IMAT 3907 Advanced Shader Programming

Coursework Report

By Radoslaw Zajdel

P17215040

Contents

[Abstract 2](#_Toc39489282)

[Introduction 2](#_Toc39489283)

[Instruction 2](#_Toc39489284)

[Terrain 2](#_Toc39489285)

[Grid 3](#_Toc39489286)

[Heights 3](#_Toc39489287)

[Normals 3](#_Toc39489288)

[Colour/Texture 3](#_Toc39489289)

[Lighting 4](#_Toc39489290)

[Fog 4](#_Toc39489291)

[Level Of Detail 5](#_Toc39489292)

[TessControlShader 5](#_Toc39489293)

[TessEvaluationShader 6](#_Toc39489294)

[Shadows 6](#_Toc39489295)

[Light set-up 7](#_Toc39489296)

[Framebuffer 8](#_Toc39489297)

[Fragment Shader 8](#_Toc39489298)

[Depth Buffer 8](#_Toc39489299)

[Conclusion 8](#_Toc39489300)

[Appendences 8](#_Toc39489301)

# Abstract

This report is to demonstrate my understanding of what was being taught and explained within the module content, the assessment task, the code it consists of and how it all works together. It will particularly cover techniques which were required to put the code up together and achieve final results.

# Introduction

To understand how the program works, it is worth to go back to basics and remind what a shader program is. The definition says that this is a program which runs in a graphic pipeline and tells the computer how to render each pixel. It is also worth to mention that by utilising the power of the GPU shader applications running on this unit are much more efficient and fluent than those using only CPU, which is usually much less powerful. The name of a shader came from the ultimate goal of this type of program, which is to control lights and shadings. [1] This is what was finally achieved in the coursework. Within this assessment, there are many techniques implemented which will all be covered later on.

# Instruction

To demonstrate all functionalities of the program there are many keys which turn different effects on and off:

**Q** – turn shadows on/off

**F** – turn fog on/off

**B** – turn the blinn-phong shading off/on

**E** – turn the grid on/off

**P** – print camera position

**WASD** – move around the scene

**Left Shift (hold while moving)** – slowly increase the movement speed

**Left Ctrl (hold while moving)** – slowly decrease the movement speed

**Mouse Scroll In –** zoom in (e.g. to see tessellation)

**Mouse Scroll Out –** zoom out

It is strongly suggested to turn first four buttons on to see the true potential of the presented program. Then while operating with the mouse and zoom experiment with turning grid on and off to see how tessellation works.

# Terrain

At the first stage of the development of this application, the terrain was generated with the use of heightmap. It gave a very nice effect and allowed to use different source files but did not have much potential for improvement and the colours looked flat and did not vary nicely. Within the further development stages, the procedural terrain generating technique was implemented. It is the one used by the program now and it utilizes perlin noise function and makes all calculations on GPU to maintain potential.

## Grid

The initial grid is created with the use of terrain constructor included within the Terrain class. It specifies the number of grids in width, height and grid size. Increasing the first two values lead to GPU lags. Therefore values given in the constructor in the final iteration of the program have numbers as they were at the beginning and are well balanced providing nice visual effect and not sweating the GPU at the same time.

## Heights

As mentioned before, all heights are generated procedurally. All calculations are done on GPU for better performance and are in the tessEvaluationShader.tes file. The whole process utilizes various functions to return the sum of noise functions. The great advantage of perlin noise is that it can easily be used for infinite terrain generation and gives a nice variety between generated terrains.

## Normals

Normals in shader programming are incredibly useful technique which helps simulate real look of rendered objects and scenes. In this project every object consists of one of the most popular primitive objects in programming – triangles. By using normals or applying normal maps, objects look much more realistic and by a low cost bring more life to the scene which is usually one of the most valuable assets.

Normals maths are also done in tessEvaluationShader.tes. As the program makes use of perlin noise, apart from usual normalize function, those calculations need cycleOctaves function to work. Octaves are essential for correct effect as they take multiple noise functions with different parameters and adds them together.

## Colour/Texture

There were two options to add some life to the project. One was colouring the terrain dependant on height and the other was to use textures. This project utilizes the first option and multiple colours including dark blue, blue, yellow, green, brown, grey and light blue for fog. Each colour is assigned to a different height. Colours next to each other mix with each other to give a sensation of a fluent colour switch. The trickiest part of implementing the height dependant colours was to tune each smoothstep value to receive nice, fluent colour transitions. In lowest parts of the scene there is dark blue colour simulating the depth of the water while upper parts, close to the land are much brighter. Yellow colour simulates sand on the beach and in lowest parts of the ground. Climbing higher, biggest part of the ground is coloured with lively, intense green. After smoothly switching to pale green, tops of the mountains are grey. This simulates rocks and snow.

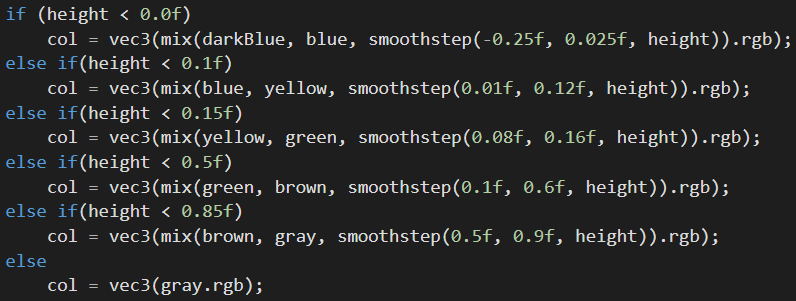


Fig. 1. Height dependant colours

## Lighting

There are two lightning options available on the scene. The default one is the usual phong reflection. After pressing B, it is possible to switch to much nicer are more realistic blinn-phong. Lightning calculations are performed in plainFrag.fs file. Switching between different lightning profiles is possible thanks to performing essential calculations separately and passing the lightning state by a uniform straight to the fragment shader file.

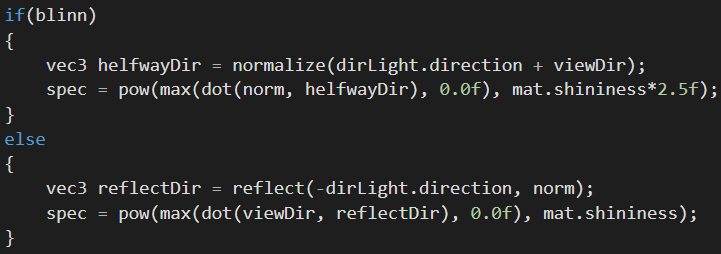


Fig. 2. Lightning calculations

## Fog

Creating and implementing fog was one of the easiest and most satisfying tasks to complete. This mechanic is very simple to understand and all it does is fading objects into the colour of the background usually called sky. This effect is strictly dependant on the distance of the terrain from the camera. The further the terrain is, the stronger the fade is. As for every other mechanic, this one also requires essential calculations to be performed correctly. In the code, they are placed in tessEvaluationShader.tes file and the visibility value is later passed to fragment shader.

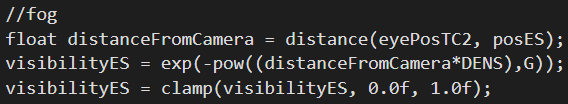


Fig. 3. Fog calculations

# Level Of Detail

Level Of Detail (LOD) is often called tessellation. This is an incredibly powerful technique which helps with utilizing the performance of GPU in the best possible way. Once implemented and tuned correctly, it makes sure that the number of vertices of the object decreases as they are further away from the camera. If adjusted correctly it will be impossible to tell that this is working but the number of displayed elements will be hugely decreased, therefore saving a lot of GPU memory. Within this there are three stages, from which two are programmable: Tessellation Control Shader, Primitive Generator and Tessellation Evaluation Shader (first and third are programmable). [2, 3]

## TessControlShader

As the first programmable part of the tessellation process, Tessellation Control Shader file uses GetTessLevel function to return the varying tessellation level parameter. This is later used to calculate the tessellation level based on the distances between the camera and the vertex. There is not much more happening in this file as everything else there is passed further through the shader pipeline.

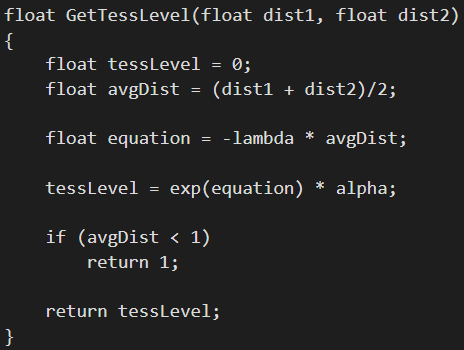


Fig. 4. GetTessLevel function

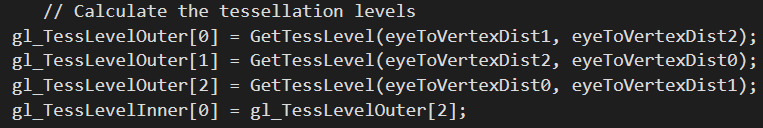


Fig. 5. Calculation tessellation levels

## TessEvaluationShader

This file is where all the magic happens as all transforms are performed here. There are two very important functions implemented called interpolate2D and interpolate3D for linear interpolation. Within this assessment, only the second function is used as everything on the scene is 3D. As mentioned above, there are some other functions responsible for the terrain generation. Those functions are called: hash, snoise and cycleOctaves. They are responsible for essential calculations to create the terrain without using the height map and thanks to their construction it is easy to adjust values as needed. There are four key values that have the biggest influence on how the terrain looks: total, maxAmplitude, amplitude and frequency.

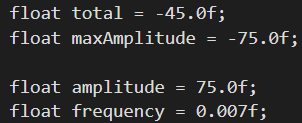


Fig. 6. Tuneable terrain values

# Shadows

Shadows were one of the hardest assets to implement. The most popular way of implementing them is by creating a shadow map to the texture. They require shadow specific calculations and a calcShadow function as well as many adjustments to the Source.cpp file. First of all, there was a need to implement a light with a direction where it would point and complete calculations for lightView and lightSpaceMatrix. Then, the results needed to be passed to the shader pipeline via uniform. After that, by using the calculated matrix and function implemented to plainFrag.fs file, the shadow could be created and applied to the texture. Like colours, shadows can also be dependent on the height. If lines 94-97 were uncommented, shadows would not appear below water level.

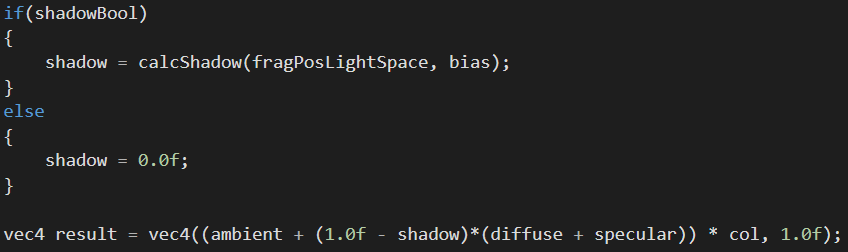


Fig. 7. Shadow final calculations

The issue of not having shadows on edges of the terrain was fixing by adjusting the ortho size value from 250 to 350.

## Light set-up

There are many different kinds of light and ways of implementing them in shader programming. What they usually have in common is that for best quality they require a combination of three types of lighting, ambient, diffuse and specular. First is even around the whole scene, second simulates the directional impact of the light and the object and the last adds realism by simulating the shiny dot that appears in real life on smooth surfaces. [4]

The light in the project and its parameters are set in the Source.cpp file. It includes a light position, direction where the light points. Those two parameters are processed into the light view and later into the light space matrix.

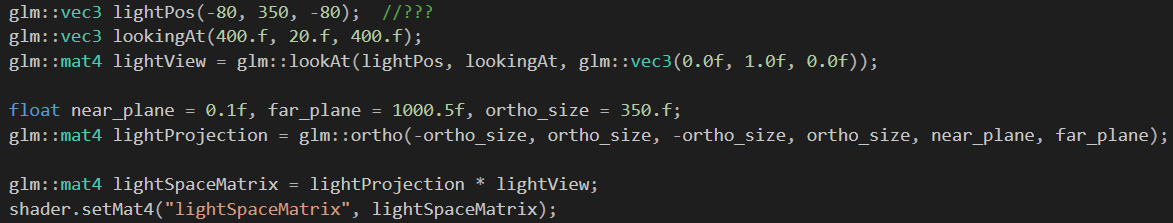


Fig. 8. Light set up

After setting light up, there is some processing done in plainFrag.fs file to calculate how the light performs and is separated into two different models: phong and blinn-phong.

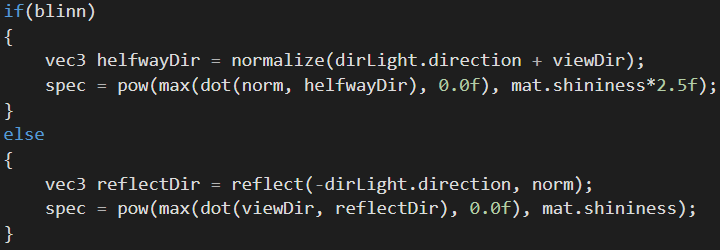


Fig. 9. Light separation

## Framebuffer

Another important part of creating shadows is implementing a framebuffer. Application of this technique allows to render the scene into the designated framebuffer first, do post-processing with it and apply effects like shadows, night vision or underwater sensation and then render it on the screen. It is a very powerful way of achieving stunning effects which are very useful in the gaming and movie industries.

## Fragment Shader

Most functionalities of fragment shader called within the assessment plainFrag.fs has been described before. To sum up, this step in the shadow pipeline is responsible for putting everything up together and outputting coloured pixels on the screen.

## Depth Buffer

As for depth buffer, there is one function designated for calculating depth called setFBOdepth. Without it, if shadows turned on, the terrain would just be darker everywhere, which is not a proper shadow. setFBOdepth is responsible for creating shadows into the framebuffer object and binding them. [5, 6]

# Conclusion

The project provides great looking final results. The terrain is colourful and bright. It has many mechanics and advanced techniques programmed and have great potential for development. Although the project seems to be completed, there could be more implemented. There are many other advanced techniques left that were not implemented such as antialiasing or adding water.

# Appendences

1. Omar Shehata, 15 Apr 2015, A Beginner’s Guide to Coding Graphics Shaders, gamedevelopment.tutsplus.com, https://gamedevelopment.tutsplus.com/tutorials/a-beginners-guide-to-coding-graphics-shaders--cms-23313
2. Fahy, Conor, 2020, Week 3 - Introduction to Tessellation Shaders, from [IMAT3907\_1920\_502 Advanced Shader Programming](https://vle.dmu.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_547930_1&content_id=_4522523_1), De Montfort University, 28th January 2020, available from Blackboard
3. Fahy, Conor, 2020, Week 4 - Tessellation Shaders, from [IMAT3907\_1920\_502 Advanced Shader Programming](https://vle.dmu.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_547930_1&content_id=_4522523_1), De Montfort University, 04th February 2020, available from Blackboard
4. Learnopengl.com, Basic Lighting, learnopengl.com, https://learnopengl.com/Lighting/Basic-Lighting
5. Fahy, Conor, 2020, Week 9 - Framebuffers, from [IMAT3907\_1920\_502 Advanced Shader Programming](https://vle.dmu.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_547930_1&content_id=_4522523_1), De Montfort University, available from Blackboard
6. Fahy, Conor, 2020, Week 10 - Shadow Mapping, from [IMAT3907\_1920\_502 Advanced Shader Programming](https://vle.dmu.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_547930_1&content_id=_4522523_1), De Montfort University, available from Blackboard