IMAT 3907 Advanced Shaders Report

Introduction

The primary focus of this project was to improve my knowledge and widen my skillset in multiple areas, I wanted to lean more details about the rendering pipeline and learn how to implement advanced shading techniques. To achieve this goal, I would create a landscape scene using C++ and OpenGL.

For my landscape I wanted to create a scene leaning towards the cartoony side, as opposed to a hyper real look, I did this as it would be easier to achieve something that looks good with a cartoonish style, and I would not have to acquire lots of high-quality assets (Textures, Models, etc.). Furthermore, a cartoonish style was chosen as it would be less resource intensive on the system running the program.

For my project to be successful it will need to showcase multiple advanced shading techniques and rendering effects at the same time, such as geometry and tessellations shaders, framebuffers, post processing, and shadows.

Resources

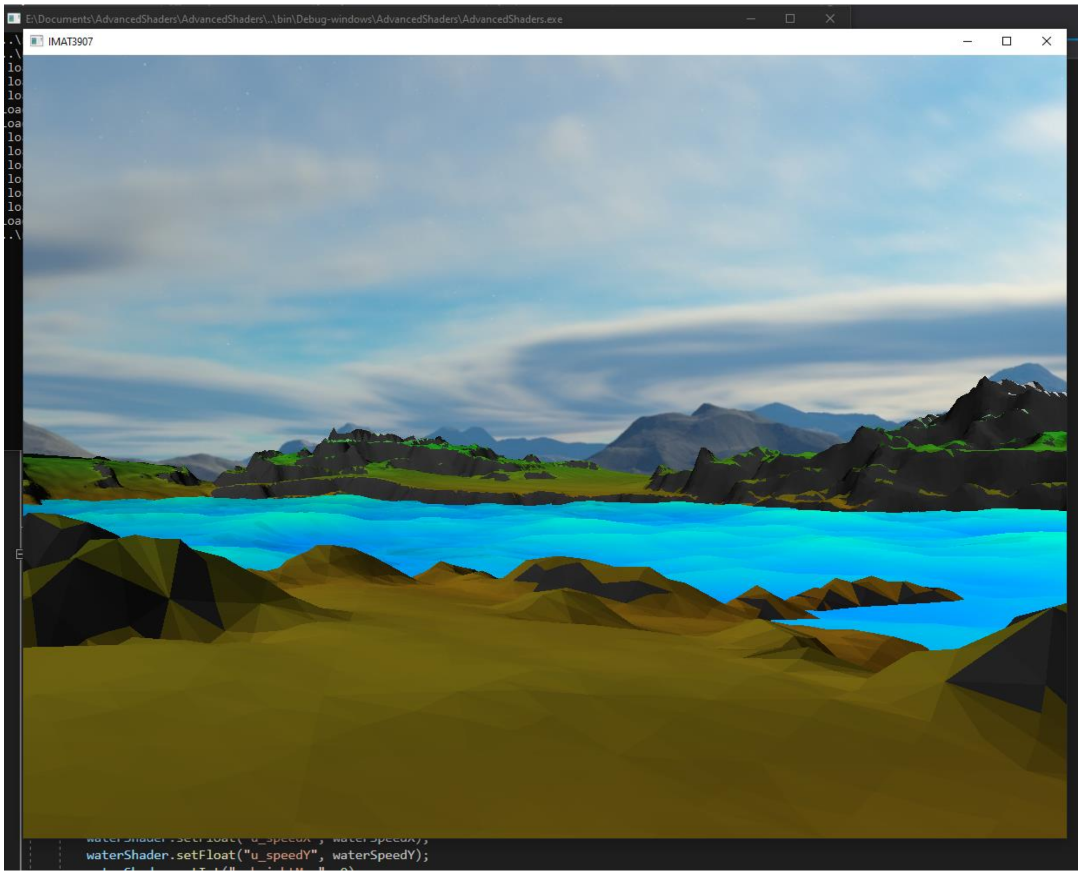
The primary learning resources for this project where the lectures and learning materials provided on the university course and ‘LearnOpenGL.com’ by ‘Joey de Vries’. Both resources where particularly helpful as they provide both theory and practical information, along with code snippets to help with understanding advanced techniques.

I also made use of the project provide to us in the leaning materials as a jumping off point, as starting from scratch would drastically lengthen the time needed to complete the project. This project includes all the libraries needed to access OpenGL functions and load in 3D models, it also includes a couple of classes that abstract functionality to make repeating operations easier.

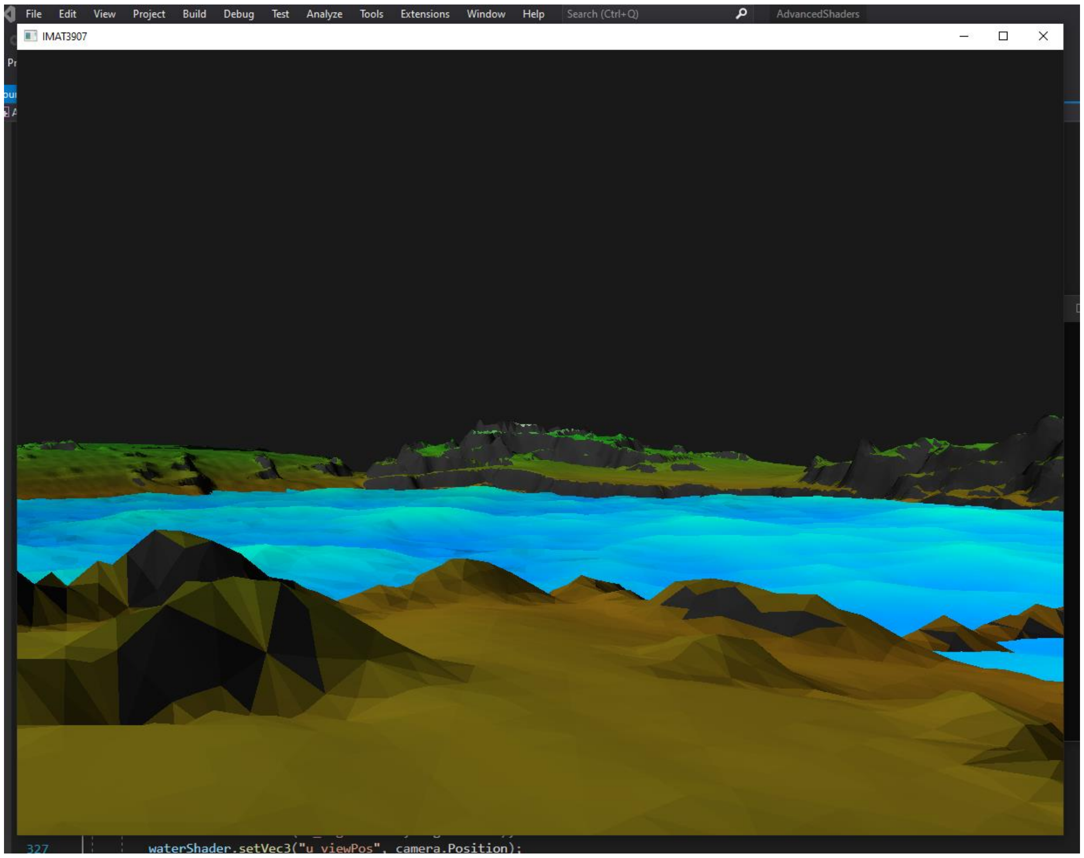
Skybox

A skybox is a method of creating a background for a level that encompasses the cameras view to make it appear that the level is much bigger than it is. They can also be used as a reflection map, to give diffuse lighting more detail. A skybox is made up of six seamless textures that when aligned will form a cube that surrounds the camera. I introduced a skybox to move my scene a more realistic look and to make the scene feel larger.

My implementation of a skybox is very simple, I first load in all the resources I need which are a cube model and 6 textures. The cube was modelled in ‘Blender’ and exported as a wavefont (.obj) file, and the textures where pulled from ‘LearnOpenGL.com’. The 6 textures are then bound to a special OpenGL texture object called a cubemap. A cubemap allows us to use up one texture unit for 6 textures and they are laid out in a way that allows us to easily pull data using texture coordinates. The cube is placed on top of the camera and drawn from the inside using a simple shader. For this shader we don’t want to consider the translation of the camera, as we want the cube to always be on top of the camera. To do this we convert a 4x4 matrix to a 3x3 matrix, and then back to a 4x4, this removes the translation part but keeps rotation. This gives us the desired effect, making the scene feel much larger and gives a sense of scale.


E:\Documents\AdvancedShaders\AdvancedShaders\..\bin\Debug-windows\AdvancedShaders\AdvancedShaders.exe
IMAT3907

```
waterShader.setFloat("u_speedY", waterSpeedY);
```


File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
AdvancedShaders
IMAT3907

```
waterShader.setVec3("u_viewPos", camera.Position);
```

Tessellation

A tessellation shader is a program that can increase the level of detail in a mesh, this functionality is controlled by two programmable programs and one fixed function program. Tessellation involves subdividing a patch into smaller primitives, and then calculating the new vertex data for each of the vertices generated.

The first stage of the tessellation pipeline is the tessellation control shader (TCS), this tells the renderer how many primitives to generate from a patch, and then feeds this data to the primitive generator, any output values from the TCS are directly fed to the tessellation evaluation shader.

The next stage in the pipeline is the primitive generator, this is the unprogrammable part of pipeline, this is where patches containing vertex data are subdivided into smaller primitives, based on the values passed by the TCS.

The final stage is the tessellation evaluation shader (TES), it takes the results from primitive generator and the outputs from the TCS and calculates the per vertex data for the new vertices. The new values are calculated using interpolation.

I implemented a tessellation shader for the rendering of the terrain and the water in my scene, they both used a fixed tessellation level, this was done because when a distance dependent tessellation level was used, gaps appeared between the levels, reducing realism.

Terrain

In order to create a terrain, a flat subdivided plane with offset vertices can be used, these vertices can be offset in the 3d model, on the CPU, or on the GPU during a shader program. I chose to offset the vertices of the plane on the GPU using a displacement map. The plane was generated using the terrain class, which takes a cell size and number of cells to generate a flat subdivided plane with those properties.

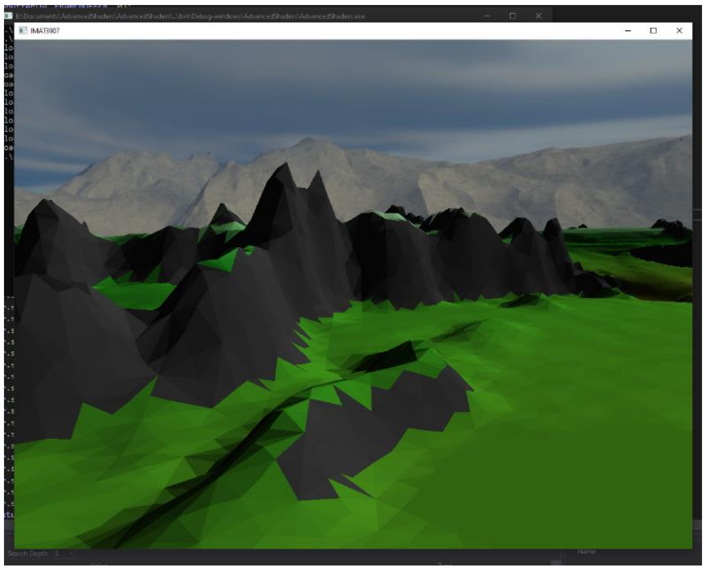
A displacement map is a greyscale texture which represents the offset of the vertices on the model, they are also sometimes called a height map. Unlike a normal map they change the position of vertices and work better with high resolution meshes. The red channel of the texture can be sampled in a shader, using a vertexes texture coordinates, this returns a value between 0 and 1. To create the terrain this value is retargeted to be between –1 and 1, this is to keep the terrain centred around is origin, and then multiplied by a scaler depending on how tall the terrain needs to be. When this value is added to the Y component of a vertex and done for all the vertices in the plane, it will deform the plane and makes it look like a terrain.

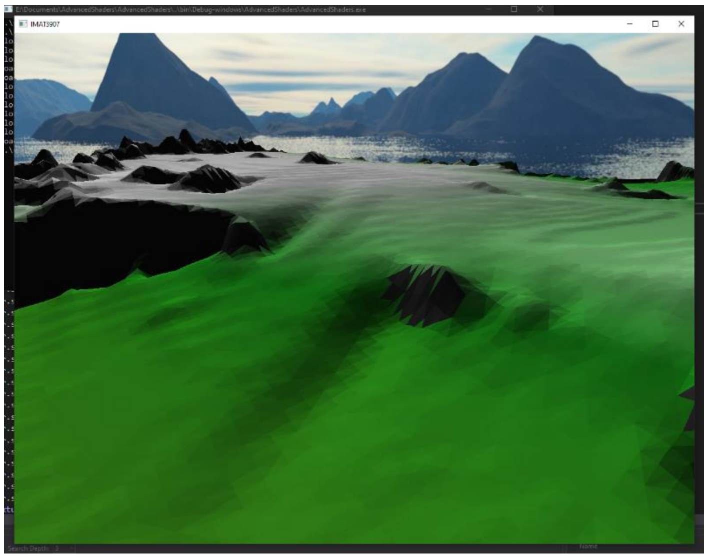
To increase the realism of the terrain a tessellation shader can be used to increase the number of vertexes, but for each new vertex we must calculate new normals and texture coordinates. The new normals can be calculated using the central difference method, which samples the height map around the new vertex to calculate the normal. Other properties such as the texture coordinates and the fragment positions can be calculated by interpolating between the 3 vertices of the original triangle/ primitive, this gives us a new value which is correct for the vertex.

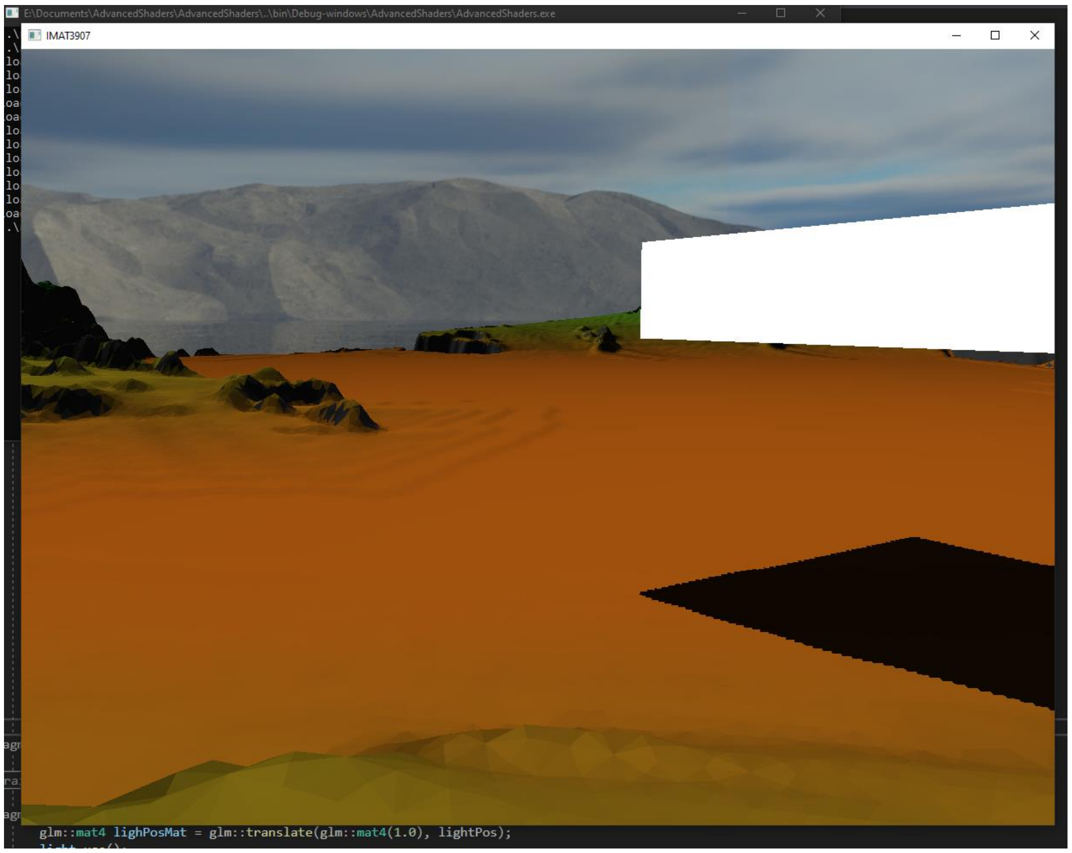
For the colour and lighting of the terrain I use a simple diffuse shader, I did not include specular lighting as I feel it didn’t match the visual style I was going for. For the colour I would pass the value sampled from the displacement map into the fragment shader and depending on the value change the colour. I chose a yellow/ sandy colour for low values, a bright green for middle values and white for high values, the colours were also interpolated to give smooth transitions between colours.

Another technique I used was calculating the dot product of the normal for the vertex and an up vector, if the dot product was high enough, I would set the colour to a grey colour to represent rock. Furthermore, in the geometry shader I would average the normal vectors of all the triangles in the plane to give a flat shaded look.

All together these techniques create an interesting, realistic looking terrain which matches the visual style I was going for.






E:\Documents\AdvancedShaders\AdvancedShaders\..\bin\Debug-windows\AdvancedShaders\AdvancedShaders.exe
IMAT3907

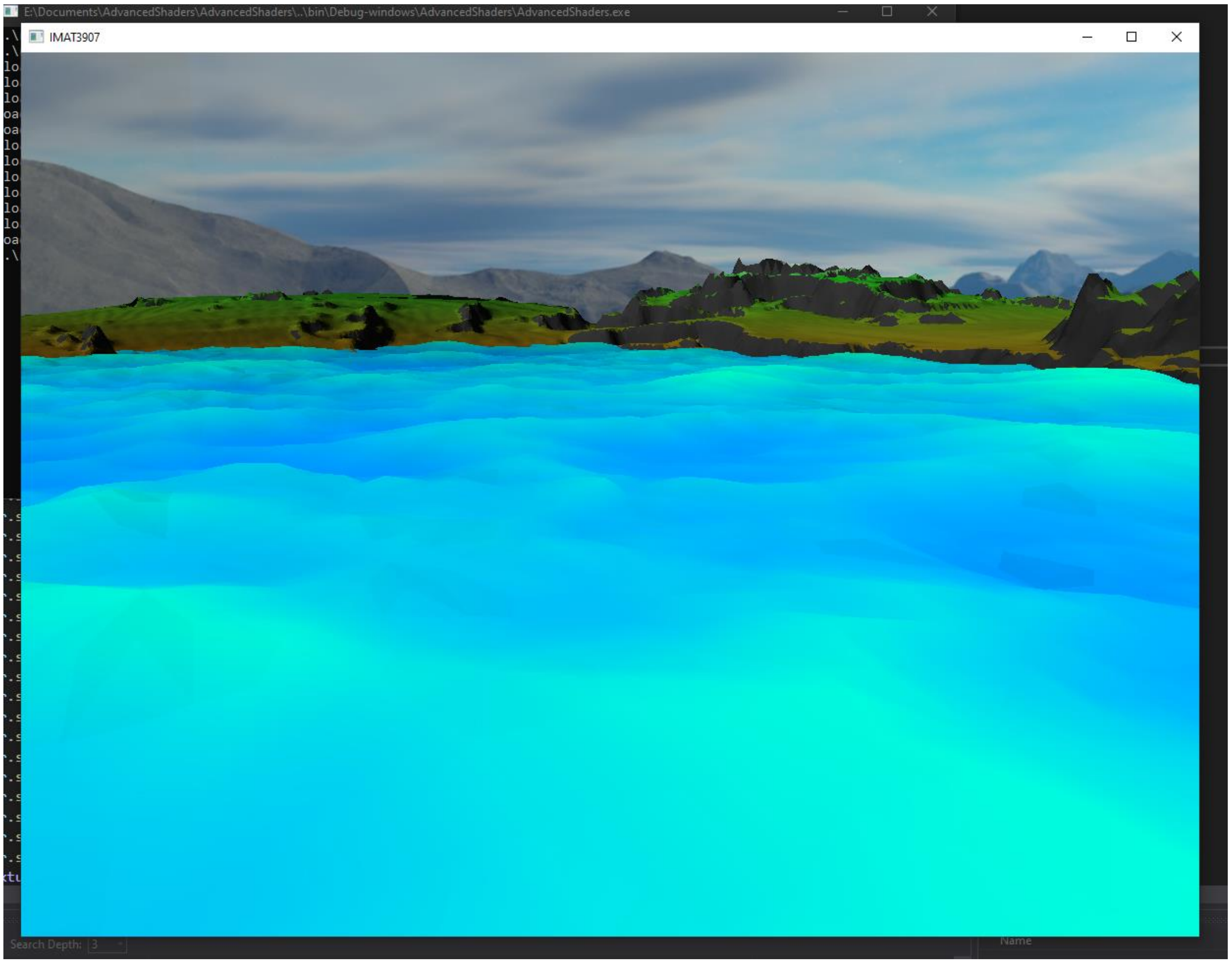
```
glm::mat4 lighPosMat = glm::translate(glm::mat4(1.0), lightPos);
```

Water

The water in the scene is very similar to the terrain, a flat subdivide terrain with offset vertices, I again offset these vertices on the GPU. However, instead of generating the terrain with a class I load it in with a model loader.

The water is offset using a displacement map, using the same method as the terrain, however I also offset the texture coordinates of each vertex, I do this using the 'glfwGetTime' function and a uniform, when this is done over a period of time it makes it look like the water is moving, this creates a very realistic effect and makes the scene dynamic rather than static.

To colour the water, I use a similar method to the terrain, depending on the height from the displacement map I interpolate between colours, a lighter greenish colour for the top of waves and a dark blue for the bottom. Like the terrain, I also average the normals of each triangle in the geometry shader to give the water a flat shaded look, keeping in line with the visual style I was going for.


E:\Documents\AdvancedShaders\AdvancedShaders\..\bin\Debug-windows\AdvancedShaders\AdvancedShaders.exe
IMAT3907
Search Depth: 3
Name

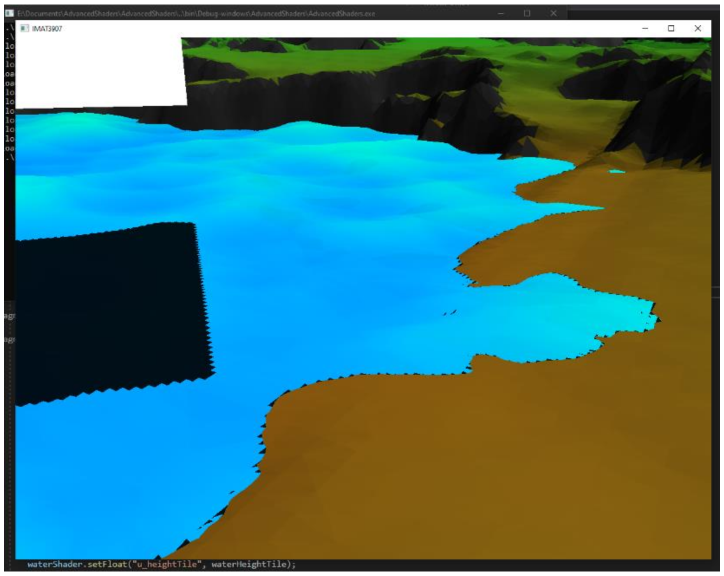
Shadow Mapping

Shadow mapping is a technique which aims to add realistic shadows to geometry, it tries to emulate occlusion of light by other objects, adding depth to the scene and making easier for the viewer to judge distances between objects.

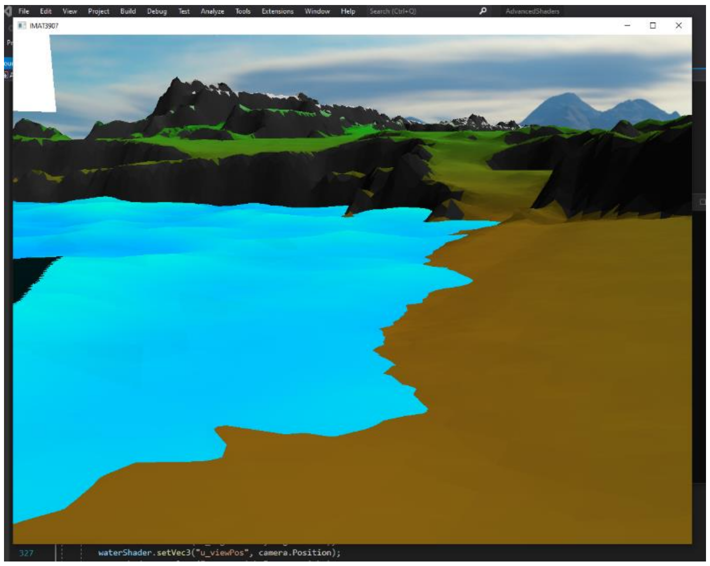
To achieve this we need to determine if a fragment is in shadow or not, we can do this using depth testing, we can render a depth map from the primary light sources position in the scene looking at the origin, then we can sample this map when we render objects to determine if it is in shadow or not.

In order to render the depth map we need to use a framebuffer to store the output, and a simple shader that renderers between black and white depending on the distance from the light source, as we are emulating the sun we can render this buffer in a orthographic view to emulate parallel rays of light. The size of the framebuffer determines the quality of the shadows, and the view matrix will determine how much of the scene will be captured in the buffer. A more complicated shader was used for objects with tessellation and movement (e.g. Terrain & Water).

When we render the colour pass the only difference is in the fragment shader, we transform the position of the fragment into the lights coordinates space, and in this space the z coordinate of the fragment is its depth value, we can then compare this to the depth buffer value and if the depth is higher it is in shadow. A shadow bias is also used to remove incorrect rendering of shadows.



```
waterShader.setFloat("u_heightTile", waterHeightTile);
```

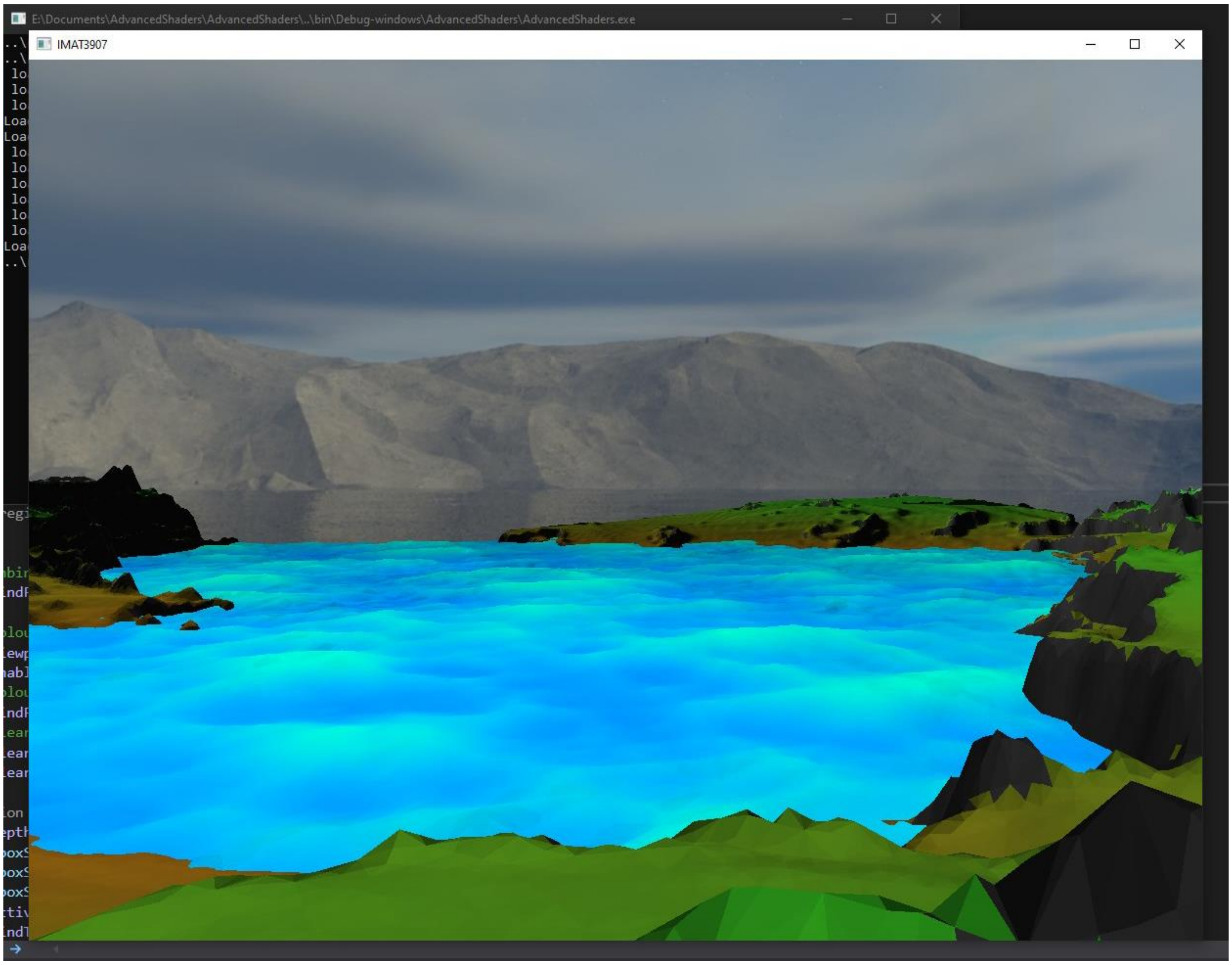

File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
IMAT3907

```
waterShader.setVec3("u_viewPos", camera.Position);
```

Conclusion

Overall, I'm especially happy with how the project went, I've learned a lot about advanced shader techniques, the project went along smoothly all the way, and I think the work I've created is an outstanding reflection of my practical skills and knowledge of C++. I am also especially happy with my water shader because I believe it pushes my work to next level adding a lot of realism.

However, if I were to continue working on the project, I would have like to add more variety to the scene with props and vegetation, and apply advanced shaders to those objects, to create a more realistic scene. Furthermore, I would have also like to have wrapped the code I have written into classes to make it easier to use, but that probably learns more into a general 3D rendering with a focus on coding skills rather than shader design.


E:\Documents\AdvancedShaders\AdvancedShaders\..\bin\Debug-windows\AdvancedShaders\AdvancedShaders.exe
IMAT3907