Graphics Programming  
Coursework

Eugén Cowie (S1714780)  
BSc Computer Games (Software Development)  
  
[ecowie202@caledonian.ac.uk](mailto:ecowie202@caledonian.ac.uk)

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.

– Eugén Cowie

**Table of Contents**

[1 Game Framework 3](#_Toc514247162)

[1.1 Game class 3](#_Toc514247163)

[1.2 Game class constructor 3](#_Toc514247164)

[1.3 Run method 4](#_Toc514247165)

[1.4 Update and render methods 4](#_Toc514247166)

[2 Models 6](#_Toc514247167)

[2.1 Model class 6](#_Toc514247168)

[2.2 Model constructor 6](#_Toc514247169)

[3 Shaders 8](#_Toc514247170)

[3.1 Colourful shader 8](#_Toc514247171)

[3.2 Phong lighting shader 9](#_Toc514247172)

[3.3 Toon lighting shader 15](#_Toc514247173)

[4 Camera 17](#_Toc514247174)

[4.1 Camera class 17](#_Toc514247175)

[5 Shader class 18](#_Toc514247176)

[5.1 Shader class 18](#_Toc514247177)

[5.2 Program class 18](#_Toc514247178)

[6 Shader implementation methods 19](#_Toc514247179)

[Appendix: Shader switching methods 20](#_Toc514247180)

[Appendix: TinyObjLoader model processing 21](#_Toc514247181)

[Appendix: Compound shaders 22](#_Toc514247182)

# Game Framework

## Game class

The game class (Figure 1) is responsible for initialising and running the game. It owns and manages the window and input manager, viewport and camera, shader programs, light sources, game objects and game state.



Figure . Game class (game.hpp).

## Game class constructor

The game class constructor (Figure 2) initialises the window, sets the viewport and camera to their initial values, loads the shader programs, creates the initial light sources, creates and loads the game objects and sets the initial game state.



Figure . Game constructor (game.cpp).

## Run method

The run method (Figure 3) contains the main game loop, which will run for as long as the window should remain open. It measures the time elapsed since the previous frame and passes it to the update and render methods as a parameter.



Figure . Run method (game.cpp).

## Update and render methods

The update method (Figure 4) processes window and input events, triggers actions based on user input, updates game objects and switches to the next shader every 2 seconds. The render method clears the screen, draws all game objects and then swaps the front and back buffers.



Figure . Update method (game.cpp).

# Models

## Model class

The model class (Figure 5) is responsible for loading and managing 3D models from the filesystem.



Figure . Model class (model.hpp).

## Model constructor

The model constructor (Figure 6) loads and processes a 3D model file. It uses TinyObjLoader to loads OBJ files, which can contain multiple individual objects, so it processes each object as a separate mesh and stores it in the list of meshes.



Figure . Model constructor (model.cpp).

# Shaders

## Colourful shader

This shader uses the vertex normal vector to colourise a mesh based on the direction of each individual vertex. Vertices which face along the +X or-X axis appear red, vertices which face along the +Y or -Y axis appear green and vertices which face along the +Z or -Z axis appear blue. This produces a colourful effect which can be seen in Figure 7.

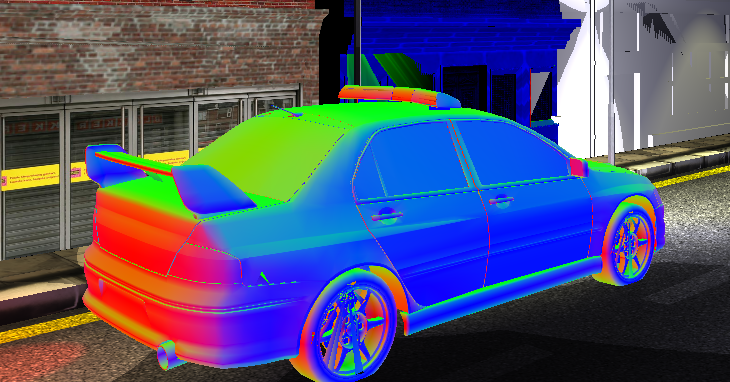


Figure . Example of the colourful shader in action.

To achieve this effect, the vertex shader takes the vertex normal vector and applies the abs function to it to get a vector containing the absolute values from the normal vector. When applied to a vector, the abs function is applied to each of the components individually and a new vector is returned, as shown by the equation in Figure 8.

Figure . Equation for applying abs to a vector.

The absolute values are needed because the normal vector can contain negative values but colour values must be positive. These values are then passed on to the fragment shader. Applying the model-view-projection matrix to the vertex position to transform it from model space to clip space is also carried out in the vertex shader. The implementation of the vertex shader can be seen in Figure 9.



Figure . Colourful vertex shader (colored.vert).

The fragment shader (Figure 10) is a passthrough shader. For efficiency, all calculations are performed on the vertex shader and the result is given to the fragment shader as an input. The fragment shader simply passes on the result from the vertex shader.



Figure . Colourful fragment shader (colored.frag).

## Phong lighting shader

This shader implements Phong lighting, which calculates the contribution of ambient, diffuse and specular lighting in a scene to provide an approximation of realistic lighting, as seen in Figure 11.

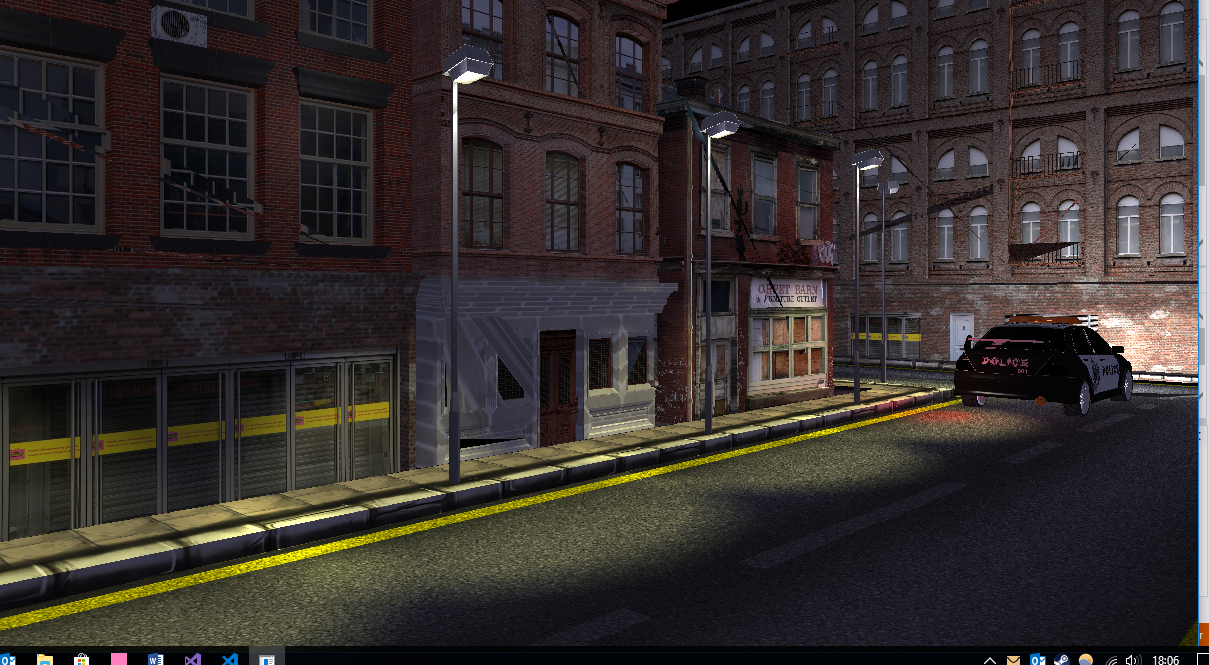


Figure . Example of Phong lighting shader in action.

The primary function of the vertex shader is to transform the vertex position and normal vectors from model space to view space, which is the coordinate space used in the fragment shader to perform the lighting calculations. In model space, coordinates are relative to the 3D model whereas in view space, coordinates are relative to the position of the camera, as seen in Figure 12. This makes the lighting calculations more efficient and simpler to implement.

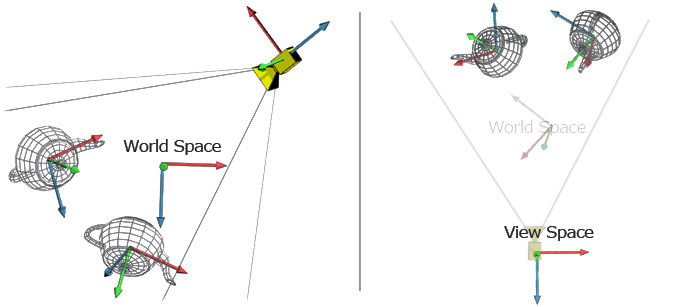


Figure . The difference between world space and view space. Source: Marco Alamia.

To transform the position from model space to view space, the model-view matrix is applied to it and the result is passed to the fragment shader. The model-view matrix is the result of combining the model and view matrices by multiplying them together. The model matrix transforms coordinates from model space to world space and the view matrix transforms coordinates from world space to view space, as seen in Figure 13. The combination of the two therefore transforms coordinates from model to view space.

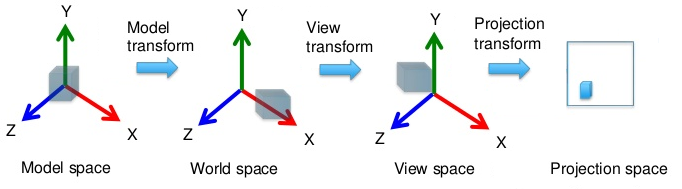


Figure . Transformation in 3D space. Source: Takao Wada.

To transform the vertex normal vector from model space to view space is not quite as simple. It is not enough to simply apply the model-view matrix to the normal because normal vectors only represent a direction and not a position. Applying the model-view matrix as-is could perform a non-uniform scale as seen in Figure 14.

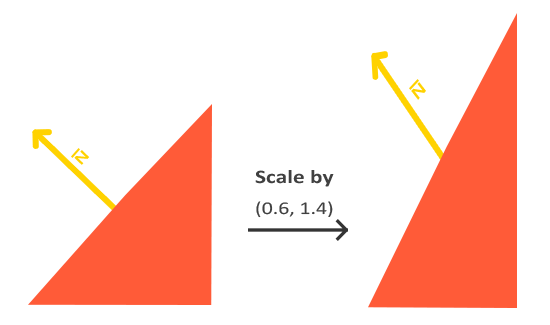


Figure . Effect of a non-uniform scale on a normal vector. Source: Joey de Vries.

Instead, a special matrix called the normal matrix is needed. The normal matrix is defined as “the transpose of the inverse of the upper-left corner of the model-view matrix”. Calculating this matrix is a computationally expensive operation so it is only performed once per mesh on the CPU in our application code, as seen in Figure 15.



Figure . Calculating the normal matrix (mesh.cpp).

The normal matrix can be accessed from the vertex shader (Figure 16) as a uniform variable. The normal matrix is applied to the normal vector and the result is passed to the fragment shader.



Figure . Phong vertex shader (lit.vert).

The Phong lighting calculations are performed in the fragment shader, which defines several data structures to manage the values required to perform the lighting calculations. The material data structure (Figure 17) contains a shininess value which determines the level of specular highlight.



Figure . Material structure (lit.frag).

The directional light data structure (Figure 18) contains values which represent a simple directional light. The direction represents the direction that the light is pointing. The ambient, diffuse and specular values control the colour and intensity of the ambient, diffuse and specular component contributions to the final output colour.



Figure . Directional light structure (lit.frag).

The point light data structure (Figure 19) contains values which represent an omni-directional point light. The position vector is the position of the light source in world space. The linear, quadratic and constant values are used to calculate the attenuation of the light. Attenuation is the way that the intensity of the light reduces over distance.



Figure . Point light structure (lit.frag).

The spot light data structure (Figure 20) contains values which represent a directional point light or spotlight. The cut-off and outer cut-off values determine the size of the inner and outer cones of light respectively. The cut-off is the angle at which the light no longer illuminates the surface. This provides a hard edge which is not realistic, so the outer cut-off is a larger angle which is used to interpolate between the illuminated inner cone and unilluminated outer cone, providing a soft edge to the light.



Figure . Spot light structure (lit.frag).

The fragment shader takes the view-space vertex position and normal from the vertex shader and outputs a colour value. It requires access to the view and normal matrices and the material of the mesh. It applies a global ambient light before performing the Phong lighting calculations, which can support multiple directional, point and spot lights. These inputs, outputs and uniforms can be seen in Figure 21.



Figure . Inputs, outputs and uniforms (lit.frag).

The implementations of the directional, point and spot light calculations are contained in separate functions to keep the body of the main function short and readable. Details of their implementations will follow.

The main method of the fragment shader stores the normalised view-space normal vector as well as the direction from the vertex position to the camera position. Because we are using view space, the vertex position vector points from the camera to the vertex, so we can simply take its negative to get the direction from the vertex position to the camera position.

To combine the results of the various lighting calculations, the fragment shader starts by assigning the global ambient light value to an initial result. It then iterates through all directional, point and spot lights and adds the output of each directional, point or spot light calculation to the result, which is then output by the fragment shader. The main method implementation can be seen in Figure 22.



Figure . Main method (lit.frag).

When calculating the diffuse contribution of a directional light, the direction of the light needs to be transformed into view space by applying the normal matrix to it. The direction of the perfect reflection of the light from the vertex normal is needed when calculating the specular contribution.

The strength of the diffuse light depends on the angle (dot product) between the surface normal and the light direction, as shown in Figure 23. The strength of the specular light depends on the angle (dot product) between the view direction and perfect reflector, as shown in Figure 24. It is highly concentrated along the perfect reflector, which is achieved by raising it to the power of the shininess property. The implementation of the directional light calculation can be seen in Figure 25.

|  |  |
| --- | --- |
| Graphics3D_LightingDiffuse.png  Figure . Diffuse light illustration. Source: GCU. | Lambert2.gif  Figure . Specular light illustration. Source: GCU. |



Figure . Calculate directional light method (lit.frag).

The point light calculation differs from the directional light calculation in that the ambient, diffuse and specular light is multiplied by an attenuation value which causes the intensity of the light to drop off as the distance from the light increases. The equation to calculate the attenuation can be seen in Figure 26, where is the constant factor, is the linear factor, is the quadratic factor and is the distance.

Figure . Equation for attenuation of a point light.

The distance is the length of the difference between the vertex surface position and the light position in view space. The light position is transformed from world space to view space by multiplying it with the view matrix. The light angle can then be obtained by subtracting the surface position from the light position. The rest of the code is the same as the directional light calculation. The implementation of the point light calculation can be seen in Figure 27.



Figure . Calculate point light method (lit.frag).

The spot light calculation differs from the point light calculation in that the ambient, diffuse and specular light is multiplied by an intensity value which constrains the light to a circular cone with a soft edge. The equation to calculate the intensity value is shown in Figure 28, where theta is the angle (dot product) between the light direction and light angle, is the outer cut-off angle and is the difference between the inner and outer cut-off angles.

Figure . Equation for intensity of a spot light.

This value is then clamped to the 0.0-1.0 range. The rest of the code is the same as the point light calculation. The implementation of the spot light calculation can be seen in Figure 29.



Figure . Calculate spot light method (lit.frag).

## Toon lighting shader

This shader extends the result of the lighting shader by applying toon shading, which uses a lookup table which assigns solid colours to ranges of values to provide a non-photorealistic effect, as shown in Figure 30.

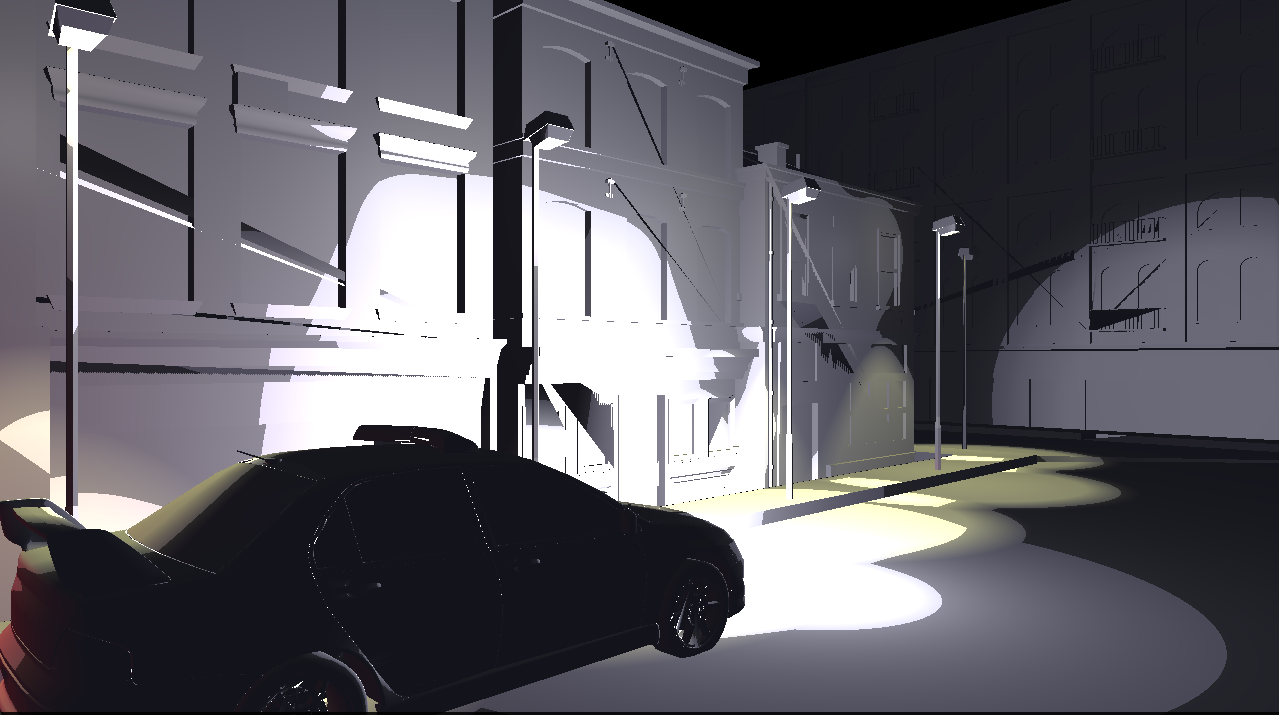


Figure . Example of toon lighting shader in action.

As the calculations in the fragment shader are also performed in view space, the vertex shader for this effect is identical to the Phong lighting vertex shader above (Figure 16, pp 11). It simply transforms the vertex position and normal vectors from model space to view space and passes them on to the fragment shader.

The fragment shader requires the same inputs, outputs and uniforms as the Phong lighting fragment shader above (Figure 21, pp 12) and the methods to calculate the directional, point and spot light contribution remain the same as well.

To achieve the toon lighting effect, the fragment shader averages the results of the lighting calculation to determine the overall intensity of the light. It then checks which range of values the intensity falls in and multiplies the original result by a constant factor associated with that range of values to get the final output. The final implementation of the main method of the fragment shader can be seen in Figure 31.



Figure . Main method (toon.frag).

# Camera

## Camera class

The camera class (Figure 32) manages the view matrix. It contains the three vectors used to generate the view matrix: the position vector, the target vector and the up vector.



Figure . Camera class (camera.inl).

# Shader class

## Shader class

The shader class (Figure 33) is a lightweight RAII wrapper around an OpenGL shader. The constructor creates an OpenGL shader of the desired type and the destructor deletes it. Passthrough functions are implemented for sending source and compiling. A secondary constructor allows the user to also provide some source code to be uploaded to the shader and compiled.



Figure . Shader class (shader.inl).

## Program class

The program class (Figure 34) is a lightweight RAII wrapper around an OpenGL shader program. The constructor creates an OpenGL shader program and the destructor deletes it. Passthrough functions are implemented for attaching/detaching shaders, linking, validating and binding/unbinding. A secondary constructor allows the user to provide a base path to a set of shaders to be loaded and attached to the program before linking and validating the program and finally detaching the shaders.



Figure . Program class (program.inl).

# Shader implementation methods

To do.

# Appendix: Shader switching methods

The game class contains four methods to assist with applying different shaders to the scene.

# Appendix: TinyObjLoader model processing

The model class contains three methods for processing models loaded with TinyObjLoader.

# Appendix: Compound shaders