**Graphics Programming Coursework**

Andrew MacFarlane

Mat#: S1511223

Computer Games (Software Development)

*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*.

*Andrew MacFarlane*

Contents

[Section 1 General Code Layout 3](#_Toc514245691)

[Section 1.1 Activating an effect 3](#_Toc514245692)

[Section 1.2 Controls 3](#_Toc514245693)

[Section 1.3 *DrawLoop()* method 3](#_Toc514245694)

[Section 1.3.1 *SetMeshPositions()* 3](#_Toc514245695)

[Section 1.3.2 *ApplyEffect()* method 4](#_Toc514245696)

[Section 1.4 *HandleInput()* Method 4](#_Toc514245697)

[Section 1.5 Applying the effects to the models 5](#_Toc514245698)

[Section 2 The shader class 5](#_Toc514245699)

[Section 2.1 Initialising Shaders 5](#_Toc514245700)

[Section 2.1.1 Initialising vertex and fragment shaders 5](#_Toc514245701)

[Section 2.1.2 Initialising an effect which uses the geometry shader 5](#_Toc514245702)

[Section 2.2 Binding the shaders 6](#_Toc514245703)

[Section 3 Toon Shading 6](#_Toc514245704)

[Section 3.1 Fragment Shader 6](#_Toc514245705)

[Section 3.2 Vertex Shader 7](#_Toc514245706)

[Section 4 Rim Toon 8](#_Toc514245707)

[Section 4.1 Fragment Shader 8](#_Toc514245708)

[Section 5 Fog 9](#_Toc514245709)

[Section 6 Explosion and Hair Effects 9](#_Toc514245710)

[Section 7 Blur 9](#_Toc514245711)

[Section 8 Phong/Blinn-Phong 9](#_Toc514245712)

[Section 9 Appendix 9](#_Toc514245713)

# Section 1 General Code Layout

## Section 1.1 Activating an effect

In this project in order to apply an effect to the models various function calls in the games draw loop are made to set the positions of the models and then subsequently bind and apply the effects. For most effects the program will render three models and apply the chose effect to all three models. The chosen model can be selected with the numeric controls.

## Section 1.2 Controls

The effect on the models can be selected by pressing the corresponding key, however it should be noted these controls do work with the number pad. Please see the table below for a list of effects which can be activated and their corresponding controls (Figure 1).

|  |  |
| --- | --- |
| Effect | Key |
| Fog | 1 |
| Rim Toon | 2 |
| Explosion | 3 |
| Explosion + Hair | 4 |
| Hair | 5 |
| Toon | 6 |
| Phong/Blinn-Phong | 7 |
| Blur | 8 |

## Section 1.3 *DrawLoop()* method

This method the setup of models and effects is essentially handled in this method. This method sets the background colour and a calls relevant methods needed for the setup of models. Additionally it updates the counter that is used to move the models in a way that will allow the displaying of effects. This method also swaps to the next window as we draw to one of the two frames which is not being displayed.

### Section 1.3.1 *SetMeshPositions()*

This method is used to set the positions, scale and rotation of the models and its general purpose is to keep the code tidy as these commands can easily clog up the draw method. This method can be seen in Figure 2

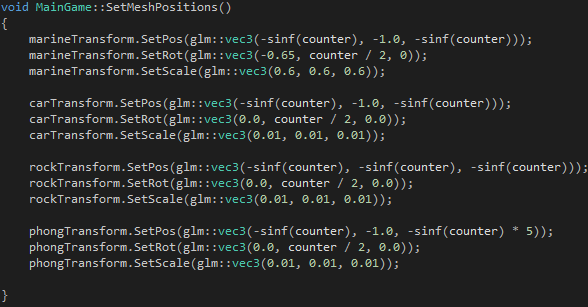


Figure 2

### Section 1.3.2 *ApplyEffect()* method

This method is used to apply the effects to the models once their positions have been set by *SetMeshPositions()* if takes in a float which it used to determine which effect should be applied. This float is gained from a case statement which is explained in section 1.4. An example of this method applying the fog shader to the models can be seen in Figure 3.

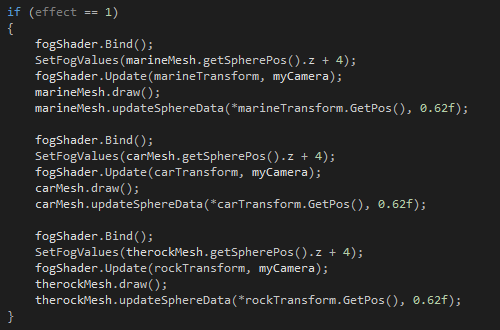


Figure 3

## Section 1.4 *HandleInput()* Method

This method uses a simple case statement to assign a value to the global *effect* float. It uses an SDL key event to determine which key has been pressed and then sets *effect*  to the appropriate values which when passed into the *ApplyEffect()* method will allow the appropriate effect to be applied. This can be seen in Figure 4.

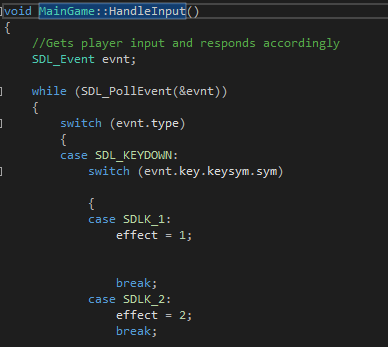


Figure 4

## Section 1.5 Applying the effects to the models

# Section 2 The shader class

## Section 2.1 Initialising Shaders

### Section 2.1.1 Initialising vertex and fragment shaders

In order to properly apply shaders to models we first need to initialise the shaders to memory. This task is completed by the *initialise()* method it takes in string values for the file paths of both the vertex and fragment shaders. It is important to note that this method is called in MainGame.cpp in order to pass the shader files in by giving this method the appropriate file paths. The method creates a shader for both the vertex and fragment shaders. The method also binds attribute locations in which to store the vertex normal and positions, which it then links to the shader programme that will then be run on the GPU.

### Section 2.1.2 Initialising an effect which uses the geometry shader

In order to make use of effect which require the geometry shader to be accessed and manipulated it is necessary to create an overload method for the method described above which performs, essentially the same function except that it will load an addition file to be used to create a geometry shader program. This can be seen in Figure 5.

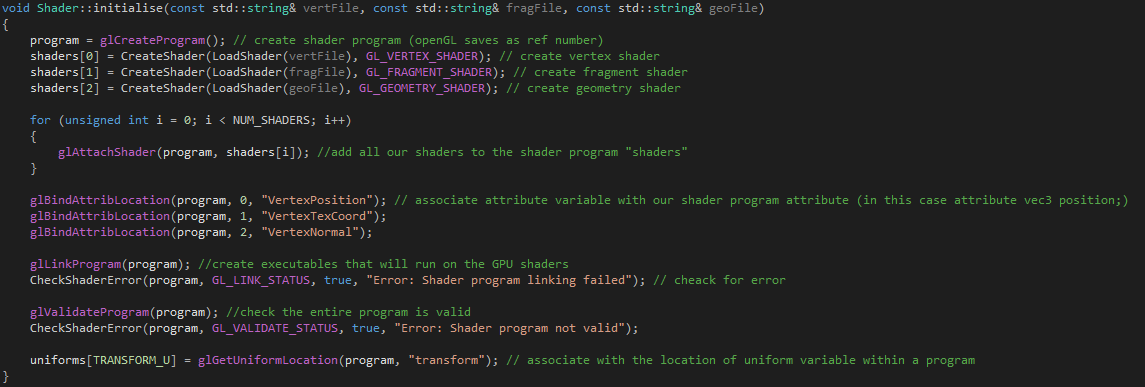


Figure 5

## Section 2.2 Binding the shaders

Binding of the shaders is handles by the *Bind()* method. This is a simple method that installs and executes the shader programme to the desired object when called in the draw loop in MainGame.cpp. This can be seen in Figure 6.

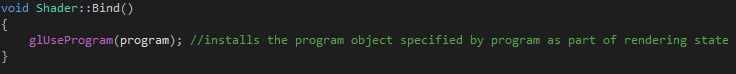


Figure 6

# Section 3 Toon Shading

Toon shading is a simple graphical effect which uses a one- dimensional texture map as a lookup table to fill a models geometry with a solid colour from the texture map. An example of this effect can be seen in Figure 7.



Figure 7 - Toon Shading

## Section 3.1 Fragment Shader

The effect is achieved by manipulating the frag colour which is output from the fragment shader. It works by calculating the diffuse lighting intensity ( the dot product of the lights direction and the surface normal). This value is then mapped to the one dimensional texture map to determine which areas should be coloured and to what extent. This can be seen in Figure 8.

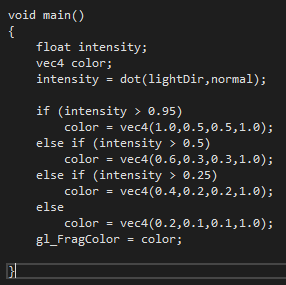


Figure 8

## Section 3.2 Vertex Shader

For a vertex shader this effect requires very little data and therefore only need to pass on the vertex normal and positions. An example of this can be seen in Figure 9

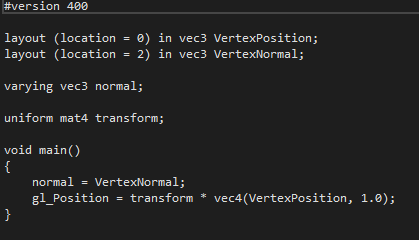


Figure 9

Henceforth any effect of which details of the vertex shader are not given it can be assumed that it using this vertex shader.

# Section 4 Rim Toon

Rim Toon is an effect which combines both toon shading and rim lighting. An explanation of toon shading can be found in Section 3. Rim lighting on the other hand is and effect that simulates the bleeding of light around and object. The lights intensity is calculated by finding the dot product between the surface normal and the view direction. An example of RimToon shading can be seen in figure 10.

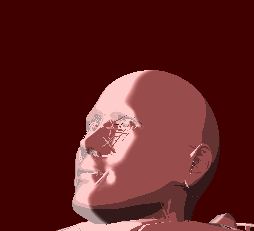


Figure 10 – Rim Toon Shading

## Section 4.1 Fragment Shader

As a result of the fact that this effect is achieved by combining the rim and toon effects the fragment shader is quite similar to that which is used for toon shading. The key difference is that a float vdn is calculated to represent the rim contribution. To create the finally outputted frag colour the rim contribution is smoothed and interpolated and then combined with the results of the toon shading colour calculations. This can be seen in figure 11.

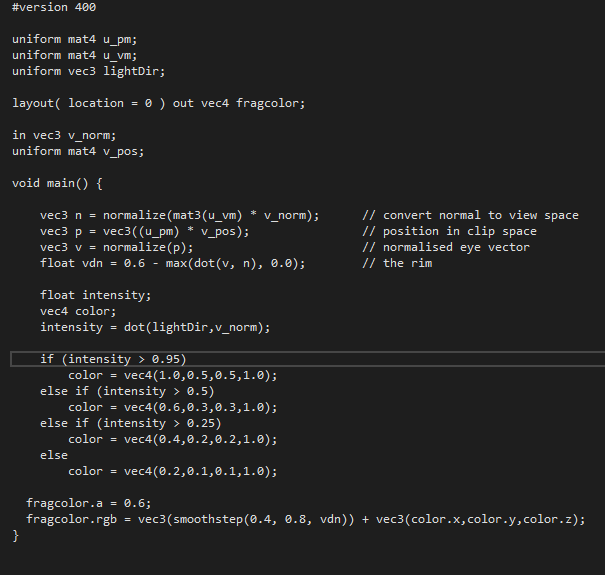


Figure 11

# Section 5 Fog

Fog is a simple effect which can be generated by mixing the colour of each fragment with a constant fog colour. The effect which fog can have on the applied model is related to the distance between the model and the camera. In this instance the fog effect becomes more intense as the model moves away from the camera This can be seen in figures 12.1 and 12.2.

Figure 12.1 – Fog Effect, close to camera



Figure 12.2 – Fog Effect far from camera

## Section 5.1 Fragment Shader

In this instance to create the fog effect it has been combined with the rim toon shader discussed in Section 4. As such the fragments for these effects are fairly similar. In order to calculate the fog factor the value is clamped between o and 1. The fog colour is then linearly interpolated (using the “mix ” command) between the fog factor and the result of the rim toon calculations and applied to the fragment. This can be seen in figure 13.

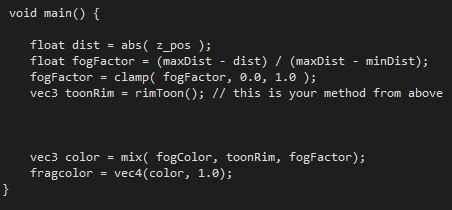


Figure 13

# 

# Section 6 Explosion and Hair Effects

## Section 6.1 Hair

This effect was achieved by implementing a shader that allows for the normal vectors of a model to be visualized. By doing this it creates the impression that the model is covered in hair. An example of this can be seen in Figure 14

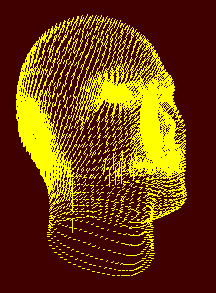


Figure 14 – Hair Shader

Hair was created mostly by code in the geometry shader as it is needed in order to create the vertices which represent the “hairs”.

### Section 6.1.1 Vertex Shader

This shader uses a reasonably simple vertex shader similar to that discussed previously. However, the models transform is additionally required to be passed out this is because the vertex normals from the model are required to create this effect. Additionally the normal are translated to clip space first as well this can be seen in figure 15.

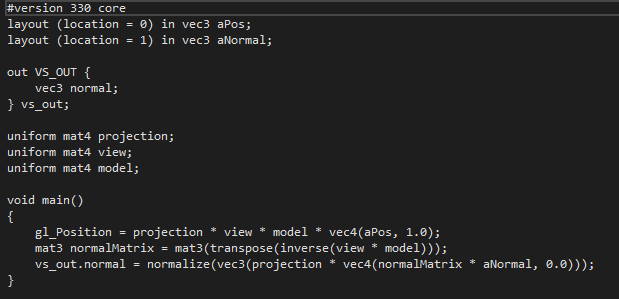


Figure 15

### Section 6.1.2 Fragment Shader

In the fragment shader because we are only drawing simple lines the frag colour is simply set to a solid yellow colour. See Figure 16.

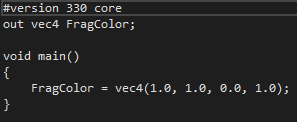


Figure 16

### Section 6.1.3 Geometry Shader

The task of the geometry shader in this instance is to draw a line in the form of a normal vector from each position on the models geometry. Additionally, these values as passed in via an interface block. Also the magnitude value is used to limit the length of the lines that are generated. This can be seen in figure 17

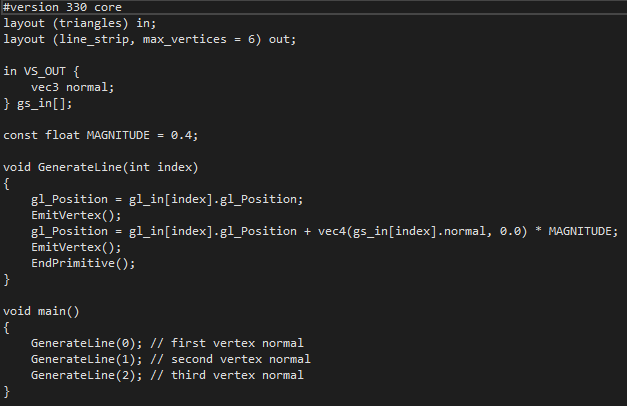


Figure 17

## Section 6.2 Explosion

This effect was implemented by moving vertices along the normal vector. This has the effect of making it look like the model is exploding which can be seen in Figure 18.1 and 18.2

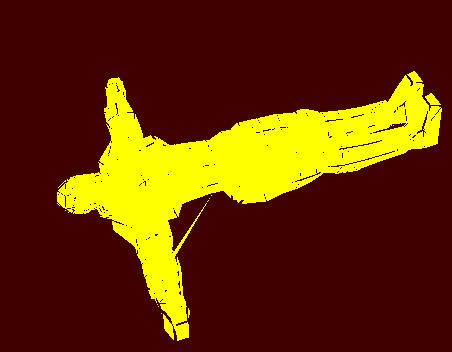


Figure 18.1 – Explosion Shader, unexploded.

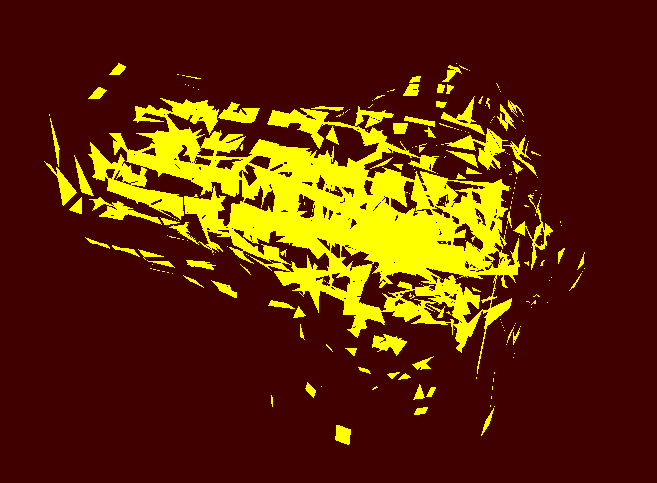


Figure 18.2 – Explosion Shader, exploded.

### Section 6.2.1 Vertex Shader

Due to the similarity of the effects the same vertex shader is used for both the hair and explosion shaders.

### Section 6.2.3 Fragment Shader

As above and because the fragments are just being moved they are also a simple solid colour as such the fragment shader is the same as that used for the Hair shader.

### Section 6.2.3 Geometry Shader

In order to create the explosion effect the shader must first acquire the normals. This is performed by the *GetNormals()* Method. Additionally the explode method will return a vector which is used to translate the vertex along the normal vector. See figure 19

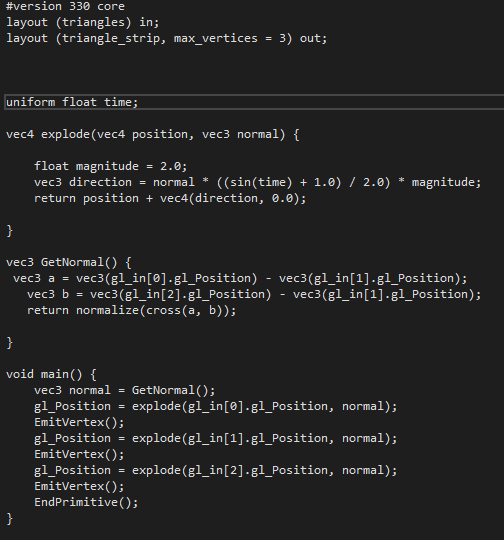


Figure 19

## Section 6.3 Combination of effects

In an attempt to combine these two effects the main changes were the combining of the two effects geometry shaders whilst the vertex and fragment shaders stayed the same. This proved somewhat unsuccessful as the models do no actually explode and it is unclear why this is the case. It could be that the model data is not being passed in correctly but the source of the problem could not be located in an appropriate time frame. The theory behind what is there was that by applying the explosion translation to the lines generated by the hairy shader would create the effect. The shader can be seen in figure 20.

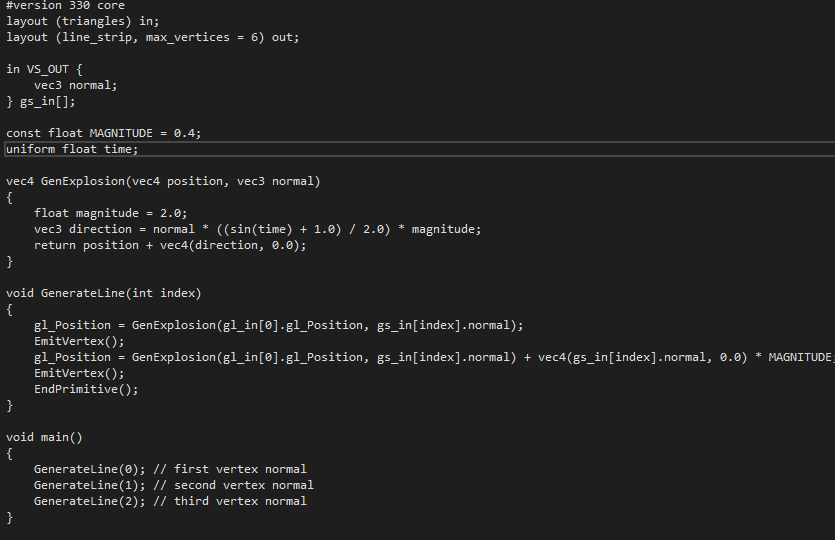


Figure 20

# Bibliography

cvbtruong. (n.d.). *Dragon Head*. Retrieved from Turbo Squid: https://www.turbosquid.com/FullPreview/Index.cfm/ID/668714

gentlemenk. (n.d.). *Typhoon*. Retrieved from Turbo Squid: https://www.turbosquid.com/FullPreview/Index.cfm/ID/505619

Ilove3D2. (n.d.). *0a GI John - The Stone*. Retrieved from Turbo Squid: https://www.turbosquid.com/FullPreview/Index.cfm/ID/706522

nahchimento. (n.d.). *Lamborghini Aventador LP 750-4 Superveloce (Low Poly)*. Retrieved from Turbo Squid: https://www.turbosquid.com/FullPreview/Index.cfm/ID/1117798

De Vries, J. (2014). [Blog] LearnOpenGL. Available at: https://learnopengl.com/ [Accessed 2 May 2018].