Graphics Programming

Conall Mcluskie, S1509449

*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*.



Contents

[Section 1 Overview 4](#_Toc514099003)

[Section 1.1 Controls 4](#_Toc514099004)

[Section 1.2 Code Structure 4](#_Toc514099005)

[Section 1.2.1 Initialisation 4](#_Toc514099006)

[Section 1.2.2 Draw 5](#_Toc514099007)

[Section 2 Shader Class 6](#_Toc514099008)

[Section 2.1 Setting Uniforms 6](#_Toc514099009)

[Section 2.2 Initialising 6](#_Toc514099010)

[Section 3 Rim-Toon Shading 8](#_Toc514099011)

[Section 3.1 Rim Shading 8](#_Toc514099012)

[Section 3.1.1 Overview 8](#_Toc514099013)

[Section 3.1.2 Vertex Shader 8](#_Toc514099014)

[Section 3.1.3 Fragment Shader 8](#_Toc514099015)

[Section 3.2 Toon Shading 9](#_Toc514099016)

[Section 3.2.1 Overview 9](#_Toc514099017)

[9](#_Toc514099018)

[Section 3.2.2 Vertex Shader 9](#_Toc514099019)

[Section 3.2.3 Fragment Shader 9](#_Toc514099020)

[Section 3.3 Combination 9](#_Toc514099021)

[Section 3.3.1 Overview 9](#_Toc514099022)

[Section 3.3.2 Vertex Shader 9](#_Toc514099023)

[Section 3.3.3 Fragment Shader 10](#_Toc514099024)

[Section 3.4 Linking 10](#_Toc514099025)

[Section 4 Explode Shader 11](#_Toc514099026)

[Section 4.1 Overview 11](#_Toc514099027)

[Section 4.2 Fragment Shader 11](#_Toc514099028)

[Section 4.3 Geometry Shader 11](#_Toc514099029)

[Section 4.4 Linking 11](#_Toc514099030)

[Section 5 Fog Shader 12](#_Toc514099031)

[Section 5.1 Overview 12](#_Toc514099032)

[Section 5.2 Vertex Shader 12](#_Toc514099033)

[Section 5.3 Fragment Shader 12](#_Toc514099034)

[Section 5.4 Linking 12](#_Toc514099035)

[Section 6 Blinn-Phong Lighting 13](#_Toc514099036)

# Section 1 Overview

## Section 1.1 Controls

|  |  |
| --- | --- |
| **Key** | **Shader** |
| 1 | Blur |
| 2 | Rim |
| 3 | Toon |
| 4 | Rim-Toon |
| 5 | Normal |
| 6 | Explode |
| 7 | Fog |
| 8 | Blinn-Phong |
| 9 | Blend |
| 0 | Standard |

To see all the shaders you will have to press a number key to change shader types. This is done using a switch statement (figure 1.1) whilst processing input which will change an enumerator variable to the desired shader.

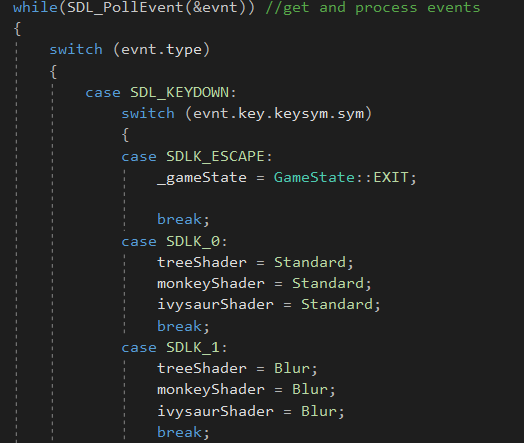


Figure 1.0

Figure .1

## Section 1.2 Code Structure

### Section 1.2.1 Initialisation

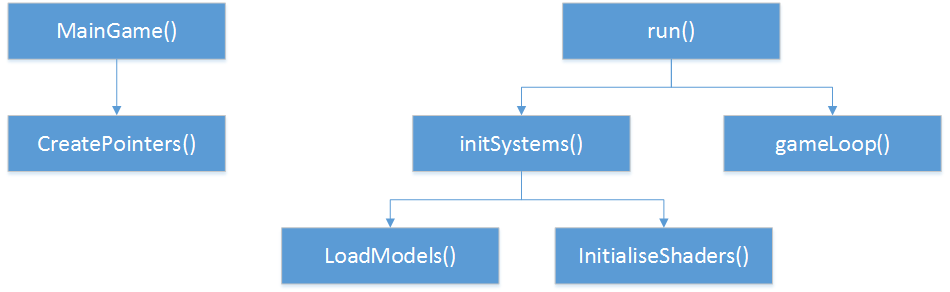


Figure 1.1

Figure 1.2 shows the flow of the Main Game class when initialising. Previously pointers were created in the constructor for the class, whereas now they get created in a new method that gets called in main game. The same has been done for the Load Models and Initialise Shaders methods when loading the models and shader programs respectively from files.

### Section 1.2.2 Draw

Every time the draw method gets called by the game loop every model has the same flow (see figure 1.3). The first method in this cycle is the update transform method, where there are three vector3 parameters. These are used for the world position, the model’s rotation, and the models scale. This will only update Three transforms that are used solely to hold data for each of the models. To apply the changes to the mesh the Update Model method is called which copies over the data.

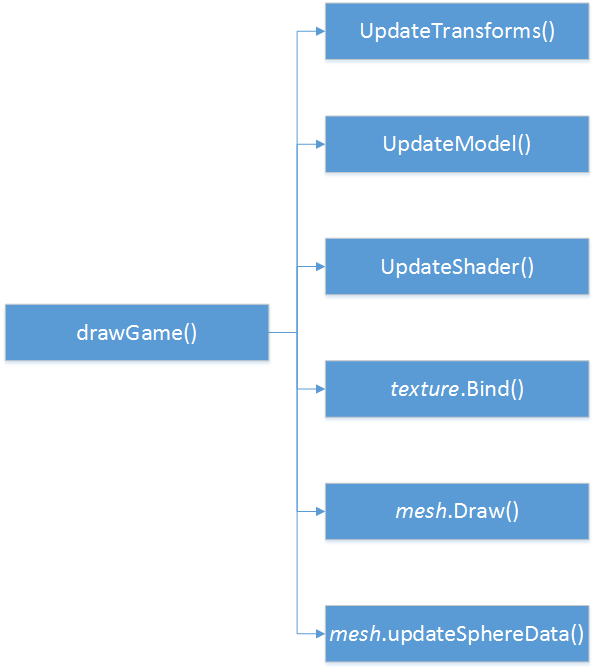


Figure 1.2

To apply the chosen shader to each mesh the update shader method is used. In this method the shader gets bound, uniforms are set, and finally updated. This is handled using a switch statement (see figure 1.4) and passing in a shader type enumerator as an argument. After this a texture is bound for the shaders that utilise one. And finally, the mesh is drawn with the shader applied to it and the sphere data is updated with the new transform.

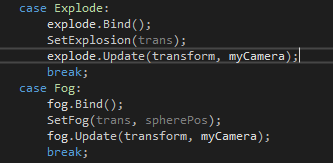


Figure 1.3

# Section 2 Shader Class

## Section 2.1 Setting Uniforms

To properly render the shaders they first need to be linked. To do this glUniform is needed. This method will set the value of a uniform in a shader. Figure 2.1 shows a method where a Boolean uniform is set. There are multiple variants of this method and are defined by the 1 and the *i*. 1 dictates that a float, integer, unsigned integer, or a Boolean is going to be set, 2 dictates your setting vector and matrix 2 variables, 3 is vector and matrix 3 variables, and 4 is vector and matrix 4 variables. Where the *i* dictates that an integer value is being passed through, alternatively an *f* is used for a floating-point value, and *ui* for an unsigned int.

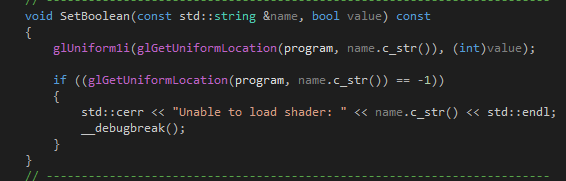


Figure 2.0

glUniform requires a location for a uniform to be passed in, therefor glGetUniformLocation is used. To get the location, the shader program (program, figure 2.1, the name of the uniform variable in the shader(name.c\_str(), figure 2.1), the amount of values to get passed (1, figure 2.1), and finally the values (&value[0], figure 2.1) all need to be passed in. In figure 2.1 a Vector2 is getting passed into a shader.



Figure 2.1

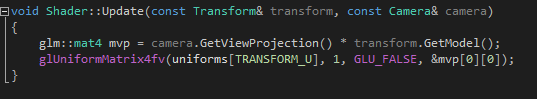


Figure 2.2 The update function links the transform uniform

2.2

## Section 2.2 Initialising

Two initialising methods are needed; one to load a shader with just vertex and fragment files, and another overloaded method that’s also loads a geometry file (as shown in figure 2.3).

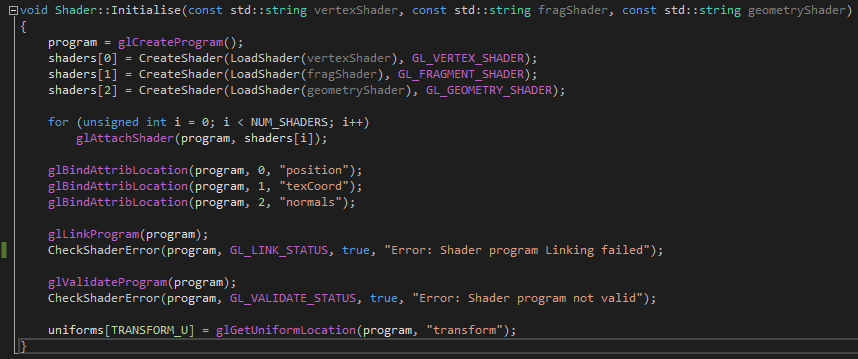


Figure 2.3

To create the shader program the individual components, vertex, fragment (and in this case geometry) need to be loaded and from their files. After we load the files they are attached to the program. Attribute locations are bound so they can be read in the vertex shader as shown in figure 2.4.



Figure 2.4

The program is then linked and validated and the uniform array is set to the shader’s transform uniform.

# Section 3 Rim-Toon Shading

## Section 3.1 Rim Shading

### Section 3.1.1 Overview



Figure

Rim shading (or back lighting) is an effect where shading is an effect where the outside of an object is lit to emphasise an object against a background.

The shading is calculated using the normal and view directions. The rim lighting’s intensity will be at its lowest when the view vector and the surface normal are collinear. When the surface normal and view direction are perpendicular the intensity will be at its highest.

### Section 3.1.2 Vertex Shader

The vertex shader (figure 3.1) for this is the same as all the other shaders unless otherwise stated.

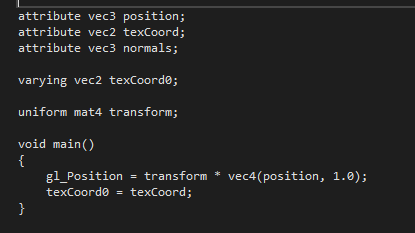


Figure 3.1

### Section 3.1.3 Fragment Shader

In the first line of code (figure 3.2) the view matrix is multiplied but the vertex model matrix, and normalised (n). The view projection and model matrix are then multiplied together to then normalise (v) also. Now the dot product of v and n need to be calculated. However, as previously mentioned, if the two vectors are collinear then this will return one as the dot product. This can be troublesome when the angle’s size increases the value will decrease. To remedy this; the dot product is subtracted from one to ensure the rim shading is larger when the angles are larger. To ensure the colours have a smooth cut off smoothstep is used when setting the output colour.

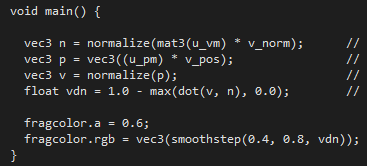


Figure 3.2

## Section 3.2 Toon Shading

### Section 3.2.1 Overview

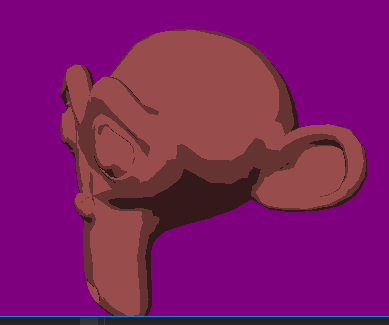


Figure 3.4

Toon Shading (or cel shading), is not a photorealistic shader. Toon shading (figure 3.4) will change the colour out using a full colour as opposed to a gradient of colours. A shading effect could be created using the relationship between light and a vertex normal.

### Section 3.2.2 Vertex Shader

As previously mentioned the vertex shader is the same as the one used in [Section 3.1.2](#_Section_3.1.2_Vertex).

### Section 3.2.3 Fragment Shader

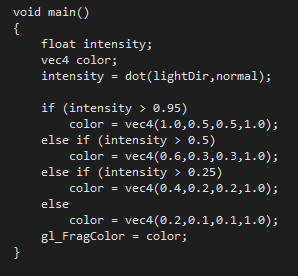


Figure 3.5

the exact colour of the geometry is dictated by the intensity of the light. This is calculated by passing in the direction the light is heading in and the surface normal of each vertex. The colour is then set accordingly and then is output.

## Section 3.3 Combination

### Section 3.3.1 Overview

The Rim-Toon shader (figure 3.6) is a combination of the individual rim and toon shaders. Therefore, the theory behind this shader has already been discussed in sections [3.1.1](#_Section_3.1.1_Overview), and [3.2.1](#_Section_3.2.1_Overview). The VDN float is used to create the rim effect and the intensity float is used to dictate the colour of the toon effect again

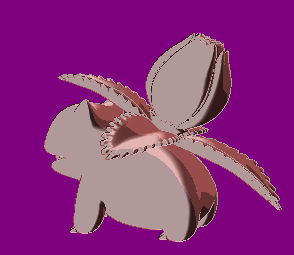


Figure 3.6

### Section 3.3.2 Vertex Shader

As previously mentioned the vertex shader is the same as the one used in [Section 3.1.2](#_Section_3.1.2_Vertex).

### Section 3.3.3 Fragment Shader

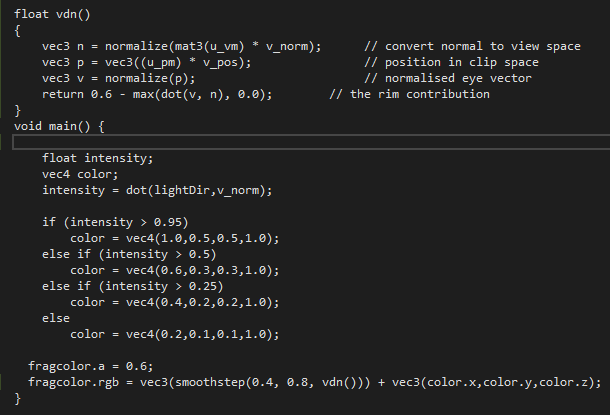


Figure 3.7

As shown in figure 3.7 the Rim shading portion has been put into its own function to return the *vdn* value. The code has previously been explained in [section 3.1.3](#_Section_3.1.3_Fragment). The main method is largely the same as the toon shader, with the only difference being the line in which the fragment colour is set (figure 3.8).

The frag colour that gets returned is simply just the values that are used for both rim and toon individually added together. It is worth noting that if these vectors are multiplied together a rim shading effect will be achieved as opposed to rim lighting.

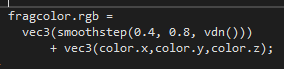


Figure 3.8

## Section 3.4 Linking

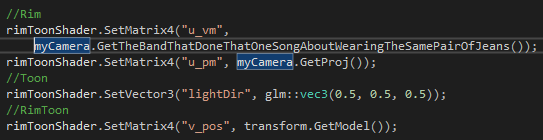


Figure 3.9

The *u\_vm* and *u\_pm* components of the shader are required for the rim portion. Whereas the light direction is passed in for the intensity in the toon portion. *v\_pos* is also passed in specifically for the Rim-Toon shader to allow the rim portion of the colour to be updated.

# Section 4 Explode Shader

## Section 4.1 Overview



Figure

The explode shader (figure 4) is a geometry shader. This will take the triangles of the mesh and, over time, separate them from each other. This then gives the effect that an explosion has taken place at the center of the model.

This effect is achieved by moving each triangle of away from the normal of the model.

## Section 4.2 Fragment Shader

The fragment shader used for this is the [Rim-Toon fragment shader](#_Section_3.3.3_Fragment).

## Section 4.3 Geometry Shader

the normal vector of each triangle needs to be calculated. To do this a vector perpendicular to the triangle needs to be found. This can be done using the cross product. Using figure 4.1 as a reference: vectors *a* and *b* are found to be perpendicular to each triangle. Then the normalized cross product is returned as the normal. This is done to ensure each triangle moves away in the same, relative, direction.

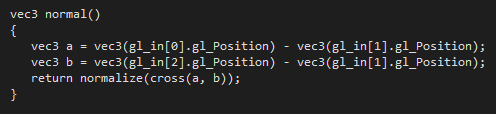


Figure 4.1

Using the normal of each triangle, as calculated above, each triangle is moved in that direction as calculated in figure 4.2. the sin function is used to ensure the model explodes outwards then loops back to its original position. The speed of this is dictated by the magnitude float. The direction that was just calculated is added to the position and returned to the main method (figure 4.3). The first thing that is done in the Main method is calculating and setting the normal so it doesn’t need to be recalculated constantly. After that the position of each point is set by calling the Explode function, and emits the variable to the output primitive. This is done three times as there are three vertices in a triangle.

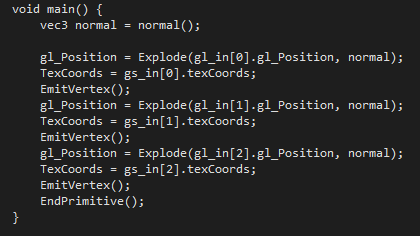


Figure 4.3

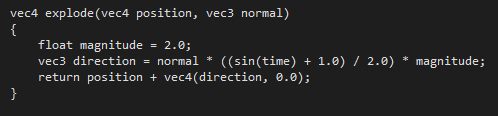


Figure 4.2

## Section 4.4 Linking

linking this shader is largely the same as the [Rim-Toon shader](#_Section_3.4_Linking), with the only discernible difference being that the time is passed in using the counter (which gets updated every time the game loops). The *15* that the counter is multiplied by changes how fast the model will explode.

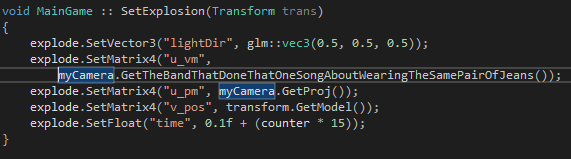


Figure 4.4

# Section 5 Fog Shader

## Section 5.1 Overview



Figure

The fog shader will draw less of the models detail the further away it gets from the camera. Figure 5 depicts the monkey’s head model at a large distance from the camera. This is done by using a minimum distance from the camera for the fog to start, and a maximum distance where the effect will stop getting more intense. Fog effects can be achieved by combined the colour of a fragment with a static fog colour.

## Section 5.2 Fragment Shader

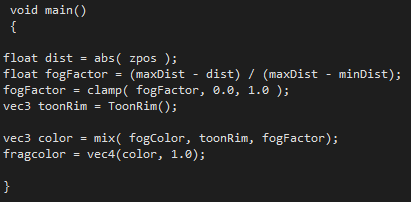


Figure 5.1

The fog effect will be applied to the [Rim-Toon](#_Section_3.3.3_Fragment) effect discussed earlier. For convenience this is in its own function called ToonRim in figure 5.1. first the absolute value of the model’s *z* position is used- it is worth noting this can also be the model’s *x* or *y* position as well, although *z* works best. The absolute value is required to ensure that a negative distance isn’t given.

The fog factor then needs to be calculated; this is done using the equation in figure 5.2 where *f* is the fog factor. This dictates the amount the fog will affect the Rim-Toon shading. The resultant is then clamped to ensure it is between a range of 0 to 1. For example, if the value of fog factor was less than 0, then it would be raised to 0, and f it was greater than 1 it would be lowered to 1. Finally, the last calculation will define the final colour and this is done my using mix to interpolate the fog factor between the fog colour and the rim-toon effect, where the fog factor acts as a weight between the two vectors.

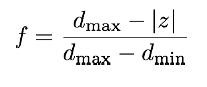


Figure 5.2

## Section 5.3 Linking

Figure 5.3 shows the method used to pass the uniforms into the fog shader with the rim-toon omitted due to it previously being covered. When the *zpos* is linked it’s, essential this is referring to a value that is related to the model. Here the sphere position has been passed in to ensure the correct value is consistently getting passed in.

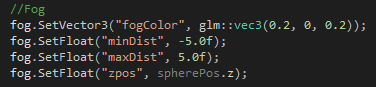


Figure 5.3

# Section 6 Blinn-Phong Lighting

## Section 6.1 Phong Lighting

Phong lighting is a simple, computationally inexpensive, and commonly used lighting model. it works off ADS (ambient, diffuse, and specular). The ADS components are given colour values where the brighter colours represent more reflectivity. Phong lighting requires a light source and if this is a physical model it is also given a colour to represent the brightness of the source. The output colour is calculated by adding the lighting and material interactions of ADS.

### Section 6.1.1 Ambient

Ambient light doesn’t come from one direction, although it has a light source somewhere. Ambient light will reflect off all surfaces. To cheaply code this a light colour is multiplied by a consistent light factor and use that as the fragment colour.

### Section 6.1.2 Diffuse

Diffuse will give the fragments a lighter colour the closer they are to the light source. Its calculated by finding the angle at which the light hits the fragment at. The closer the light ray is to being perpendicular to the object the greater the light is. To find said angle the normal is calculated - the normal is the perpendicular vector to the fragment (figure 6).



Figure

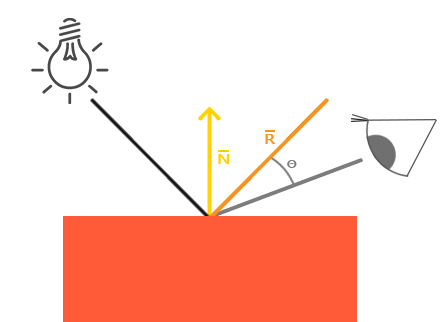


Figure 6.1

### Section 6.1.3 Specular

Similarly, to diffuse, specular lighting requires a light source and is calculated using normal vectors. However, specular light also utilises the direction that the camera is pointing. The surface normal of the fragment is considered a mirror. The lighting will be strongest where it would be reflected. The light ray is reflected in the normal. The view vector for specular is calculated using the camera’s world position and the fragments position. This is depicted in figure 6.1.

## Section 6.2 Blinn-Phong Lighting

Blinn-Phong is an extension of the Phong lighting model, and an optimisation of it. The only difference is when the specular light is calculated. As opposed to calculating the reflection vector, a vector that’s is halfway between the view direction and light direction is calculated. We call this the halfway vector. The closer this vector aligns with the surface normal the higher the specular component is.

## Section 6.3 Fragment Shader

Figure 6.2 shows the ADS portion of the lighting being calculated. For the ambient portion two vectors are created to store the colour values; one to set the initial colour of the model, and another to add the ambient effect. In this example the constant light factor is *0.05.* after this, the diffuse portion is calculated. The light direction vector is found by normalising the distance between the light and the vertex position and the vertex normal is normalised. The diffuse amount is found by taking the dot product of both the light direction and the normal. Then comparing it to *0.0,* and using the max function, which returns the highest value. The diffuse vector is set by multiplying the colour that was set at that start by the diffuse amount. At the end of the ADS portion of the shader the specular component needs to be calculated. Values previously calculated for diffuse can be reused. Two new vectors are needed the view, and the reflect direction. The view direction is calculated by normalising the distance between the view and vertex positions. The reflect direction is calculated using the reflect function. The inverse light direction is used as the incident vector, and the normal as the normal vector. The float for the specular value is defined before it is set by the Phong or Blinn-Phong.

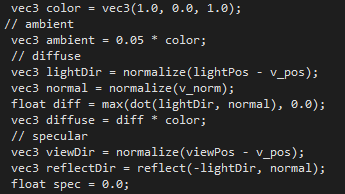


Figure 6.2

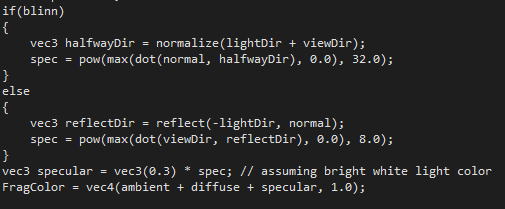


Figure 6.3

At this point is where Blinn-Phong differs from Phong. As seen in figure 6.3; a Boolean has been set that will decide whether Phong or Blinn-Phong is used. First for Blinn-Phong: the halfway vector is calculated by normalising the sum of the light direction and view direction. The specular value is calculated in three steps. First is getting the dot product of the normal and halfway direction. This is then compared against 0 using max. this value is then raised to the power of 32 using the pow function. Whereas, when calculating the Blinn the process is largely the same with only two differences; the halfway direction is the reflection direction, and instead of raising the max value to the power of 32, it is raised to 8. The specular component is then calculated by multiplying a vector 3 by the specular amount.

Finally, the ambient, diffuse, and specular components are added together and this is what gets set as the Fragment Colour.

## Section 6.4 Linking

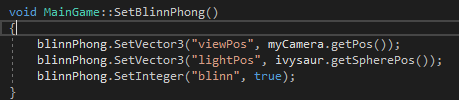


Figure 6.4

The light position variable is set to the ivysaur model’s position to make this model a light source. The Blinn Boolean is also set to true to ensure that Blinn-Phong is used.