322Com Submission 2

Procedural Generation

Contents

[Introduction 2](#_Toc25919452)

[Project Progression 3](#_Toc25919453)

[Diamond-Square algorithm 3](#_Toc25919454)

[Diamond 3](#_Toc25919455)

[Square 4](#_Toc25919456)

[Multi-Texturing 4](#_Toc25919457)

[Secondary Object and animation 5](#_Toc25919458)

[Fractal Trees 7](#_Toc25919459)

[Placement 7](#_Toc25919460)

[Skybox 8](#_Toc25919461)

[Optimisation 10](#_Toc25919462)

[3D Camera Controls 11](#_Toc25919463)

[Reflective Review 12](#_Toc25919464)

[Video Link 12](#_Toc25919465)

[References 13](#_Toc25919466)

# Introduction

The aim of the assignment is to implement a real-time procedural generator for gaming environments based on C++ and modern 3D graphic API such as modern OpenGL.

To complete this I will need to write, from scratch, a real-time procedural generator who’s focus is computer games, such as flight simulators. The main implementation needs to consist of a large terrain including 2 other procedurally generated items.

# Project Progression

## Diamond-Square algorithm

The diamond Square algorithm is used as an average between specific points on a plane to randomise the heightmap whilst keeping the terrain smooth. This terrain will be used to make up the bulk of my scene.

### Diamond

The Diamond part of the Diamond-Square algorithm is used as the start point. It takes the original four corners of the plane, averages out their height and then sets the centre point height value to that average height.

void Generate Terrain::diamond Step(int x, int z)

{

std::random device rd.;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<float> dis(-this->maxRange, this->maxRange);

float avg = 0.0f;

avg = terrain[x - stepSize / 2][z - stepSize / 2] + terrain[x + stepSize / 2][z + stepSize / 2] + terrain[x + stepSize / 2][z - stepSize / 2] + terrain[x - stepSize / 2][z + stepSize / 2];

terrain[x][z] = (avg / 4.0f) + dis(gen) \* randMax;

}

Figure - Diamond

### Square

The Square part of the Diamond-Square algorithm is then used as a follow up to Diamond. This section of the algorithm will take the original four corners and the centre point created in Diamond, then calculate the average height and then sets the mid-point of that edge on the plane to the average height.

float GenerateTerrain::squareStep(int midX, int midZ)

{

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<float> dis(-this->maxRange, this->maxRange);

//let avg = average of four corners of diamond

float avg = 0.f;

int count = 0;

if (midZ + stepSize / 2 >= 0 && midZ + stepSize / 2 < MAP\_SIZE) {

avg += terrain[midX][midZ + stepSize / 2];

count++;

}if (midZ - stepSize / 2 >= 0 && midZ - stepSize / 2 < MAP\_SIZE) {

avg += terrain[midX][midZ - stepSize / 2];

count++;

}if (midX + stepSize / 2 >= 0 && midX + stepSize / 2 < MAP\_SIZE) {

avg += terrain[midX + stepSize / 2][midZ];

count++;

}if (midX - stepSize / 2 >= 0 && midX - stepSize / 2 < MAP\_SIZE) {

avg += terrain[midX - stepSize / 2][midZ];

count++;

}

float heightMap = (avg / (float)count) + dis(gen) \* randMax;

return heightMap;

}

Figure - Square

## Multi-Texturing

Once the height of the terrain at different points had been calculated I was then able to use those heights to texture the terrain at different points with different textures. To accomplish this, I created my own Texture class and then created a new VertexShader and new FragmentShader that will deal solely with the height and texturing of the terrain. I know of an alternative to creating multiple textures onto one terrain, however after using multiple objects in one shader I decided I wanted to give this new method a try so that my own knowledge in OpenGL was expanded. After completing the module, I can say that one shader to handle all the calculations is a lot less complex, but on the flipside it is a lot harder to follow all of the calculations compared to having specific shaders for a specific object.

vec4 texColor;

if(yValue < downRange) { texColor = sandTexColor; }

if(yValue > downRange - blendRange && yValue < downRange + blendRange) { texColor = mix(sandTexColor, grassTexColor, blendFactor); }

if(yValue > downRange && yValue < upRange) { texColor = grassTexColor; }

if(yValue > upRange - blendRange && yValue < upRange + blendRange) { texColor = mix(grassTexColor, rockTexColor, blendFactor); }

if(yValue > upRange && yValue < maxRange) { texColor = rockTexColor; }

if(yValue > maxRange - blendRange && yValue < maxRange + blendRange) { texColor = mix(rockTexColor, snowTexColor, blendFactor); }

if(yValue > maxRange) { texColor = snowTexColor; }

Figure - Multitexture

## Secondary Object and animation

To meet the requirements of the assignment brief I need to create at least three different procedurally generated objects. After completing my base terrain, I was left to decide between Trees, Clouds, or an ocean. I decided that my secondary terrain would be an ocean.

To complete this I object-oriented my terrain code that was shown above, I then made an instance of this class to create my main terrain and another instance of the class to create my ocean. However, this method alone was not what was asked of me, to separate the two methods out I had to change how the ocean looked and acted in the scene.

To accomplish this, I went back to my original VertexShader, I then added in Sine and Cosine calculations so that the plane is animated through waves in the final scene.

float amplitude = 0.3f;

vec4 pos;

pos = terrainCoords;

float frequency1 = 1.;

pos.y = sin(pos.x \* frequency1);

float t = 0.01\*(-frequency\*130.0);

pos.y += sin(pos.x\*frequency1\*2.1 + t)\*4.124;

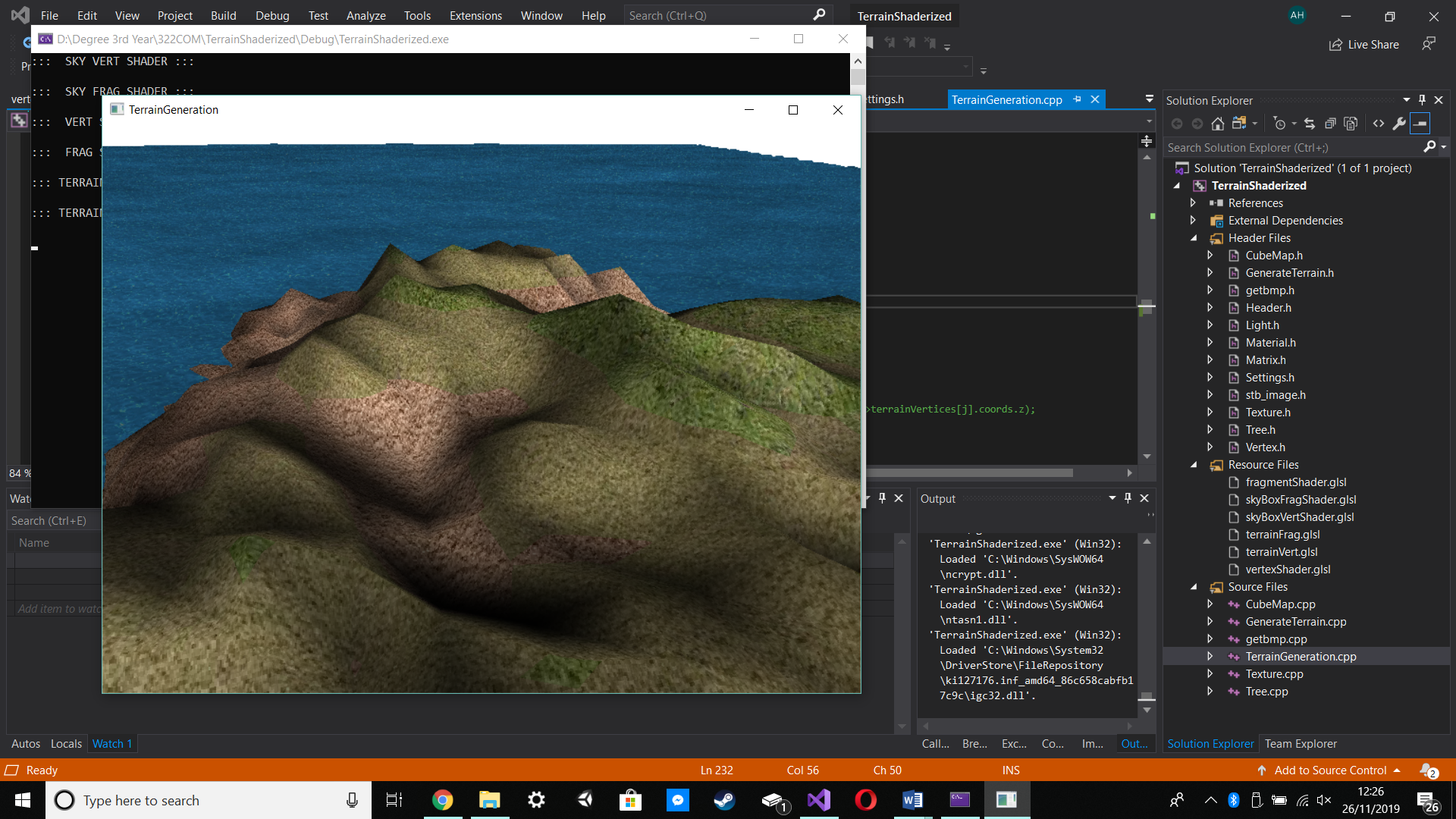
pos.y += sin(pos.x\*frequency1\*1.72 + t\*1.121)\*3.184;

pos.y += sin(pos.x\*frequency1\*2.221 + t\*0.437)\*4.040;

pos.y += sin(pos.x\*frequency1\*3.1122+ t\*4.269)\*1.628;

pos.y \*= amplitude\*0.06;

Figure - Animation



Figure

Figure 5 Shows the combined results of all the steps that I have taken above, except for the animation.

## Fractal Trees

After I had completed the Ocean and Land terrain generation with the animation and the multi-texturing I began looking into creating trees to populate the scene with. To create a fractal tree, I needed to create the original branch, which would then branch out recursively to create more branches from the original until the tree has been created. From there the leaves of the tree can then be drawn through leaf indices which is done through more recursive fractals.

### Placement

To instantiate the class, I used to create all my fractal trees I coded it so that the tree required a starting vec3, this meant that I could then code each tree with a percentage chance of spawning along with a specific location in the world if that percentage was met. To prevent the trees from consuming the entire terrain I only allowed them a 10% of a successful spawn rate, and then I only allowed them to spawn between specific heights on the terrain.

for (int i = 0; i < terrains.size(); i++)

{

terrains[i]->setupMesh();

for (int j = 0; j < MAP\_SIZE \* MAP\_SIZE; j++)

{

int randNum = rand() % 10 + 1;

if (randNum > 9)

{

if (terrains[i]->terrainVertices[j].coords.y > 4 && terrains[i]->terrainVertices[j].coords.y < 20)

{

//cout << trees.size() << endl;

Tree\* temp = new Tree(terrains[i]->terrainVertices[j].coords.x, terrains[i]->terrainVertices[j].coords.y, terrains[i]->terrainVertices[j].coords.z);

trees.push\_back(temp);

}

}

}

}

Figure - Tree Placement

Figure 6 - Tree Placement Shows the code I used to instantiate my tree class. The code will run through the entire terrain size drawing in the height of each point. Once the height of the point has been drawn the code will then work out a random value between 1 and 10, if the value is 10 then the tree has successfully been spawned but it still needs a location, the code will then work out the height of that specific point, if the height is anywhere between 4 and 20, the tree mesh is created in that location and added into the vector of trees. This vector of trees is then referenced again later when it comes to drawing the entire scene.

## Skybox

After finalising my fractal trees, I began working on my seamless skybox. To accomplish this, I used new library called SOIL (Simple OpenGL Image Library).

unsigned int loadCubemap(vector<std::string> faces)

{

unsigned int textureID;

glGenTextures(1, &textureID);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, textureID);

int width, height, nrChannels;

for (unsigned int i = 0; i < faces.size(); i++)

{

std::string filepath = "Textures/skybox/" + faces[i] + ".jpg";

unsigned char\* skyImage = SOIL\_load\_image(filepath.c\_str(), &width, &height, 0, SOIL\_LOAD\_RGBA);

if (skyImage!=0)

{

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGBA, width, height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, skyImage);

}

else

{

std::cout << "Cubemap texture failed to load at path: " << faces[i] << std::endl;

}

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_R, GL\_CLAMP\_TO\_EDGE);

return textureID;

}

Figure – Skybox

The code shown in Figure 7 – Skybox loads jpg images from the source directory and returns the textureID of those images. This can then be used by my program to draw in the skybox, make it seamless and then make it follow the camera which imitates the skybox being infinite. An example of this can be seen in Figure 8 – Skybox example.

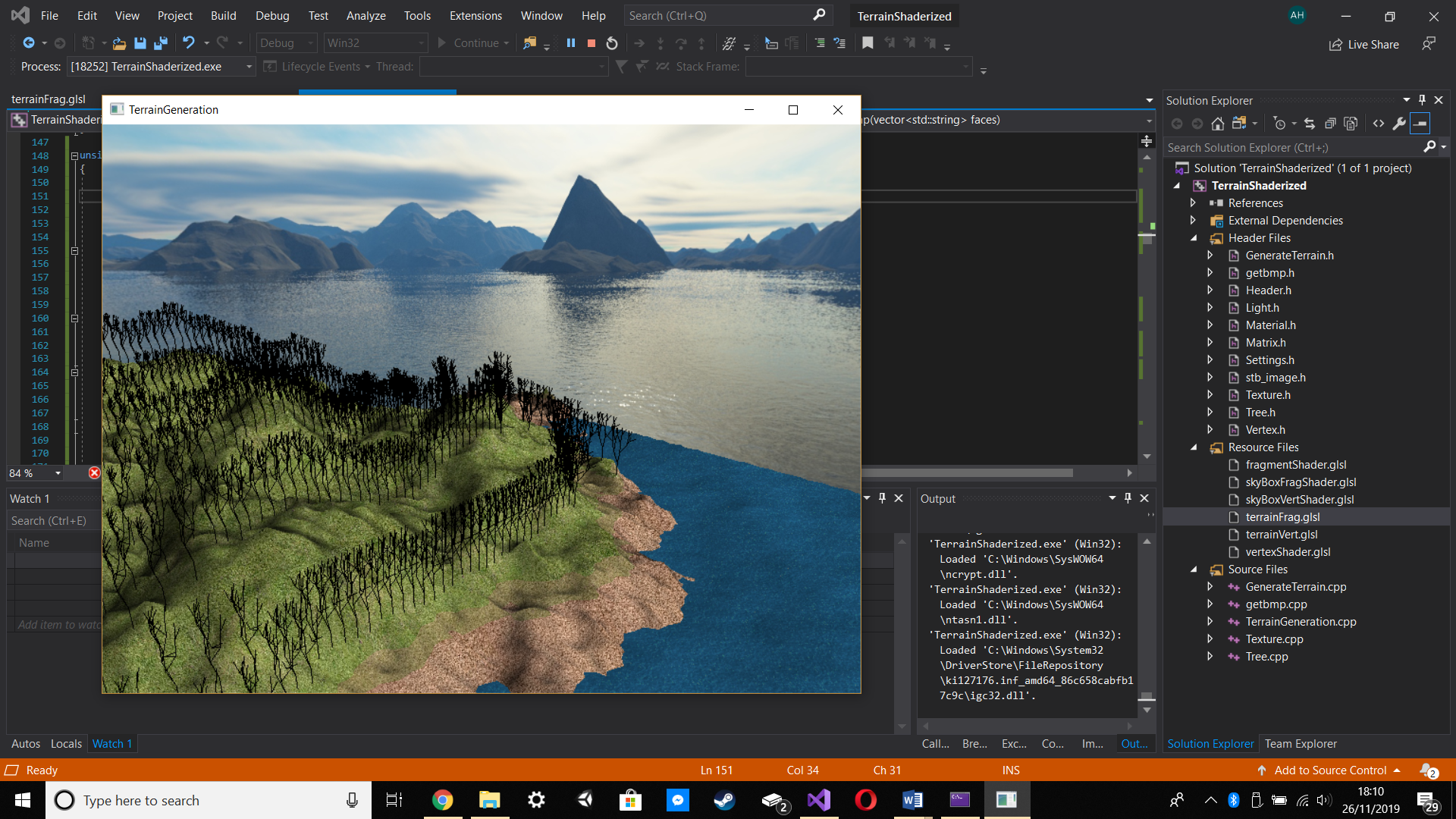


Figure – Skybox example

## Optimisation

To optimise my code, I enabled

glEnable(GL\_CULL\_FACE);

glCullFace(GL\_BACK);

This prevents the code from loading faces that are back facing from the camera. This can be seen in Figure 7 – Optimisation.

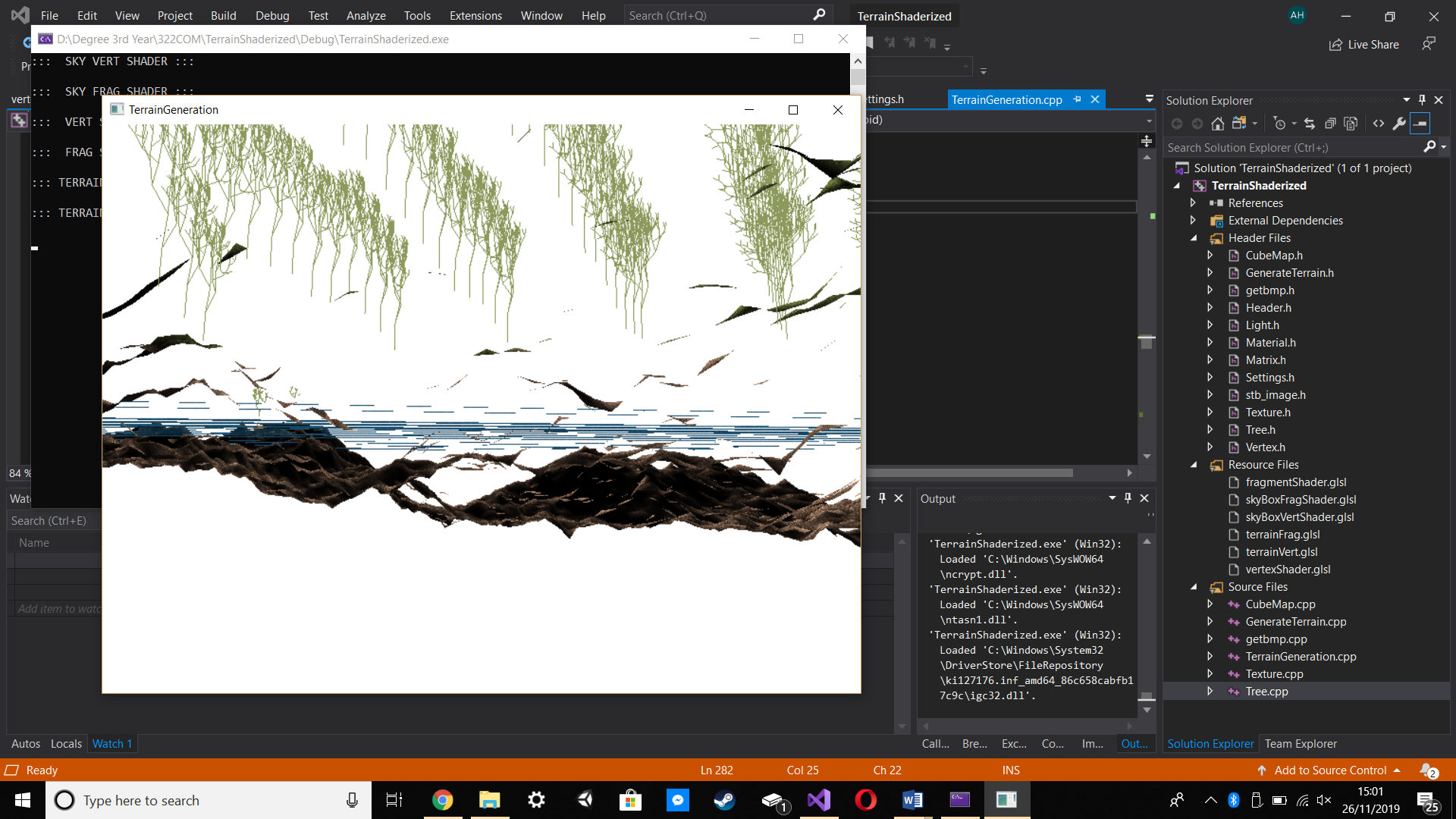


Figure – Optimisation

## 3D Camera Controls

The specification also asked for a 3D camera to be implemented into the scene. To accomplish this, I decided to use key presses to edit the positioning of the camera. This can be seen in Figure 8 - Camera Controls

void keyInput(unsigned char key, int x, int y)

{

switch (key)

{

case 27:

exit(0);

break;

case 'w':

cameraPos += cameraSpeed \* cameraFront;

break;

case 's':

cameraPos -= cameraSpeed \* cameraFront;

break;

case 'a':

cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

break;

case 'd':

cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;

break;

case 't':

if (debugMode) { debugMode = false; }

else { debugMode = true; }

break;

default:

break;

}

glutSwapBuffers();

}

Figure - Camera Controls

To add more functionality to this code I then added in the Glut function

glutPassiveMotionFunc(setMouseMove);

This function allows me to constantly move my mouse around in the scene so that the final overall effect is that the camera can fly around the scene through the user inputs.

# Reflective Review

When starting the project, I got the terrain to load with a wireframe so that I could see the effect of the diamond-square algorithm and built off that. I used OpenGL for my procedural generation over SDL since OpenGL would give me access to my own shaders. After the ray-casting assignment that I worked on for the last assignment I found the diffuse and ambient lighting effects that I used relatively simple, I however did not use Phong’s shading since I felt that that shading technique was not necessary as the terrain would not be as smooth as the primitives used in the last assignment.

Whilst I progressed through this assignment I learnt more about OpenGL and its capabilities, in this case if I were to redo the project I would also attempt at implementing leaves onto my trees as I was unable to do this as I did not understand the logic.

The lectures that I have attended in the forerunning to handing in this project have helped me to understand how to procedurally generate in almost real-time along with how to manipulate vertices in OpenGL to simulate an animated scene.

I am pleased that I managed to implement an infinite skybox for this project. I were to redo the project; however, I would want to attempt to implement a day/night cycle to improve to realism of the project. In terms of optimising my code, Culling the back-faces serves a purpose but if I were to redo the project I would also like to attempt multi-threading as it allows the program to use multiple CPU cores at the same time which would speed up the overall processing of my project.

## Video Link

<https://youtu.be/EXTRuz58kWA>

# References

[Figure 1 - Diamond 3](#_Toc25686044)

[Figure 2 - Square 4](#_Toc25686045)

[Figure 3 - Multitexture 4](#_Toc25686046)

[Figure 4 – Animation 5](#_Toc25686047)

[Figure 5 6](#_Toc25686048)

[Figure 6 - Tree Placement 7](#_Toc25686049)

[Figure 7 – Skybox 8](#_Toc25686050)

[Figure 8 – Skybox example 9](#_Toc25686051)

[Figure 9 – Optimisation 10](#_Toc25686052)

[Figure 10 - Camera Controls 11](#_Toc25686053)