Ben Madine

S1507379

Games 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: Ben Madine*

Contents

[1.0 Class Descriptions 3](#_Toc514232795)

[1.1 Main Game Class 3](#_Toc514232796)

[1.2 Shader Class 3](#_Toc514232797)

[1.2.1 initFragVert 3](#_Toc514232798)

[1.2.2 initFragVertGeom 4](#_Toc514232799)

[1.2.3 Binding 4](#_Toc514232800)

[1.2.4 Update 4](#_Toc514232801)

[1.2.5 ShaderCreator 4](#_Toc514232802)

[1.2.6 loadingShader 5](#_Toc514232803)

[1.2.7 CheckError 5](#_Toc514232804)

[2.0 Shaders 6](#_Toc514232805)

[2.1 Fog Shader 6](#_Toc514232806)

[2.1.1 Vertex Shader 6](#_Toc514232807)

[2.1.2 Fragment Shader 7](#_Toc514232808)

[2.2 Visualizing Normal Shader 8](#_Toc514232809)

[2.2.1 Vertex Shader 8](#_Toc514232810)

[2.2.2 Geometry Shader 9](#_Toc514232811)

[2.2.3 Fragment Shader 10](#_Toc514232812)

[2.3 Explosion Shader 11](#_Toc514232813)

[2.3.1 Vertex Shader 11](#_Toc514232814)

[2.3.2 Geometry Shader 11](#_Toc514232815)

[2.3.3 Fragment Shader 13](#_Toc514232816)

[2.4 Phong Lighting Shader 14](#_Toc514232817)

[2.4.1 Vertex Shader 14](#_Toc514232818)

[2.4.2 Fragment Shader 14](#_Toc514232819)

[2.5 RimToon Shader 17](#_Toc514232820)

[2.5.1 Vertex Shader 18](#_Toc514232821)

[2.5.2 Fragment Shader 18](#_Toc514232822)

[2.6 Ripple Shader 20](#_Toc514232823)

[2.6.1 Vertex Shader 20](#_Toc514232824)

[2.6.2 Fragment Shader 20](#_Toc514232825)

[3.0 Appendix 21](#_Toc514232826)

# Class Descriptions

This section of the report will discuss and highlight the two separate classes that were primarily used in the development of the shaders shown in the project.

## Main Game Class

The Main Game class is a crucial part of the program. The class contains a range of important methods and variables which range from: model transforms, model textures, the meshes for the models, and all the different shaders used in this project.

The main game class has undergone severe changes when being compared to the original lab code. To begin with all the audio and collision code has been removed – this is due to the fact that the code was not necessary in the development of shaders. As well as this, the code has been organized into a certain structure will allows for an easier implementation of new code.

## Shader Class

The shader class is used to create the certain shader types used in a shader, these being: vertex shader, fragment shader and a geometry shader.

The vertex shader handles the processing of vertex data and changes the position of a vertex. It is fed data from the Vertex Attribute Data from the vertex array object using the draw function.

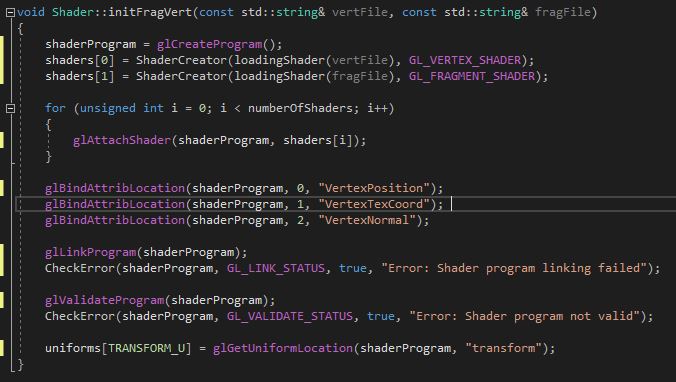
The fragment shader takes a fragment produced by the rasterization process and turns it into a set of colour’s and a single depth value.

Lastly, the geometry shader handles the processing of primitives. The geometry shader takes a single primitive as input and can output zero or more primitives.

As well as this the Shader class contains multiple methods:

### initFragVert

This method contains two parameters, two different strings that allow the vertex and fragment files to be passed in. Firstly, the shader program is created which openGL saves as a reference number. This then follows on to the creation of the vertex and fragment shaders. These shaders are then added to our shader program using a *for* loop which takes in the number of shaders being used. The next part of the method is to associate the selected variable with the correct shader program. The last part of this method is creating executables that will run on the GPU. Additionally, an error check is implemented to check if there are any issues with each individual shader. Furthermore, the whole shader program is then validated along with an error check to validate the integrity of the program. The final line of this method links the location of the uniform variables within the shader program. This method can be seen in *Figure 1.0*:



Figure

### initFragVertGeom

The initFragVertGeom is almost identical to the initFragVert method apart from two aspects. These two aspects are: initFragVertGeom has three string parameters, one for the vertex shader, fragment shader and geometry shader. The second difference is a geometry shader is created as well as the vertex and fragment shader.

### Binding

The binding method contains *glUseProgram* which installs a program object as part of the rendering state. glUseProgram installs the shader program. There are executables created in this process by attaching the shader objects which are made in the initFragVert(Geom) methods.

### Update

The update method has two different parameters: a transform and a camera. The method produces the model view project matrix (MVP). The MVP consists of three separate matrices: the model, view and projection matrix.

### ShaderCreator

The ShaderCreator method has two parameters: a string and an integer. The method creates the shaders and is called in the initialize methods. The first line of code is *glCreateShader*, this indicates the type of shader to be created. It creates an empty shader object that returns a value greater than zero. If the returned value is a zero, then an error message is displayed in the console window. The next stage of the method is to convert the strings into a list of c-strings. C-strings are simply an array of characters. We then need to send the source code to openGL which will allow openGL to compile the shader code. Finally, an error check is in place and a shader is returned.

### loadingShader

The loadingShader method has one parameter a string. This string is the file name which is used to open the shader files.

### CheckError

The CheckError method simply is used to return error messages when a problem is found in the shaders.

# Shaders

## Fog Shader

The fog shader implements a fog effect to certain models. In this projects case, the fog is applied to the *Egg Man* model. To link the fog shader to the model, certain variables are used which assign values to particular fog attributes. These variables can be seen in *Figure 2.0*:

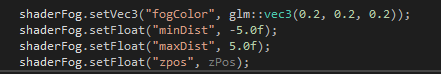
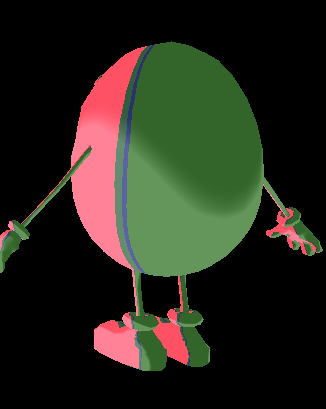


Figure 2.0

The first variable in the sample code above assigns the *fogColor* attribute with a colour. This is done by using a vector 3 value, RGB. The *minDist* variable is the distance from the eye to the fog’s starting point, so when the fog is noticeably visible and effecting the model. Whereas, the *maxDist* variable specifies where the fog is at maximum influence. The *zpos* variable is used as an estimation of the actual fog distance.



Effect 1.0 Effect 1.1

*Effect 1.0* and *Effect 1.1* demonstrate the effect the fog shader has on the model. *Effect 1.0* resembles the model when closest to the viewer whereas *Effect 1.1* resembles the model when furthest away.

### Vertex Shader

The start of the vertex shader assigns the layout qualifiers. This obtains the attribute zero in the shader program and assigns the values to the variable called *VertexPosition*. This process is repeated for the vertex normals. The next stage of the vertex shader is to pass in the value of the transform which is assigned in the shader class. The next stage in the vertex shader is passing out the *v\_norm* (the vertex normal)and *v­­\_pos* (the vertex position)values to the fragment shader.

The main method within the vertex shader assigns the *v\_norm* and *v\_pos* variables. As well as this, the position of the current vertex is set, this is done using *gl\_Position*. The vertex shader can be seen in *Figure 2.1:*

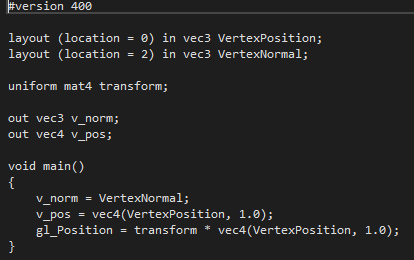


Figure 2.1

### Fragment Shader

The fog fragment shader also contains code for the toonRim shader. However, that code will be discussed further on in this project. The fragment shader consists of many variables but the initial fog variables are minDist, maxDist, *fogColor*,and *zpos* which are all discussed in [*Section 2.1*](#_Fog_Shader). The main method inside the fog fragment shader consists of the calculations which produce the fog effect. These can be seen in *Figure 2.2:*

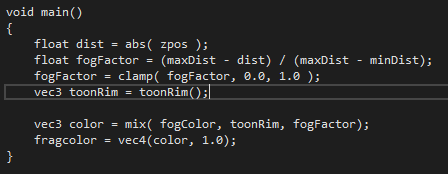
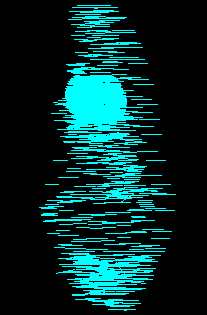
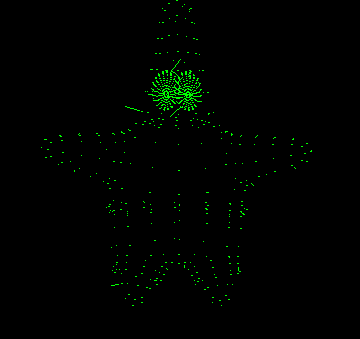


Figure 2.2

Firstly, the variable *dist* defines the distance from the surface point of the model to the eye position. The variable *fogFactor* contributes to the colour of the fog. It is calculated using the former equation in *Figure 2.2*. The *fogFactor* value is then clamped, so the vale is between zero and one, this is because the variable *dist* might be out with the values of *minDist* and *maxDist.* Lastly, the *fogColor*, *toonRim*, and, *fogFactor* variables are mixed to produce the fragment colour.

## Visualizing Normal Shader

The visualizing normal shader allows a models normal vectors to be visible. The theory behind this shader is to supply the vertex normals from the model to the geometry shader rather than generating the normals manually. This can be done in the vertex shader.



Effect 2.0 Effect 2.1

*Effect 2.0* and *Effect 2.1* are examples of the visualizing normal shader during run-time.

### Vertex Shader

The vertex shader is similar to the fog vertex shader apart from one main differences. The difference being the way the vertex is assigned its value. This time a normal matrix is used to transform the normal before transforming the normals into clip space coordinates. This is done by using the code *transpose(inverse(transform))* which swaps the matrices rows for its columns. The normals is then multiplied by the transform and the sum of the new normal matrix and vertex normal. These calculations can be seen in *Figure 2.3*:

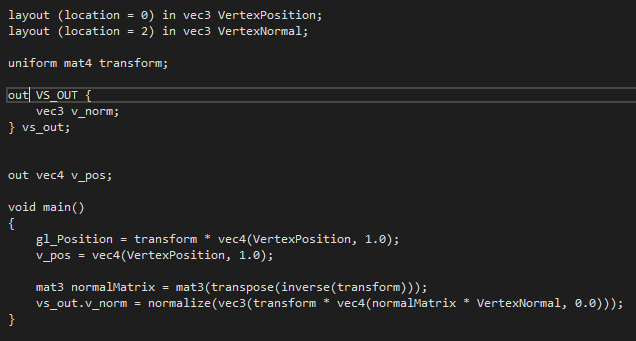


Figure 2.3

### Geometry Shader

The geometry shader starts off by assigning the layout qualifiers by passing in triangles and passing out a max number of vertices and line segments. The following part passes out the vertex normals and then a magnitude is set to reduce the size of the vertex normals.

The first method in the geometry shader, *GenerateLine*, is used to generate the vertex lines. This is done by holding the position of the current vertex, by using *gl\_Position*, and then calling *EmitVertex().* After *EmitVertex();* is called the vector which *gl\_Position* is holding is added to the primitive. Further down in this method then vertex which is being held by *gl\_Position* is made equal to the position of the normal and then that normal is multiplied by the magnitude which reduces the size of the rendered line. Lastly *EmitVertex();* and *EndPrimitive();* is called. Calling both these methods one after each other will result in numerous primitives being generated. This code is demonstrated in *Figure 2.4:*

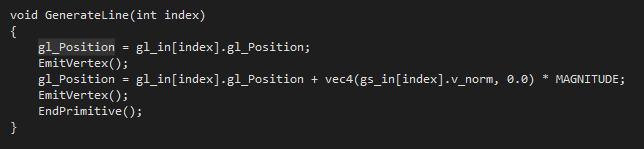


Figure 2.4

The final method in the geometry shader is the *Main* method. This method simply calls the *GenerateLine* method for each of the vertex normals.

### Fragment Shader

The fragment shader is a rather simple shader. The starting code assigns the layout qualifier by passing out a vector four for the fragment colour. The code then creates three different floats which specify each component of the colour vector. This was done so a random colour is assigned each frame.

Inside the *main* method the random colours are assigned to a vector four which is then assigned to the layout qualifier, *fragcolor*. This can be seen in *Figure 2.5* and *Figure 2.6:*

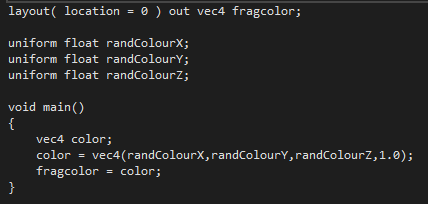


Figure 2.5

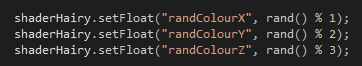


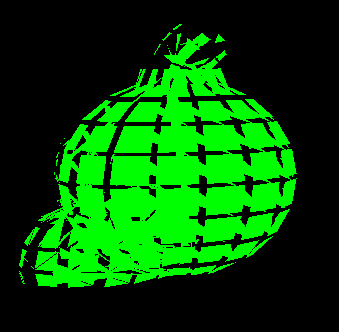
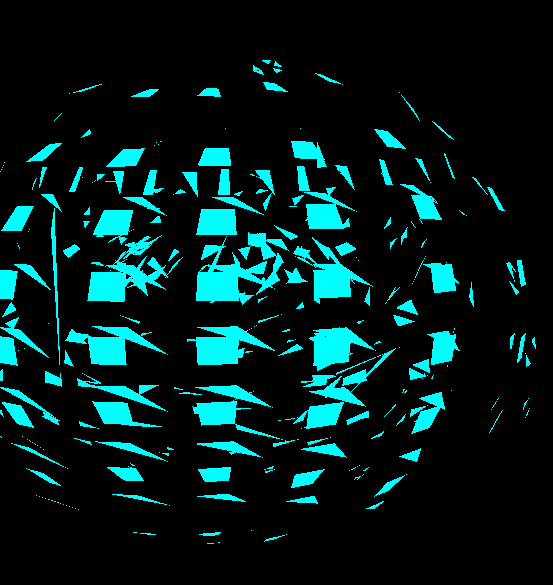
Figure 2.6

## Explosion Shader

The explosion shader allows the model to appear like it is exploding. To implement this, we move each triangle on the model along the direction of their normal vector. To link this shader to the model, only one variable has to be assigned a value. This variable was *time* which allowed the triangles to move in their specified direction over a period of time. This can be seen in *Figure 2.7*:



Figure 2.7



Effect 3.0 Effect 3.1

*Effect 3.1* and *Effect 3.2* are examples of the explosion shader before and after the normals have been manipulated.

### Vertex Shader

The vertex shader which the explosion shader uses is identical to the vertex shader used in the fog shader which was discussed in [*Section 2.1.1*](#_Vertex_Shader)*.*

### Geometry Shader

To begin the geometry shader the code assigns the layout qualifiers by passing in triangles and passing out the max vertices and triangle strips. Triangle strips are a chain of connected triangles which share the same vertices. The code then has the texture coordinates passed in and creates a variable which specifies passing out the texture coordinates. Following this, the time variable is created which has been discussed in [*Section 2.3*](#_Explosion_Shader)*.*

The first method used in the geometry shader, *explode*, returns a vector four and has two parameters: a vector four and a vector three which resembles the positon vector along the path of the normal vector. In this method a magnitude float is assigned which is used to amplify the how far each triangles travels across its direction. The direction is then calculated using a vector three. The direction calculation consists of a *sin* function which is used to prevent the triangles moving inwards on the object by changing the range of *sin* from -1, 1 to 0, 1. The value produced by the *sin* function is then multiplied by the magnitude and normal vector. This method can be seen in *Figure 2.8:*

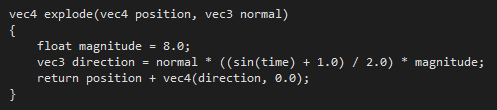


Figure 2.8

The next method, *GetNormal*, is used which returns a vector three. In this method the code computes a vector that is opposite to the exterior of the triangle while only utilizing the three vectors which we have access to. To do this calculation, two vector threes are created: *vector a* and *vector b*. Each vector is then made equal to the subtraction of two other vectors which are parallel to the plane. The cross product is then used which contains the values of *vector a* and *vector b* to find the normal vector. These calculations can be seen in *Figure 2.9:*

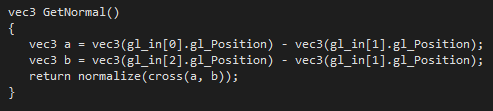


Figure 2.9

The last method inside this shader is the *main* method. This method first creates a new variable for the normals which is made equal to the vector returned by the *GetNormal* method. The position of the current vertex is then made equal to the returned vector four in the *explode* method. The parameters for the *explode* method are the current vertex positon and the new normal variable. Following this, the texture coordinate variable is set and *EmitVertex()*; is called. This process is repeated twice more but with one difference. The position of the vertex is set to the new current vertex as well as the texture coordinate. Finally, *EndPrimitive();* Is called which allows for multiple primitives to be generated. This explanation is highlighted in *Figure 2.10:*

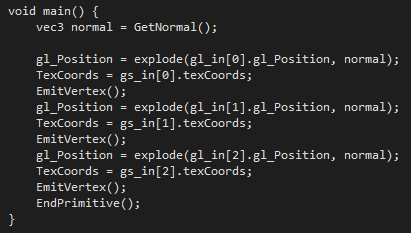


Figure 2.10

### Fragment Shader

The fragment shader which the explosion shader uses is identical to the fragment shader used in the visualizing normal shader which was discussed in [*Section 2.2.3*](#_Fragment_Shader)*.*

## Phong Lighting Shader

The Phong lighting shader is an accumulation of three different types of light: ambient, diffuse and specular reflectivity. Each effect is assigned their own colour value. If the colours are bright this represents a higher amount of reflectivity. The final calculated colour value is then the sum of the interactions between the light and material.

Ambient light doesn’t come from any particular light direction; however, it does have an original light source. The objects which are illuminated by ambient light are evenly lit in all surfaces in all directions. To calculate the contribution that ambient light makes to the final colour, the ambient material property is scaled by the ambient light values.

Diffuse light is a directional light source. The reflected light is scattered equally in all directions and appears the same to all viewers no matter their position. The strength of diffuse light depends on the angle between the light source and the surface normal. This can be calculated by using the *dot* *product* between these two variables.

Specular light is like diffuse light but is much more precise, it has highlight directional properties. The light interacts more sharply with the surface and in a particular direction. The reflected light is concentrated along the direction of the perfect reflector. The light is visible to the viewer if they are within the angle between the perfect reflect and the viewer.

Implementing the Phong lighting required a moving light source to be added which would highlight the affect the shader has on a stationary model. This required two different shaders to be constructed: a shader for the light source and one for the light effect.

To link the lighting effects to the model, many variables had to be declared and associated in the *MainGame* class. These variables can be seen in *Figure 2.11:*

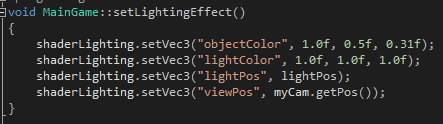


Figure 2.11

The first variable that is assigned is *objectColor* this simply assigns a colour to the model that is going to be illuminated. The next variable *lightColor* specifies the colour of the lighting effect. The *lightPos* variable identifies the position of the light source. Lastly, *viewPos* identifies the view direction which interacts with the diffuse and specular lighting.

### Vertex Shader

The vertex shader which the Phong lighting shader uses is identical to the vertex shader used in the fog shader which was discussed in [*Section 2.1.1*](#_Vertex_Shader)*.*

### Fragment Shader

#### Light Source

The light source fragment shader is extremely simple. The shader simply allocates a colour for the model which is this case is white. The code first assigns a layout qualifier. It obtains the attribute zero in the shader program and assigns the values to the variable called *fragColor.* The only method in this shader is the *main* method. Is this method the colour is assigned. This can be seen in *Figure 2.12:*

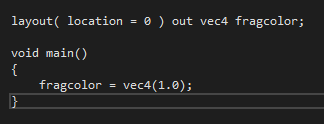


Figure 2.12

#### Lighting Effect

The lighting effect fragment shader is more complicated than the light source, this is due to the different lighting effects encompassed by my this shader. The code starts in a similar way by identifying the same layout qualifier that is in the light source fragment shader. Following from this the vertex normals and fragment position are passed in. Once the values have been passed in, the uniform variables are declared. These are the variables which are assigned in the *MainGame* class and can be seen in [*Section 2.4*](#_Phong_Lighting_Shader).

The main body of the code resides inside the *main* method. In this method the ambient, diffuse and specular lighting effects are implemented.

The ambient light is calculated by creating a float variable, which identifies the ambient strength and a vector three variable which identifies the ambient light. The ambient light is calculated by multiplying the strength by the light colour value. This calculation can be seen in *Figure 2.13:*



Figure 2.13

Diffuse light is computed after the ambient light. Diffuse light consists of creating three, vector threes and one float variable. The first vector three identifies the surface normal, which is then normalized. The next vector is the light direction. The light direction is calculated by the subtraction between the light position and fragment position. This value is then normalized so it points in the same direction. The code then declares the diffuse float variable which returns the largest value. This variable is calculated using the dot product between the surface normal and the light direction. Lastly, the final diffuse value is calculated. This is done by multiplying the diffuse float variable by the light colour. *Figure 2.14* reinforces these computations:

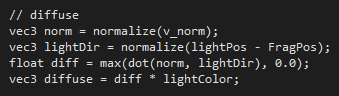


Figure 2.14

The final lighting effect is specular. The specular lighting effect starts by declaring a strength for the light. The view direction is then calculated; this is achieved by subtracting the view direction from the fragment position. The reflecting direction vector is then specified. To produce an accurate reflecting direction value, the *reflect* function is used. The reflect function calculates the reflection direction for an incident vector. The function assumes that the first vector, *lightDir*, is point from the light source towards the fragments position. To fulfil this, the vector, *lightDir* is made negative. The second argument which the *reflect* function requires is a normal vector, so the *norms* vector is used. The remaining code in the specular light section calculates the specular component. To calculate this the dot product is used between the view direction and reflection direction values. This value is then raised to the power of sixteen using the *pow* function. The *pow* function simply returns the value of the first parameter raised to the power of the second. The value is raised to the power of sixteen because sixteen is the shininess value of the highlight. If the sixteen value is increased, then the reflecting light is more precise. This can be seen in *Figure 2.15 Figure 2.16:*



Figure 2.16, Reflecting Level 256



Figure 2.15, Reflecting level 16

The final aspect of specular lighting is to calculate the final value. This is achieved by the multiplication between the specular strength variable, the specular value previously calculated and the light colour variable. The code used to formulate this result can be seen in *Figure 2.16:*

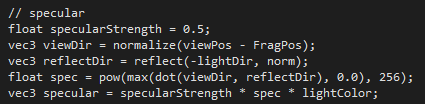


Figure 2.17

To achieve the final Phong effect, the three different effects are added together and multiplied by the objet colour.

## RimToon Shader

The RimToon shader is a combination of both rim-lighting and toon lighting effects.

Rim Lighting, also known as back-lighting, is a lighting effect that simulates the bleeding of light around the exterior of an object. The rim lighting effect is produced by determining how close the view direction comes to glancing the surface. If the view direction is facing the surface, the view vector will be collinear to the surface normal. Consequently, the rim lighting effect will be minimum. If the view direction vector glances the surface, the surface normal and view vector will be almost perpendicular. This means that the rim lighting effect will be greatest.

Toon shading also known as cell shading is a non-photorealistic lighting effect that changes the colour output using less shading colour than a shading gradient. This is achieved using a one-dimensional texture map as a lookup table to fill geometry with a solid colour from the texture map. To implement toon shading, the dot product is used between the values of the light direction and model normals.

To link this shader to the *MainGame* class many variables are assigned. These varaibles can be seen in *Figure 2.18:*

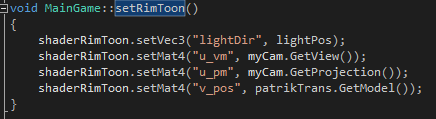
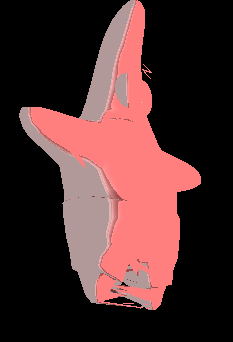


Figure 2.18

The first variable to be declared is the *lightDir*, this variable is simply the light direction. In this case, the direction is set to a moving light source. The next three variables are matrices and are used to pass in data regarding the project, view and model position.



Effect 4.0 Effect 4.1

*Effect 4.0* and *Effect 4.1* highlight the RimToon effect at different light source positions.

### Vertex Shader

The vertex shader which the RimToon shader uses is identical to the vertex shader used in the fog shader which was discussed in [*Section 2.1.1*](#_Vertex_Shader)*.*

### Fragment Shader

The RimToon fragment shader starts by declaring four uniform matrices, *u\_pm*, *u\_vm*, *v\_*pos, and *lightDir*. These are discussed in [*Section 2.5*](#_RimToon_Shader). The next part of the fragment shader assigns the layout qualifier. This obtains the attribute zero in the shader program and assigns the values to the variable called *fragcolor*. The code then passes in the model normal data.

In the *main* method, the rim lighting contribution is calculated. This is accomplished by converting the normals into the view space which is achieved creating a vector three which hold the value of the view matrix multiplied by the surface normals. The position in the clip space is then needed, another vector three is used. This vector contains the formula: the projection matrix multiplied by the model position. A vector towards the eye positon is then created which is normalized which allows the vector to point in the correct direction. Finally, the rim lighting contribution is calculated. This is computed using the dot product between the eye position and the vector which holds the value for the converted normals. *Figure 2.19*, reinforces the calculations previously described:

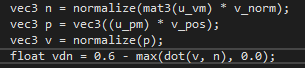


Figure 2.19

The next stage of the *main* method is to implement toon shading. To implement toon shading a float variable is created which contains the intensity of toon shading. To calculate the intensity, the dot product is used between the light direction and model normals. Depending on the block values of the intensity the colour is changed. This is done using a series of *if* statements. The toon shading effect can be seen in *Figure 2.20:*

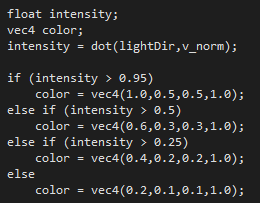


Figure 2.20

The last step of this shader is to apply the colour values to the layout qualifier, *fragcolor*. To apply the gradual colour change, the function *smoothstep* is used. This is used to gradually move the colours over. *Smoothstep*, performs Hermite interpolation between two values. Hermite interpolation is a technique which allows to interpolate values as a polynomial function.

## Ripple Shader

The ripple shader is intended to give the model a water like ripple effect. For the ripple effect to be successful, textures were required to be passed in and linked in the main game. Currently the only variable that is assigned in the *MainGame* class is *time*. The time variable determines how long the ripple will last. This can be seen in *Figure 2.21:*



Figure 2.21

### Vertex Shader

The vertex shader which the Ripple shader uses is very similar to the vertex shader used in the fog shader which was discussed in [*Section 2.1.1*](#_Vertex_Shader).

### Fragment Shader

The fragment shader begins with declaring the uniform float *time*, which was discussed in [*Section 2.6*](#_Ripple_Shader). Following this the fragment shader assigns the layout qualifier. This obtains the attribute zero in the shader program and assigns the values to the variable called *fragcolor*. The next part of the code creates uniforms variables for the textures which should be applied to the model and a float is made why resembles the radius of each ripple. An interface block is then created which passes in the texture coordinates and then a variable is made to pass the texture coordinates out.

Within the *main* method, the ripple effects are handled.

The first line of code in the ripple *main* method creates a float *T* which has its values clamped between zero and one. Further on, the passing out texture coordinate variable is set to the texture coordinate that is being passed in and a vector two variable which specifies the direction is make equal to the texture coordinate. The next section of code returns the distance between the texture coordinate and a vector two. An offset is created which is used to distort the ripple effect, to allow for a realistic effect. The texture coordinate is made equal to the next vertex plus the offset and the textures are applied to a vector four. Lastly,the textures and texture coordinates are held within a vector four and the final fragment colour is applied.

# Appendix

##### Model Source

Turbosquid.com. (2018). free mario bros 3d model. [online] Available at: https://www.turbosquid.com/FullPreview/Index.cfm/ID/621837 [Accessed 16 May 2018].

Turbosquid.com. (2018). free light bulb 3d model. [online] Available at: https://www.turbosquid.com/FullPreview/Index.cfm/ID/505822 [Accessed 16 May 2018].

Turbosquid.com. (2018). free low-poly character 3d model. [online] Available at: https://www.turbosquid.com/FullPreview/Index.cfm/ID/742463 [Accessed 16 May 2018].

Turbosquid.com. (2018). free obj model star. [online] Available at: https://www.turbosquid.com/FullPreview/Index.cfm/ID/656961 [Accessed 16 May 2018].

Turbosquid.com. (2018). egg cartoon character obj free. [online] Available at: https://www.turbosquid.com/FullPreview/Index.cfm/ID/859564 [Accessed 16 May 2018].

##### Shader Source

Learnopengl.com. (2018). LearnOpenGL - Basic Lighting. [online] Available at: https://learnopengl.com/Lighting/Basic-Lighting [Accessed 16 May 2018].

Learnopengl.com. (2018). LearnOpenGL - Geometry Shader. [online] Available at: https://learnopengl.com/Advanced-OpenGL/Geometry-Shader [Accessed 16 May 2018].

Blackboard.gcal.ac.uk. (2018). Lectures. [online] Available at: https://blackboard.gcal.ac.uk/webapps/blackboard/content/listContent.jsp?course\_id=\_87992\_1&content\_id=\_2295776\_1 [Accessed 16 May 2018].

Gist. (2018). Ripple effect for GLSL. [online] Available at: https://gist.github.com/alco/3070640 [Accessed 16 May 2018].