**Matthew Cadden**

**S1508111**

**Game Programming (Software Development)**

**Graphics Programming**

*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*.

Matthew Cadden

Table of Contents

[1. Main Game Class 2](#_Toc514250890)

[1.1 Main Game Header 2](#_Toc514250891)

[1.2 Main Game Source 2](#_Toc514250892)

[2. Shader Class 4](#_Toc514250893)

[2.1 Shader Header 4](#_Toc514250894)

[2.2 Shader Source 5](#_Toc514250895)

[3. Blur Shader 6](#_Toc514250896)

[3.1 Blur Shader Implementation 6](#_Toc514250897)

[3.2 Blur Vertex Shader 7](#_Toc514250898)

[3.3 Blur Fragment Shader 7](#_Toc514250899)

[4. Toon Shader 8](#_Toc514250900)

[4.1 Toon Shading Implementation 8](#_Toc514250901)

[4.2 Toon Vertex Shader 8](#_Toc514250902)

[4.3 Toon Fragment Shader 8](#_Toc514250903)

[5. Rim Shader 9](#_Toc514250904)

[5.1 Rim Shading Implementation 9](#_Toc514250905)

[5.2 Rim Vertex Shader 9](#_Toc514250906)

[5.3 Rim Fragment Shader 9](#_Toc514250907)

[6. Toon/Rim Shader 10](#_Toc514250908)

[6.1 Toon/Rim Shader Implementation 10](#_Toc514250909)

[6.2 Toon/Rim Vertex Shader 10](#_Toc514250910)

[6.3 Toon/Rim Fragment Shader 10](#_Toc514250911)

[7. Toon/Rim Fog Shader 11](#_Toc514250912)

[7.1 Toon/Rim Fog Shader Implementation 11](#_Toc514250913)

[7.2 Toon/Rim Fog Vertex Shader 11](#_Toc514250914)

[7.3 Toon/Rim Fog Fragment Shader 11](#_Toc514250915)

[8. Blinn-Phong Lighting 12](#_Toc514250916)

[8.1 Blinn-Phong Lighting Implementation 13](#_Toc514250917)

[8.2 Blinn-Phong Lighting Vertex Shader 13](#_Toc514250918)

[8.3 Blinn-Phong Lighting Fragment Shader 13](#_Toc514250919)

[9. Normals Shader (Hairy) 14](#_Toc514250920)

[9.1 Normals Shader Implementation 14](#_Toc514250921)

[9.2 Normals Vertex Shader 14](#_Toc514250922)

[9.3 Normals Geometry Shader 15](#_Toc514250923)

[9.4 Normals Fragment Shader 15](#_Toc514250924)

[10. Explosion Shader 15](#_Toc514250925)

[10.1 Explosion Shader Implementation 15](#_Toc514250926)

[10.2 Explosion Vertex Shader 16](#_Toc514250927)

[10.3 Explosion Geometry Shader 16](#_Toc514250928)

[10.4 Explosion Fragment Shader 16](#_Toc514250929)

[Appendix 17](#_Toc514250930)

# Main Game Class

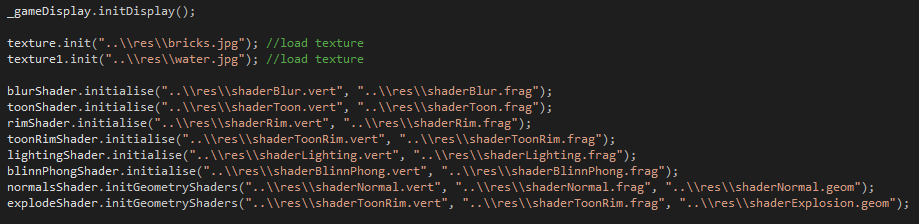
The Main Game Class is where most of the code is that is to display the different models and shaders.

## Main Game Header

This section is where all of our variables and methods that we will be using in the source file for Main Game. This includes setting up the shaders and models so that they can be used them in the game.

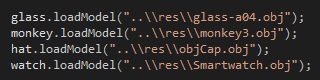
## Main Game Source

The source file contains all the code that is used to run the game.



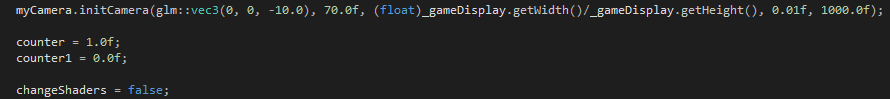
*Fig 1*

This be started off by initialising the game display and then initialising the shaders that will be used in the game as can be seen from fig 1 above.



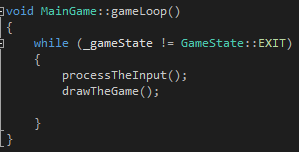
*Fig 2*

After this the models will then be loaded into the correct positions so that they can be used later in the game.



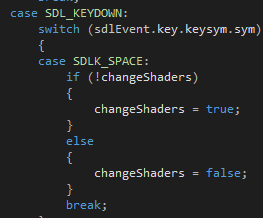
*Fig 3*

The camera is also initialised here with a starting position and a field of view. The camera also sets up an aspect ratio as well as a near and far clip plane. The last thing that is set up is the two counters which will be used in the game as well as a Boolean “changeShaders” which is used so that when a user hits the spacebar different shaders will be applied to the models.



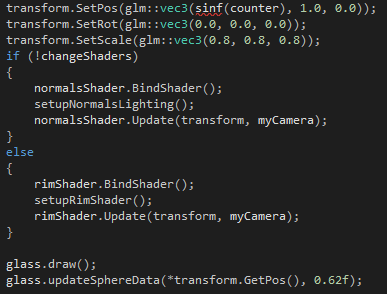
*Fig 4*

After this is the main game loop which will run as long as the game is not exited. This will just constantly check for inputs from the user and draw the game onto the screen.



*Fig 5*

To change shaders the process input method will just check to see when the spacebar gets pressed and when it does will either make a Boolean value true or false.



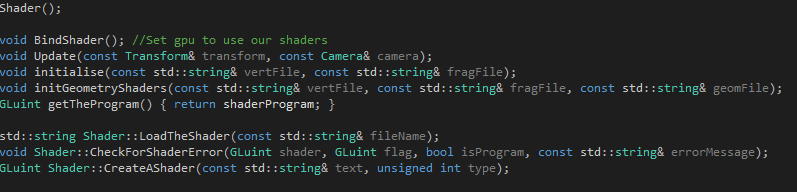
*Fig 6*

This is how a model is set up with shaders in the draw the game method. First it will set up the position, rotation and scale of the model. After this depending on the Boolean variable seen in Fig 5 it will attach one of two shaders onto the model. This is done by first binding the shader then setting up the values needed by the vertex and fragment shaders. Finally, it will update the shader based on the transform and camera and draw the model with the shader applied onto the screen. This is the same for all four models but with different choices of shaders on each model. How the lighting is setup will be discussed more in-depth in the individual shader sections.

# Shader Class

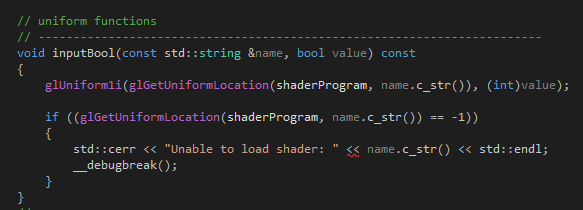
The shader class contains all the code that is used by the shaders in the main game.

## Shader Header



*Fig 7*

This is where the methods are set up so that we can use them when we create a shader.

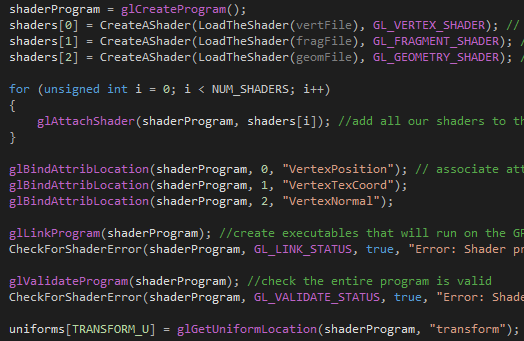


*Fig 8*

In the header file there are also methods so that uniforms can be input into the shaders as can be seen from the example of inputting a Boolean uniform from Fig 8 above. These will also return error messages if they cannot load the shader program.

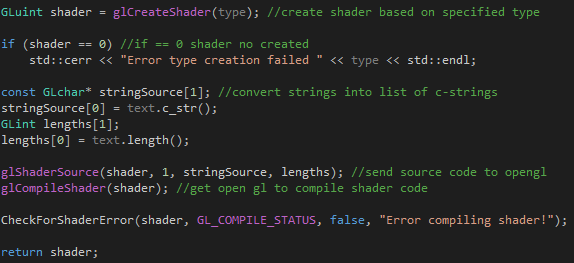
## Shader Source

To initialise a shader there is two different methods that are in use. One is used when there is just a fragment and vertex shader to load and the other method will be used when we also have to load in a geometry shader.



*Fig 9*

First a shader program is created and then the shaders are loaded into a GLuint which holds an array of shaders. After this the shaders are attached to the program. After this the program attributes are given a location to associate the attribute variables. An executable is created after this so that the program will run on GPU shaders. Finally, it is checked that the shader is valid and the transform uniform is associated with the location of a uniform within the program.



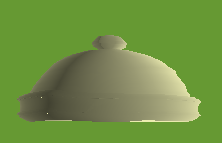
*Fig 10*

When creating an individual shader the program will first create a shader based on the type that was passed into it. After this it will convert the shader code so that the source code can be sent across to OpenGL. OpenGL will then compile the shader and if there were no errors this shader is then returned.

To load a shader it is opened up and then read in line by line as a string.

# Blur Shader

The blur shader is used to make a model appear blurrier.

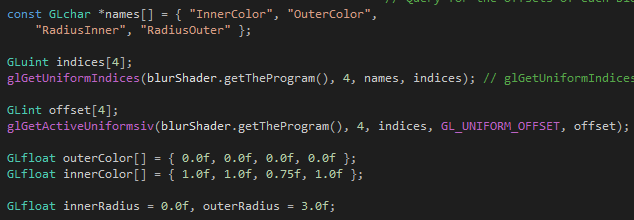


*Fig 11*

This is how it appears on the model in game.

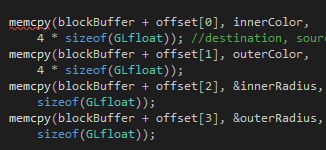
## Blur Shader Implementation

When using the blur shader the shaders need uniforms for an; inner colour, outer colour, inner radius and an outer radius. To get this we make use of the blob effect method.



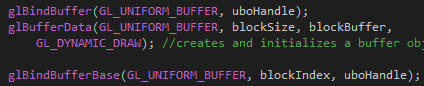
*Fig 12*

This is where the variables are created and assigned values so that they can be passed into the shaders. Also at this stage information about the uniforms in the shader program will be returned.



*Fig 13*

After this the data is stored into memory given a destination, source and number of bytes



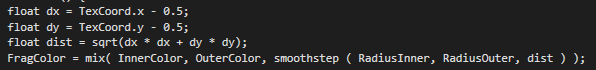
*Fig 14*

Finally, a buffer objects data store is created and initialised it is then bound to an indexed buffer target.

## Blur Vertex Shader

The vertex shader for blur will make use of the transform uniform to update the position of the vertices for the shader. This vertex shader will also output the Texture Coordinates.

## Blur Fragment Shader

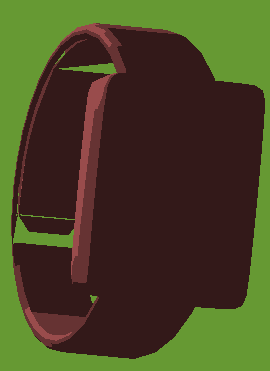


*Fig 15*

As can be seen in fig 15 a distance is calculated using the texture coordinates. Outer colour will define the colour outside the circle while inner colour will define the colour inside of the circle. The inner radius is the distance from the centre of the circle to the inner edge of the fuzzy boundary and this is a solid colour. The outer radius is the outer edge of the fuzzy boundary and at the very edge of this is where the colour equals outer colour. The smoothstep function is used as it provides a value that will smoothly go from 0 to 1. The mix function is used in conjunction with this to linearly interpolate between the inner and outer colour.

# Toon Shader

Toon shading makes use of a one-dimensional texture map as a lookup table so that it can fill geometry with a solid colour. It will diffuse the lighting intensity at the texture coordinate into a one-dimensional texture that contains a gradually brightening colour table. Toon shading will give the model a cartoonish, non-photo realistic look.



*Fig 16*

Fig 16 shows how this model appears in the game.

## 4.1 Toon Shading Implementation

To set up toon shading the only value that needs to be passed into the fragment shader is the light direction.



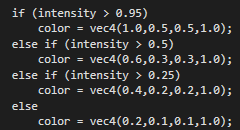
*Fig 17*

## 4.2 Toon Vertex Shader

The vertex shader for toon shading will calculate the normal and the position of the vertices.

## 4.3 Toon Fragment Shader

The fragment shader will use the light direction provided and the normal calculated in the vertex shader. To calculate the intensity of the light it will get the dot product between the light direction and the normal.



*Fig 18*

The intensity value that is returned from this will then be used to calculate which colour will be used on that normal.

# Rim Shader

The rim shader is used to highlight certain areas of a model, normally the rim. This rim shader will highlight areas that are not on the 360o z of the model.



*Fig 19*

This wine glass is how rim shading appears in game.

## Rim Shading Implementation

When setting up rim shading the fragment shader will need the view matrix and the projection matrix which we do in the Main Game.



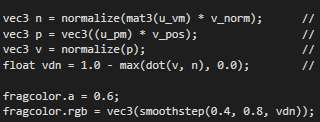
*Fig 20*

To do this methods are used that were set up in the camera class so that the view and projection matrices can be returned.

## Rim Vertex Shader

In this vertex shader the normal and positions of the vertices must be outputted onto the fragment shader as well as calculating the positions of the vertices.

## Rim Fragment Shader



*Fig 21*

In this fragment shader the normal is converted so that it will be in view space by using the view matrix that was passed in as a uniform. After this the position is changed to be in clip space using the projection matrix. The eye vector is then normalised and the rim contribution is calculated by using the dot product of the normalised eye vector and the normal in view space. The colour of the vertices are then calculated from this.

# Toon/Rim Shader

The toon rim shader is a combination of the toon and rim shader that were previously discussed above. So this shader will give the model an unrealistic cartoonish look as well as a highlight on areas that are not on the 360o z of the model.



*Fig 22*

Here we can see it in game and compare it to the earlier shading seen on the same model just using the toon shader in fig 16.

## Toon/Rim Shader Implementation

As this is just a mixture of toon and rim shading implementation is largely the same as it was for they two shaders.



*Fig 23*

The uniforms we use to pass in are the same as before but are now joined together so the light direction, view matrix and projection matrix are being passed into the same shader.

## Toon/Rim Vertex Shader

The vertex shader used for this is the exact same vertex shader used for the Rim shader which was discussed in section 5.2.

## Toon/Rim Fragment Shader

Both of these shaders have now been joined together and are put into the same fragment shader however there is no change in code apart from at the very end when it is calculating the colour.



*Fig 24*

Now the colour is also multiplied by color.xyz which is the colour calculated by the toon shader part of this fragment shader. This is how both of these shaders are joined together.

# Toon/Rim Fog Shader

This will use the Toon/Rim Shader from above but will also add fog onto the model. The extent and colour of the fog depend on how far the object is from the eye. This means that the fog will get heavier on the model the further it is from the eye position and a user will be able to make out less detail.

*Fig 25 Fig 26*

As can be seen from figures 25 and 26 when the head of the monkey gets closer to the eye position more detail can be made out from it.

## Toon/Rim Fog Shader Implementation

When setting up this shader we must pass in the variable used previously in Toon/Rim which are the light direction, projection matrix and view matrix. However, the fog colour, minimum distance and max distance of the fog must also be passed into the fragment shader. This is done so that the position the fog starts and where it is at its greatest can be defined along with the colour of the fog which will gradually be applied more to the model as it gets closer to the max distance. Another variable we must pass in is the model z position for its sphere as seen in figure 27 below.



*Fig 27*

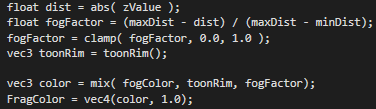
This must be passed in so that we can constantly update the z position of the model and are not left with just a static colour on the model.

## Toon/Rim Fog Vertex Shader

The vertex shader used for this is the exact same vertex shader used for the Rim shader and the Toon/Rim shader which have been discussed in section 5.2 and section 6.2.

## Toon/Rim Fog Fragment Shader

The fragment shader starts out similar to how it did in toon/rim shading however instead of being in the main method of the shader toon/rim shading is calculated in a separate method which will return the colour calculated from it at the end.

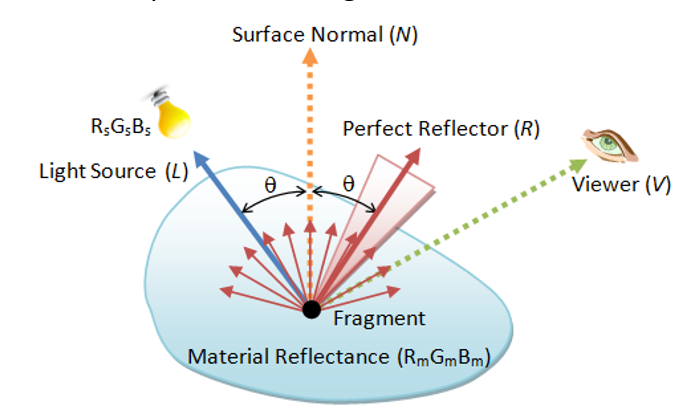


*Fig 28*

Figure 28 shows the new main method in the fragment shader. The float dist is used to store the distance from the surface point to the eye position. We use the z value passed in from the models sphere position for this. After this the fog factor is calculated but must be clamped between 0 and 1 as dist can never be between minDist and maxDist. The toonRim method is then called which will return a colour for the fragment using only the toon/rim shader. Finally, the fog colour and the result of the toon/rim shader is mixed together based on the fog factor and this result is used as the fragment colour.

# Blinn-Phong Lighting

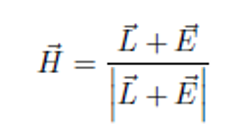
Blinn-Phong Lighting is an optimisation of the phong lighting model. This works on the simple principle that objects have three material properties which are ambient, diffuse and specular reflectivity. These properties are assigned colour values with brighter colours representing a higher amount of reflectivity. Light sources have these same properties and are assigned colour values that represent the brightness of the light. The final colour value is the sum of all the lighting and material interactions of these three properties.



*Fig 29*

Figure 29 shows how the phong lighting model works.

The Blinn-Phong lighting model is considered an optimisation on this as instead of calculating the dot product of the perfect reflector R and surface normal N at each shaded point a halfway vector H is introduced. H is halfway between the light vector and the eye vector and is calculated as shown below in figure 30.



*Fig 30*

*Fig 31 Fig 32*

Figures 31 and 32 show the blinn-phong effect and how the light updates depending on the models position.

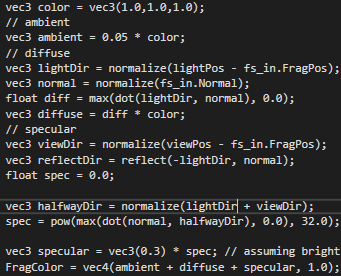
## Blinn-Phong Lighting Implementation

To implement blinn-phong lighting a light position for the source light and a view position were passed into the shader.

## Blinn-Phong Lighting Vertex Shader

The vertex shader for blinn-phong will output a fragment position and normal onto the fragment shader. It will also update the position of vertices.

## Blinn-Phong Lighting Fragment Shader



*Fig 33*

First a colour is set. That colour is then multiplied by 0.05 to get the ambient light. To get the diffuse the light direction is found by normalising the light position minus the fragment position. After this the normal has to be normalised. To get float diff the dot product is taken of the light direction and normal. This will either take this value or 0 so that there are no negative numbers. Diffuse is then calculated by diff multiplied by colour. To calculate the specular light first, the view direction is calculated by normalising the view position minus the fragment position. The reflect direction is calculated using the inverse of the light direction and the normal. A float for spec is then set. What sets this apart from Phong lighting is now the halfway vector is calculated and using this we get a value for spec. This then uses spec and a bright white colour to get the specular colour. To get the fragment colour we add these up in a vec4.

# Normals Shader (Hairy)

This shader will visualise the normal vectors on an object. This will show all the normal vectors emitting off of a model as yellow lines. To do this unlike in previous shaders a geometry shader is also going to be used.



*Fig 34*

This is how it looks in game for the wine glass object.

## Normals Shader Implementation



*Fig 35*

To implement this all that was needed to be passed into the vertex shader was the projection, view and model matrices which make up the mvp matrix.

## Normals Vertex Shader

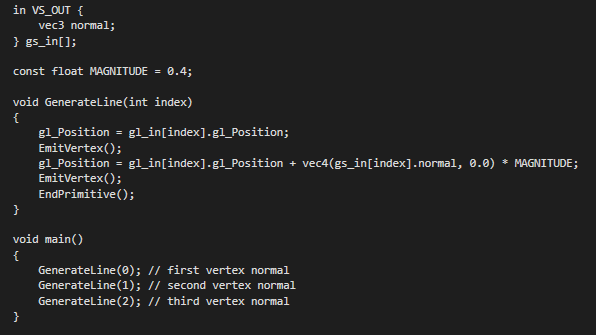
First the normal have to be transformed with a normal matrix before it is moved to clip-space which is done in the vertex shader. The position of the vertices is updated using the transform uniform.



*Fig 36*

The normal matrix is also calculated here and then used so that the normal outputted have been multiplied by the matrix.

## Normals Geometry Shader



*Fig 37*

The geometry shader here will take each vertex with a position and normal vector and draw a normal vector from each position vector. The magnitude is set to limit the size of the vectors that will be displayed on the screen.

## Normals Fragment Shader

This shader will just set the fragment colour of the lines, in this case yellow.

# Explosion Shader

The explosion shader works by moving the triangles on the object along the normal vector over a certain period of time.

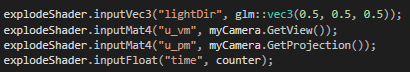
 

*Fig 38 Fig 39*

This shows one of the models with the exploding geometry shader attached to it, the other two shaders are just toon rim shaders as the explosion is handled exclusively in the geometry shader.

## Explosion Shader Implementation

To implement this as it uses Toon/Rim shading values for they shaders also had to be passed in. A time value had to be passed into the geometry shader so the object can explode and repiece itself.

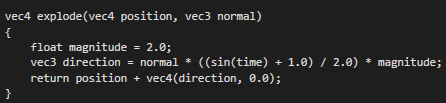


*Fig 40*

## Explosion Vertex Shader

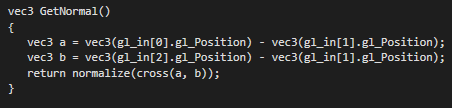
This shader is the same shader used in Toon/Rim shading discussed in section 7.2.

## Explosion Geometry Shader



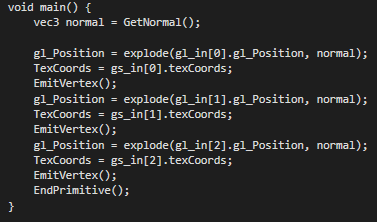
*Fig 41*

From this a direction vector is calculated and this is added to the position vector.



*Fig 42*

Subtracting these three points in the triangle from each other results in a vector parallel to the plane.



*Fig 43*

Now we are moving all the points as well as the texture coordinates along the normal to create the effect of the explosion.

## Explosion Fragment Shader

This shader is the same shader used in Toon/Rim shading discussed in section 7.3.

# Appendix

Avalon, S. (2016, July 2). *Smartwatch*. Retrieved May 14, 2018, from TurboSquid: https://www.turbosquid.com/FullPreview/Index.cfm/ID/1051121

bescec. (2012, January 26). *glass-a04*. Retrieved May 14, 2018, from TurboSquid: https://www.turbosquid.com/FullPreview/Index.cfm/ID/651391

Mons, M. (2017, January 29). *Tweed Cap*. Retrieved May 14, 2018, from TurboSquid: https://www.turbosquid.com/FullPreview/Index.cfm/ID/1118174

Vries, J. D. (2014). *Advanced Lighting*. Retrieved May 11, 2018, from Learn OpenGL: https://learnopengl.com/Advanced-Lighting/Advanced-Lighting

Vries, J. D. (2014). *Geometry Shader*. Retrieved May 15, 2018, from Learn OpenGL: https://learnopengl.com/Advanced-OpenGL/Geometry-Shader