**Peter Lunardi**

**Student No: S1636120**

**Computer Games (Software Development)**

**Module: Graphics Programming**

**Coursework Documentation**

*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

[Introduction 3](bookmark://_Toc8369712#_Toc8369712)

[Vertex Shader 4](bookmark://_Toc8369713#_Toc8369713)

[Fragment Shader 7](bookmark://_Toc8369714#_Toc8369714)

[MainGame.cpp 8](bookmark://_Toc8369715#_Toc8369715)

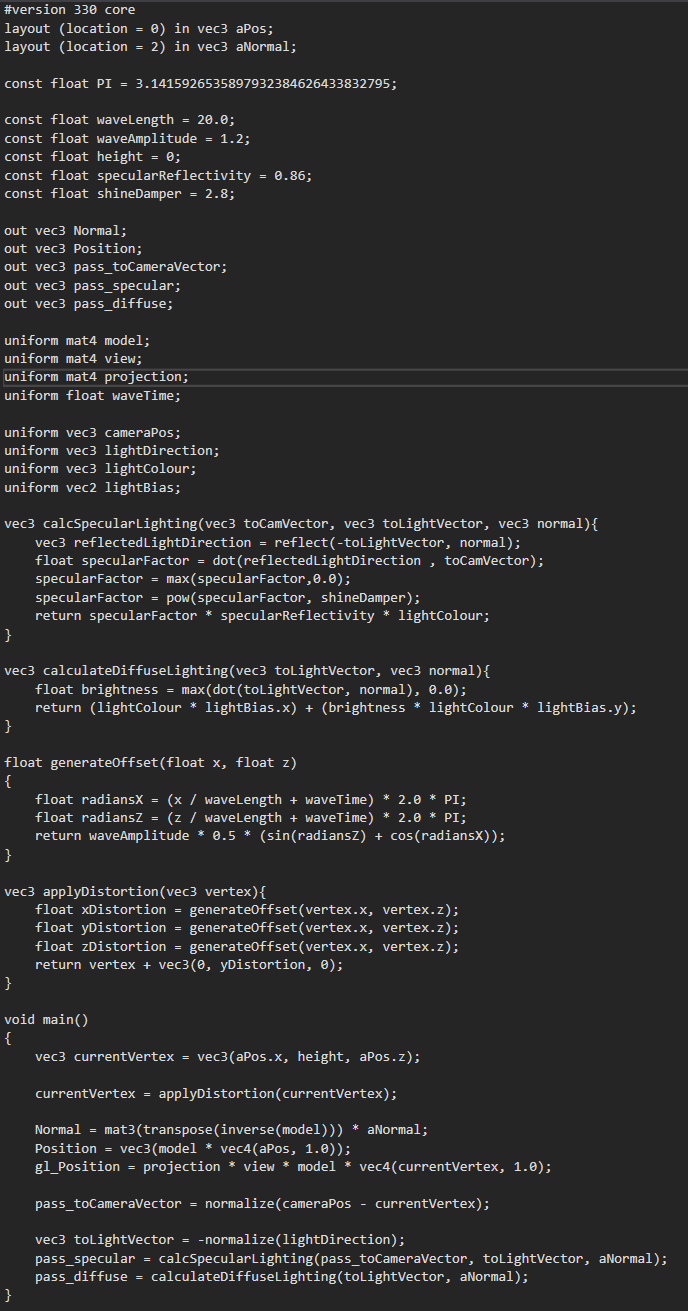
[Improvements 9](bookmark://_Toc8369716#_Toc8369716)

[References 10](bookmark://_Toc8369717#_Toc8369717)

# Introduction

The additional graphical technique that I chose to implement was a water effect. I decided to implement this effect since water has many properties that can be altered to produce interesting results. To implement the water I first loaded a flat plane .obj model created in Blender. This plane was made up of 128x128 vertices and included normals and uv coordinates. This model was loaded into the project using a function called initModel() in the Mesh.cpp class, which takes in an IndexedModel and generates the necessary vertex array objects and vertex buffer objects for the vertex positions, texture coordinates and normals. It also sets up a buffer containing the indices associated with the model to ensure no vertices are shared and save on memory. To this plane model I applied a shader program made up of a vertex and a fragment shader which was responsible for creating a water effect.

# Vertex Shader



This is the vertex shader that is applied to the water. First the version of GLSL is stated, in this case version 330 core. Two vertex attributes are then defined one for the position and one for the normal. These are set when the model is initialised. A float constant PI is defined and this is used to convert degrees to radians in certain methods within the shader. Constants to do with the wave distortion are also defined, one float for the wavelength and one float for the amplitude. The constant “height” is used to set all the vertices to the same height before distortion. “specularReflectivity” and “shineDamper” are used to alter the appearance of light of the surface of the water.

The vertex portion of the pipeline passes out a few variables to be used in the fragment shader. It passes out the normal and position of the vertex, the vector pointing from the vertex to the camera and the specular and diffuse calculations to be used in the lighting of the water.

The model view projection matrix is passed in as a uniform as well as the “waveTime” float which is used to add movement to the waves of the water over time. Uniforms necessary for the lighting calculation are also passed in, these being the position of the camera, the direction to the light source, the colour of the light and the light bias which is the effect the lighting has on the final colour.

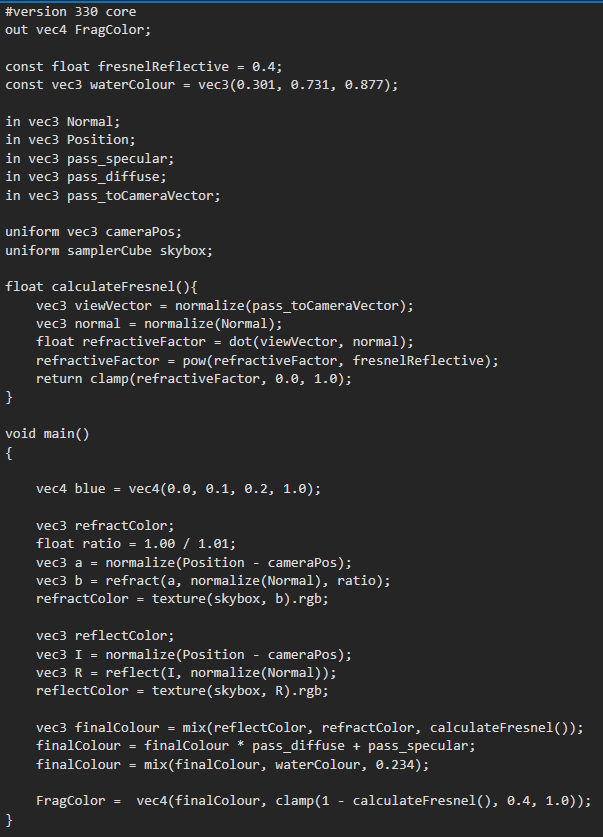
The first method in the vertex shader, calcSpecularLighting() return a three dimensional vector to be used in the final colour. First a 3D vector called reflectedLightDirection is defined, and this is the reflected direction of the incident vector, calculated using the normal. The “specularFactor” is then calculated, and this is the dot product of the reflected direction and the vector pointing towards the camera and this determines the angle between the camera and the reflected light. Using the GLSL method max ensures that this value never drops below 0. The specular factor can then be raised to the power of the previously defined shineDamper to increase its effect further. This specular factor is multiplied with the reflectivity and the light colour and returned.

The next method calculates the diffuse portion of the lighting. A float brightness is defined as the dot product between the vector pointing towards the light source and the normal, and is set to always be above 0. The light colour multiplied by the x component of the lightBias vector is added to this newly calculated brightness value multiplied by the light colour and the y component of the lightBias vector.

For the movement of the waves, two methods are used at the beginning of the vertex shader. generateOffset() takes in two floats and calculates two values in radians which are used in sin and cosine functions and multiplied to the wave amplitude. applyDistortion then uses the generateOffset method to return a 3D vector with the offset applied to each component of the vertex. In this case only the y component is actually set to be distorted so the waves only move up and down, instead of side to side, which looks smoother.

In the main method of the vertex shader, a 3D vector is defined to represent the position of the current vertex, with the y values being set to the constant height. Distortion is then applied to this current vertex. The output gl\_Position is then simply the position of the current vertex, newly distorted, multiplied by the model view projection matrix. The normal, position, specular lighting and diffuse lighting are also passed along to the fragment shader here.

# Fragment Shader



The fragment shader is responsible for outputting the final colour of each fragment of the water, and so after defining what GLSL version is being used, the 4D output vector FragColor is set. A constant float called fresnelReflective is defined here and this is used in the calculation of the Fresnel effect, and simply determines how reflective the surface of the water is when it is at its shiniest. A 3D vector describing the water colour is set here to a light blue colour, and this is applied at the end of the fragment shader to give the water a blue hue. The five 3D vectors from the vertex shader are shown. Two uniforms are taken in at the fragment shader, a 3D vector for the camera’s position as well as the texture of the skybox cubemap, which is used for reflection and refraction of the environment.

The first method in the fragment shader calculates the Fresnel effect. This is an effect that describes how water gets more or less transparent based on the angle of the camera to the surface of the water, with the water being at its most transparent when the camera is looking straight down the direction of the surface normal. The vector pointing towards the camera is sent in from the vertex shader, which is then normalized, since only the direction matters and not the length. A float “refractiveFactor” is then set to the dot product of the vector poiting to the camera and the normal, and this is the value that affects how transparent the water is. This value can be raised to the power of the previously defined “fresnelReflective” to increase the reflection of the surface. Finally the refractive factor is clamped between 0 and 1.

In the main method of the fragment shader the reflection and refraction colours of the water are calculated. For the refraction, the vector from the fragment position to the camera is calculated. This vector is then used in the GLSL function “refract” along with the normal and a refraction ratio to produce a 3D vector which points through the surface of the water. This vector is then used to sample the colour of the skybox cubemap texture, which is sent in as a uniform, and returns this colour value.

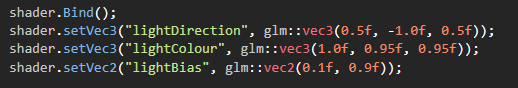
Reflection is done in almost the exact same way except using the GLSL function “reflect” which reflects off the surface of the fragment and samples a colour from the skybox.

These two colours for reflection and refraction are then mixed together using the GLSL function mix, which linearly interpolates between two values based on a third parameter between 0 and 1, which is set to the previously calculated Fresnel effect factor. This resulting colour then has lighting applied to it by multiplying it by the diffuse and specular lighting, as well as mixing it with the blue water colour.

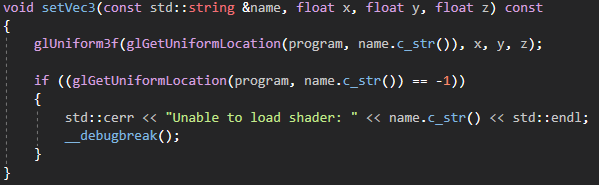
Finally the alpha value of the colour is set to 1 – the Fresnel factor to increase transparency.

# MainGame.cpp

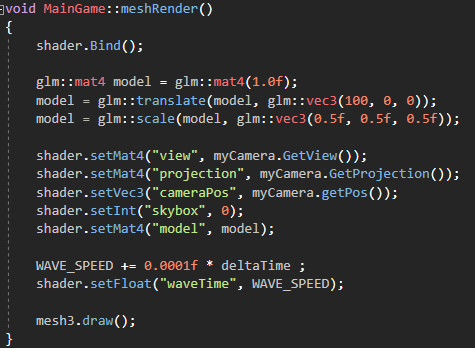
This is the class in which the rendering of the scene takes place, as well as the passing in of uniforms for the vertex and fragment shaders.



This block of code is run once at the beginning of the program since none of the values change throughout. The shader.set methods all work in a similar way, by getting the uniform location of the named uniform using glGetUniformLocation, and then using this value in glUniformxf to set the x values.



All of the other uniforms which are passed into the shader pipeline are done so every frame in a method called meshRender. This binds the shader to be used and then passes in the uniforms that change every frame, in this case the MVP matrix, the camera position and the “waveTime” uniform which causes the waves to move.



deltaTime is a float that keeps track of the time elapsed between each frame using SDL\_GetTicks() and this is multiplied to everything in the scene that moves so that movement is not dependent on framerate of different machines. This meshRender function is called in the draw loop, but not before setting the depth mask to false using glDepthMask(GL\_FALSE); This allows for transparency in the water by making the depth mask read only instead of updating it when drawing the water. It is set back to GL\_TRUE once the water has been rendered.

# Improvements

The reflection of the water is done only using environment mapping which means that other objects in the scene are not reflected in the water’s surface, only the skybox. This could be changed by rendering everything above the water to a texture using frame buffer objects and then using this texture on the surface of the water. Clipping planes would need to be used to cut off everything below the water. The same is true for refraction except reversed to show everything below the water. Projective texturing could be used to make sure the reflection and refraction textures move with the waves of the water.

Due to the nature and simplicity of a water mesh it could just be generated in code, although difficulty with correctly calculating the proper normals was encountered. This could allow for split mesh, where no vertices are shared and interesting effects like low-poly water could be achieved.

The lighting model used in the project could have also been improved by spending more time researching different methods of lighting, since at the moment it is still quite difficult to see the waves at certain camera angles.

An element of randomness could also be added to the waves to make them less uniform.

# References

Deer model, heart model, terrain model - <https://free3d.com/>

Low-poly water effect - <https://www.youtube.com/watch?v=5yhDb9dzJ58>

Rendering to textures - <https://www.youtube.com/watch?v=5bIpaXPiPIA&t=2140s>