**Adam S Hosie**

**Student No: S1624519**

**Computer Games (Software Development)**

**Module: Graphics Programming**

**Coursework Documentation**

*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*.

*Adam Hosie*

Contents

[1 Custom Shader Scripts 3](#_Toc8330724)

[1.1 Vertex Shader 3](#_Toc8330725)

[1.2 Fragment Shader 5](#_Toc8330726)

[2 Main Game Script 8](#_Toc8330727)

[2.1 setCustomValues() Method 8](#_Toc8330728)

[2.2 processInput() Method 8](#_Toc8330729)

[2.3 drawGame() Method 10](#_Toc8330730)

[3 References 11](#_Toc8330731)

# 1 Custom Shader Scripts

For the purposes of implementing a shader using GLSL, the use of two types of files known as a vertex shader and a fragment shader must be used. These can be found in the Rendering Pipeline. The vertex shader is used to handle the processing of individual vertices. The fragment shader is used to provide a final colour and texture data. A third middle shader can also be used which is called a geometry shader, this governs the processing of primitives. For the purposes of the shader discussed in this document only a vertex and a fragment shader were used. This shader has the effect of giving the object it is applied to a refractive quality, which allows it to appear transparent, along with this a Rim-Toon lighting effect has been applied, the shader allows for the player to have control over the colour, intensity and contrast, through gamma correction, of the lighting effect. The player also has control over how much refraction is applied. These are all applied through key inputs.

## Vertex Shader

Figure 1A - Vertex Shader


Figure 1 – Vertex Shader

The above image (Figure 1) shows the entirety of the vertex shader used for the generation of the custom graphical technique.

In Line 1, we see the version number used for this shader. In this case we are using the version number 400. Following this we have two layout qualifiers; these are used for accessing the Vertex Position and the Vertex Normal. The first of these, **layout (location = 0) in vec3 VertexPosition;** refers to position of each vertex. The second of these layout qualifiers is used for accessing the vertex normal, this is a value which is stored at each vertex for use in reflection through shaders, this essentially represents the direction in which light would reflect off the surface of each vertex.

Following this, two uniform variables are set. These uniform variables are 4x4 floating point matrices used to represent the model and the associated transform. These are declared in a method inside the **MainGame.cpp** script, this will be covered later. We also pass out two 3D vector variables. These represent the changes we make to the 3D vector variables declared in the layout qualifiers, for use in the fragment shader.

The **main()** method within the vertex shader is used to assign the variables to be passed to the fragment shader with meaningful data.

Firstly we calculate **v\_norm**, to do thiswe first attain a 3x3 matrix from the transpose of the 4x4 model matrix, which is inverted using **inverse(model)**. This transpose causes the new matrix to be created by interchanging each row and the corresponding column, the inversion is done for use in the refraction calculations. This 3x3 matrix is then multiplied by the **VertexNormal** variable in order to convert it to the 3D vector as required.

Following this the **v\_pos** 3D vector variable is calculated through multiplying the 4x4 matrix **model** with the 4D vector of **VertexPosition** contributing to the first 3 dimensions and 1.0 being set as the 4th dimension within the vector.

**gl\_Position** is then set as is necessary for the vertex shader, this variable contains the position of the current vertex. This value is used by fixed functionality operations on primitives after vertex processing has occurred. This is a 4D vector variable which if not written to will not stop the program from running but will not render the object that this shader is attached to. For this instance of **gl\_Position**, we simply multiply the 4x4 matrix **transform** by the same 4D vector we multiplied **model** with for calculation of **v\_pos**.

## Fragment Shader

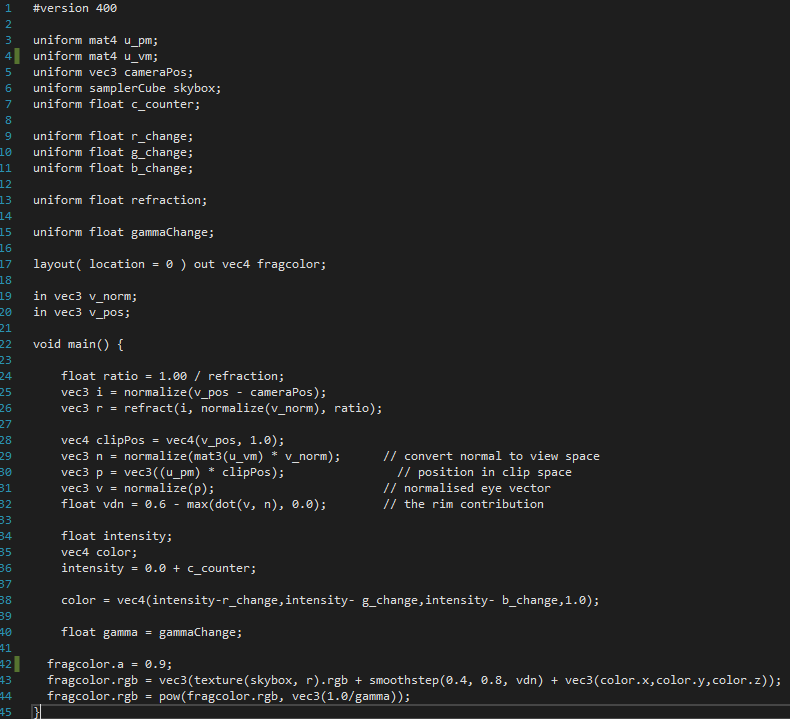


Figure 2 – Fragment Shader

This image (Figure 2) shows the fragment shader used for the implementation of the custom graphical technique. The version number for this shader is the same as that of the vertex shader.

For the purposes of this shader 10 uniform variables have been used, these can be seen from lines 3-15. These uniform variables are set within the **SetCustomValues()** method in the **MainGame.cpp** script. The first two of these uniform variables are 4x4 floating point matrices of **u\_vm** and **u\_pm** which represent the view matrix and the project matrix. Another 3D vector uniform is then to be passed into the shader, this represent the camera position.

The uniform variable skybox represents the skybox object present in the screen; this is used for the reflection calculations.

The remaining uniform variables are all floats, these are used for altering the lighting effects such as the colour, contrast and intensity. The **c\_counter** variable is used for altering the brightness of the light being applied to the game object. The next three variables represent the change in colour for the RGB colour values of the object. This will be explained further in the **main()** method description.

The last two uniform float variables are used for setting the amount of refraction applied to the object and the gamma correction. These variables can both be changed through user input within the **MainGame.cpp** script along with the colour and the brightness.

This is then followed by our reference to the 4D vector of **fragcolor** which is the final colour that will be created for each fragment through our shader. This has to be included in some form in order for the shader to function as intended.

The remaining lines before our **main()** method reference the data we have passed through from the vertex shader, in this case this is **v\_norm** and **v\_pos**.

The code used to generate the effects needed for this graphical technique is contained within the **main()** method. The first thing we do in this method is create a float named **ratio** this float is equal to 1 divided by our refraction value. We then create a 3D vector which we simply name **i** this is calculated by normalizing our **v\_pos** that we passed in from the vertex shader and our uniform variable of the camera position. To normalize we calculate the unit vector in the same direction as the original vector this in turn gives us a vector with the same direction as its parameter. This is done to understand the view/camera direction. We then calculate the refraction which has been named **r**, this gives us the refraction direction of an incident vector, in this case **i** is the incident vector. We make sure to normalize the **v\_norm** vector, our normal vector, as is required for both vectors within refraction calculations. The **i** vector does not need normalized as this has already occurred on the line before. In order to complete a refraction calculation a float must be specified, in this case we use the float **ratio** we created earlier, this allows us to alter the amount of refraction applied to the object, by default in this application the refraction is set to 1.0 which simply makes the object appear transparent.

Then the calculations used for generated our lighting are written. Firstly, a 4D vector named **clipPos** is created, this is the 3D vector of **v\_pos** but with a 1.0 as the 4th dimension. We then have a 3D vector named **n** which is used to represent the value we generate by converting normal to view space. This is done by normalizing the sum of a 3x3 floating point matrix generated from **u\_vm** multiplied by our vertex normal value generated from the vertex shader. The position in clip space is represented by **p** which is equal to the 3D vector generated by multiplying **u\_pm** by the clip-space position we named **clipPos**. This value **p** is then normalized and given the name **v**. A float used for the rim contribution to the Rim-Toon lighting effect is then created, this is named **vdn** and it is calculated by first taking 0.6 and subtracting the max calculated from the dot product of our **v** and **n** values and 0.0. **max()** establishes the higher of two values, this means that if our dot product is less than 0.0 it will return 0.0 as a failsafe. This will result in a rim value of 0.6.

Then a float named **intensity** is declared to represent the lighting intensity along with a 4D vector which we name **color**, this represents the colour we use for the lighting. The variable representing the intensity of our colour is then set equal to 0.0 + **c\_counter**, it is likely easier to make this simply equal to **c\_counter** alone but the 0.0 value had been altered many times throughout development so it has been left in the case that further development were to occur. This is then used in our calculation for our lighting colour. To calculate this we create a 4D vector, the first 3 values of which are our intensity value subtracted by the red, green and blue change uniform variables we assign in **MainGame.cpp**. The fourth dimension, representing our alpha value, is set to 1.0. Subsequently we create a float named **gamma** which is set equal to our **gammaChange** uniform variable, this will be used for the gamma correction operation at the end of the fragment shader.

Our final fragment colour is finally generated at the bottom of the above image. To do this we first set the alpha value to 0.9, however, this doesn’t have any effect in game but must be included to allow the operation to run. We then perform an initial calculation for our rgb values, to do this we create a 3D vector using firstly the texture method, this method retrieves texels from a texture, on our **skybox** and **r** uniform variables. We then add this to the smoothstep calculation of 0.4, 0.8 and the **vdn** float we created previously. **smoothstep()** performs Hermite interpolation between two values, 0.4 and 0.8 represent the lower and upper value of our Hermite function with **vdn** representing the source value used for this interpolation. The values representing x, y and z from our **color** 4D vector are also added into this method for attaining our 3D vector. We then take this value and perform a pow calculation on it, this in turn returns the value of the first parameter raised to the power of the second. The first parameter in this instance is the **fragcolor** rgb values and the second parameter is generated from a 3D vector of 1.0 divided by the **gamma** float. This calculation enables the gamma correction which we apply to our object in the scene.

# Main Game Script

## setCustomValues() Method

A screenshot of a cell phone

Description automatically generated

Figure 3 – setCustomValues() method

Above in Figure 3 we can see the method used to generate our custom values. This method passes in the uniform variables we use for generation of our additional graphical technique. Fully comments are added for this method. For all the float variables we pass in, these values are generated throughout the MainGame.cpp script. The model matrix is attained through a method found within the Transform script and as for the **u\_vm** and **u\_pm** matrices, these are attained through the Camera script, along with the camera position.

## processInput() Method

A screen shot of a social media post

Description automatically generated

Figure 4 - Snippet of processInput method which is used for controlling our lighting inputs

The above image shows a brief snippet of the method used for changing the values used to generate our brightness, contrast and refraction amounts. These are essentially just keyboard inputs which either add or subtract a certain float from the value representing these amounts themselves. For example, were a user to press the “c” key, they would increase the value of colorCounter by 0.5 which would have an effect on the fragment shader and would increase the brightness of the model the shader is attached to.

A screenshot of a cell phone

Description automatically generated

Figure 5 - Inputs for changing colours

This image shows how the **redChange**, **greenChange** and **blueChange** values are affected depending on the number pressed. The associated colour has been commented above the values.

## drawGame() Method

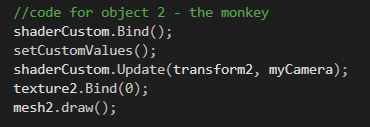


Figure 6 - Method for attaching shader and drawing model

Figure 6 shows specifically the order in which methods are called to properly load the model and the shader attached to it. Firstly we bind the custom shader before running the method to set the custom values. The next method then passes in the transform to the shader. Following this we bind our texture to the mesh, this however isn’t used as we generate a new texture. Lastly, we draw the model. The method used for setting the mesh’s transform is contained elsewhere to be more computationally efficient. The reason behind this is that we call the **drawGame()** method within a loop each frame, having to set the transform also in this loop when they are only generated once at the start and not changed is not efficient so we create a separate method to do this, which is only called one time on start up.

# 3 References

Sphere Model: <https://www.turbosquid.com/3d-models/sphere-obj-free/1114394>

Rat Model: <https://free3d.com/3d-model/low-poly-rat-3205.html>

Monkey Model: provided in lab sessions

Skybox Texture: provided in lab sessions