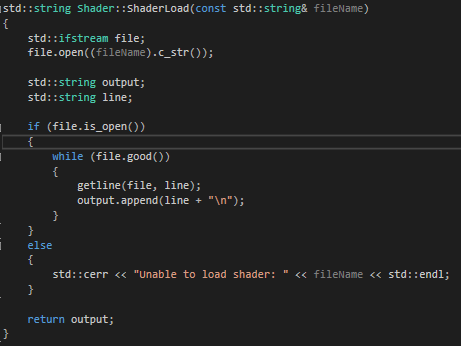


Figure 2.. – Load string from shader file

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

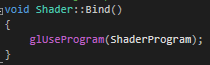
This method call is to the shader of this instance (different shaders need separate calls), that tells openGL to use that shader program (i.e. draw model using that shader effect).

Figure . – Use shader

Figure . – Bind Shader

### 2.2.2 – Set Shader Values

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



Figure . – Set Uniform values

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

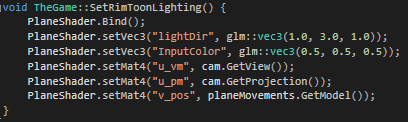


Figure . – Set values in shader

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.

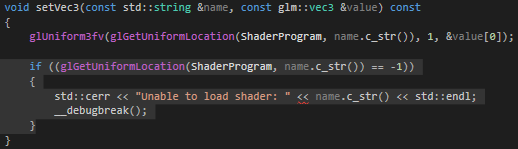


Figure . – Set Uniform values using uniform locations

### 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 . – Models transform and matrix information

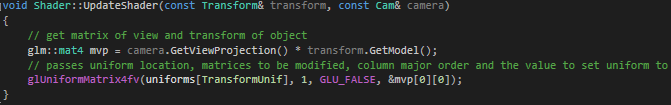


Figure . – Pass MVP to shader

The code in figure 2.11 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.

# 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 . – Set Texture

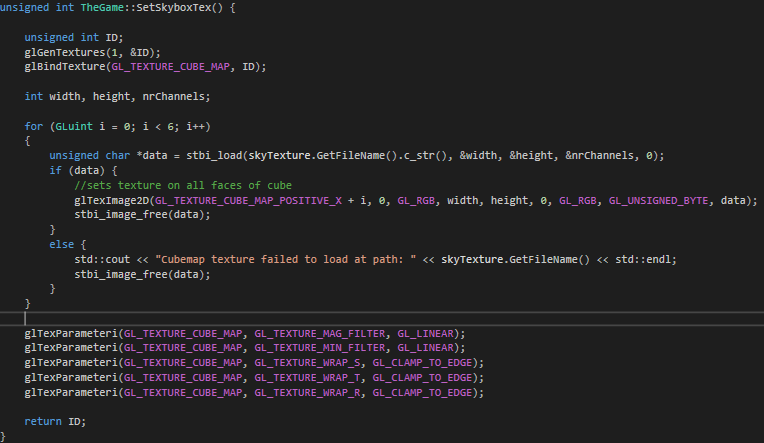
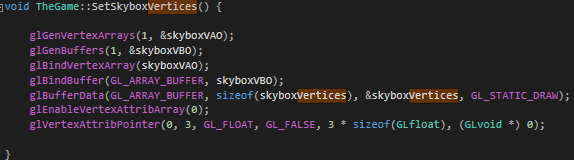


Figure . – Texture setting

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”.



### 3.1.3 – Shader

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

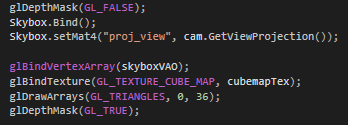


Figure . – Drawing The skybox

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.

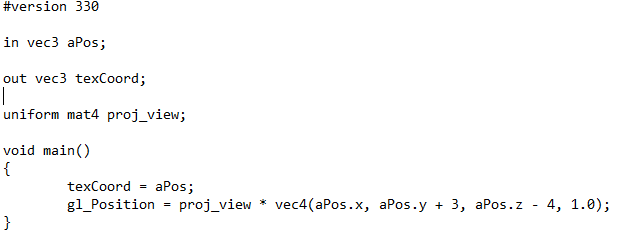


Figure . – Vertex Shader

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

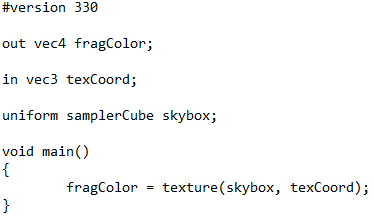


Figure . – Fragment Shader

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.

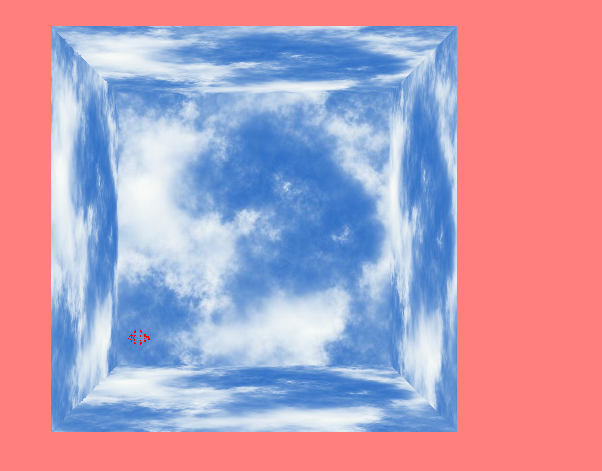


Figure . – Cubemap

## 2.2 – Environment Mapping

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