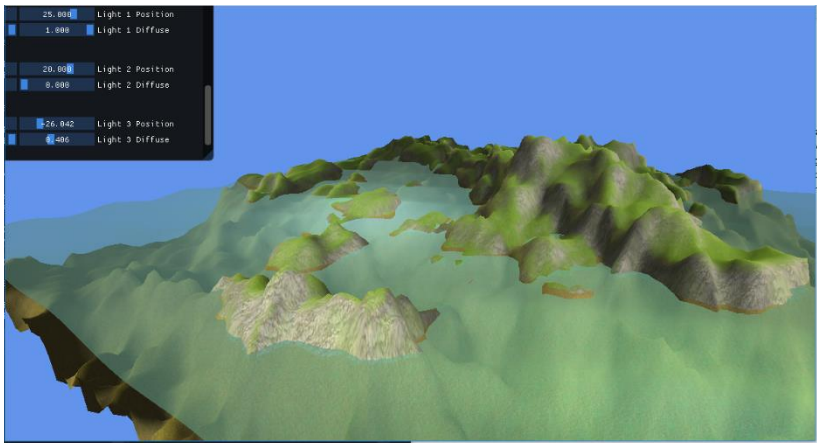
CMP305 Report

Lee Gillan

1701370

The Application Outline

The application is developed using C++ and DirectX 11 alongside ImGui and is made to produce procedurally generated terrain in real time that can be explored. It takes in adjustable (amplitude, frequency e.g.) values and uses them to create terrain using an algorithm. The generated terrain can then be modified by making use of ImGui to edit the values so that the terrain can be fine-tuned to look desirable. Techniques used to create generation are Faulting, Circular Faulting, Perlin Noise with some Improved Noise and Smoothing. Using these techniques, the terrain can be made to look like sharp rocks, tall mountains and other sorts of terrain. The texture of the terrain changes depending on the slope of the next point, if it is under the water or depending on the height the terrain is at. So, flatter areas look grassier and steeper areas are made to look rocky. Under water terrain has a sand texture applied and if the terrain goes beyond a limit up top it will become snowy. The camera provided is used so that the area is explorable when you are modifying the terrain. There are 4 lights that can be moved and changed to light the area up to see clearer or to create a moody landscape. The application makes use of vertex and pixel shaders to create the wavy effect of the water. This can also be changed through the ImGui.



```
25.000   Light 1 Position
1.000    Light 1 Diffuse

20.000   Light 2 Position
0.000    Light 2 Diffuse

-26.042  Light 3 Position
0.406    Light 3 Diffuse
```

Features

Procedural Terrain

The terrain is generating using a height map which is used to store the height values for each point on the terrain. This is created by using a plane mesh and looping through the size of the terrain's row and columns and applying values (generated by an algorithm that makes use of sin/cos, amplitude and frequency) to the height map array. This is then used to create the vertices of the plane to create the triangles that make it up. The indices and normals are then created and the normals are also smoothed out by averaging the neighbours. The buffers are also created or updated (if already made).

Wavy Water

The water waviness is controlled using ImGui. Using the alpha buffer, it has a transparency that can be modified to give it a different look. It is created the same way as the terrain.

Height/Slope Based Texturing

The terrain has 4 different textures that are used depending on the height or slope that it is. If the terrain is below sea level the texture will become a sand texture.

If the terrain is above a certain height it will become snowy to represent the mountainous terrain.

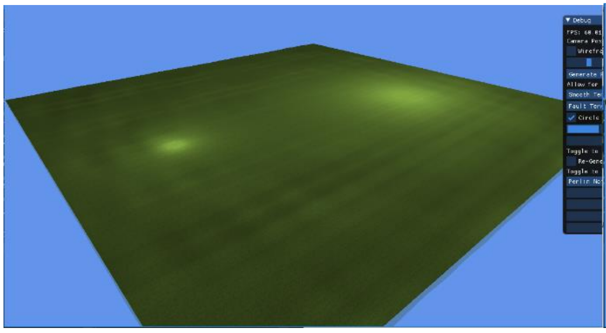
Depending on the slope, calculated by '1.0 – normal', the texture will blend from a rocky texture to grass. If the steepness is high it will be rocky and if it is on the flatter side, it will just be grass.

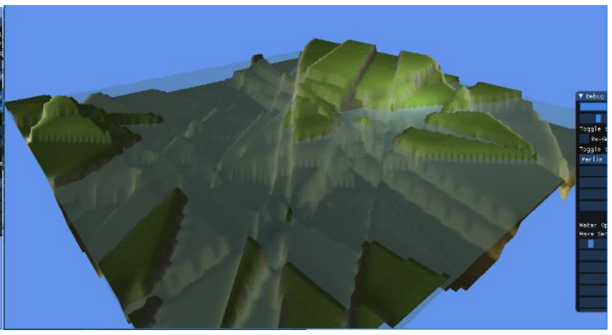
Techniques Used

There were multiple techniques used to produce the above terrain one of which is Faulting.

Faulting

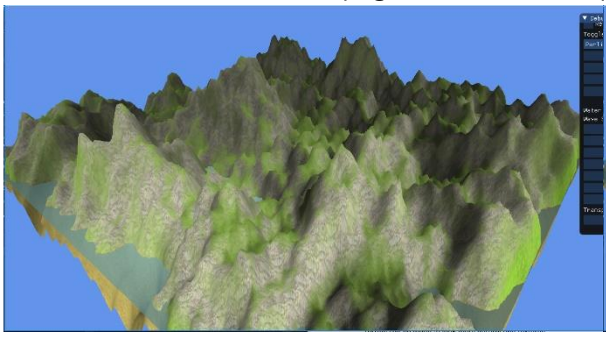
Faulting is a technique use to generate a terrain that has many vertices. A random line is drawn across the terrain to divide it into two sections. On one side of the line all the vertices will be displaced vertically. Through multiple runs of this technique the terrain will come out looking semi natural if there is some smoothing involved. In the application there is smoothing after several iterations so that the terrain doesn't become to spiky.

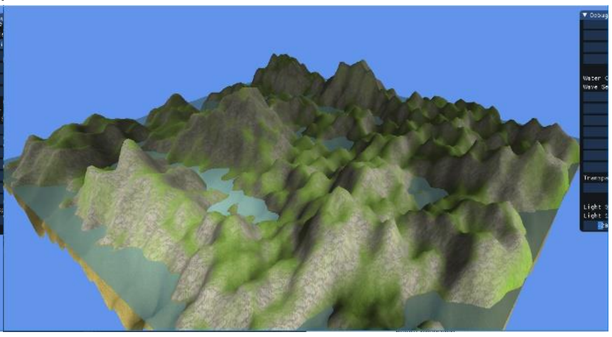

Before Faulting


After Faulting

Circular Faulting

Like Faulting but used to displace the terrain in random circles. The centre of the circle is a random point on the terrain and points inside the circle are displaced upward but, differing from faulting, no points are moved down. Iterating through this technique produces naturally looking terrain and smoothing is only needed after a lot of iterations. (e.g. 5000 iterations)


(5000 circle faults) Before Smoothing


After Smoothing

Another technique used is Perlin Noise.

Perlin Noise
Through using pseudo-random numbers applied to the terrain's height map the terrain can be made to look more natural and textured.
Using permutations and gradients noise is added to the height map points to change give the terrain a randomly looking natural effect.

Smoothing

By looping through the terrain and looking at each point's neighbouring height in the height map we can calculate an average and then divide it by how many neighbours the point has. This gives us the average height that the point is surrounded by. Using this the application then changes the height of the current point by the average height calculated * 0.5.

In-Depth Description of Key Procedural Features

Faulting

Using some random numbers and a few calculations we can get a line that passes through the terrain and displaces it on either side.

The application gets some random values through the rand() function in c++.

It then uses the random value in 'sin' and 'cos' to be used to get if the height map point is on either side. It then finds the distance the sides of the terrain are from each other and then gets a random point.

Using the distance formula 'a * x1 + b * z1 - c' the application can find the distance between two points on the terrain. The distance on one side of the terrain will be negative, positive on the other and '0' on the line.

If the terrain is centred on the origin, the distance from the line to the origin, i.e. the point is (0,0), is defined by 'c'. Meaning the value for 'c' can be smaller than the distance from the furthest point in the terrain to the origin. Can do this by using 'd = sqrt((res*res) + (res*res))' to get the furthest point on the terrain. 'c' can be defined as number in between -d/2 and d/2.

After doing this the application iterates though the height map array and displaces the height of the terrain up or down. The amount of iterations is passed through the function allowing for free modification of the terrain.

Circle Faulting

Circles with a random centre and a random radius are what gets displaced on the terrain to create some texture in the terrain in a random way. Points inside the circle are displaced upwards whereas others stay their original height.

To start the application finds random values between 0 and the resolution size. It also finds a value to make the circle that will get displaced.

Looping through the terrain size it checks to see the points in the terrain are in range of the circles centre using a modified distance equation where the random position is subtracted by the position of the loop, multiplying the value by 2 and dividing the value by the circle size.

Using the absolute values of the points calculated the application checks to see if it is within the circle and displaces them using code from the lighthouse3D

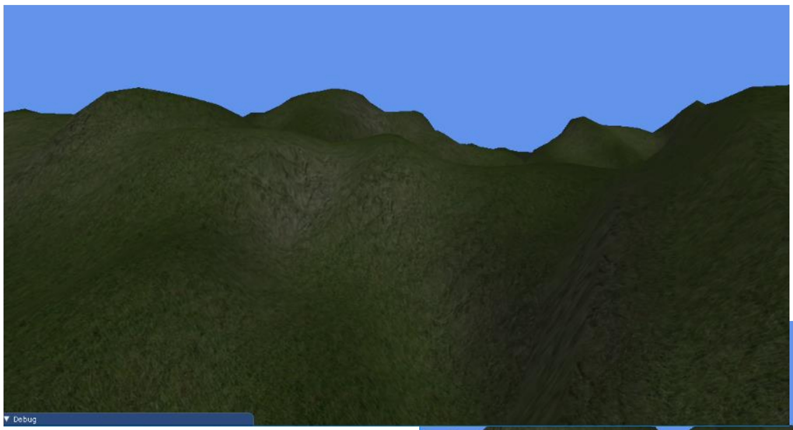
```
height(tx,tz) +=  disp/2 + cos(pd*3.14)*disp/2;
```

where displacement is 0.5 in the application and it uses conditional operator to assure the height map plus the displacement is below the max height.

The terrain is then changed to add the displacement that is calculated onto the original value that is stored in the height map and a circular bump in the terrain is created.

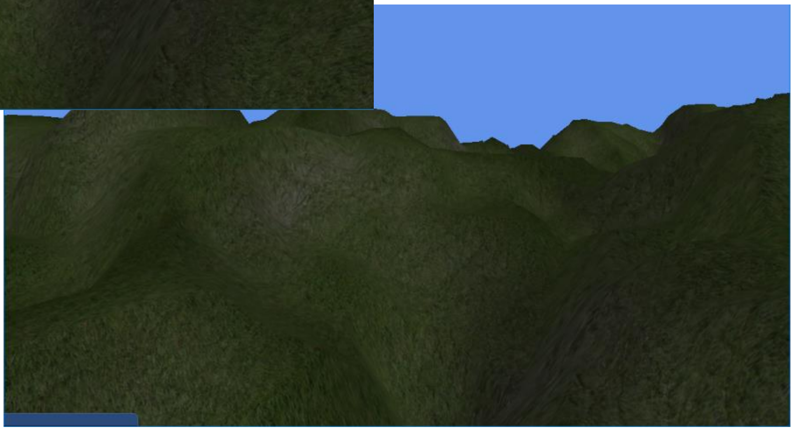
Perlin Noise

Perlin Noise created by Ken Perlin is a function that was created in 1988 and is a texture generation algorithm. In this application however, it is used to make the terrain look more realistic by adding pseudo random noise. The version used in the application was taken from the MyLearningSpace and modified a bit to work.



w/out Perlin Noise

No texture to the terrain



w/ Perlin Noise

More texture to the terrain

Perlin Noise is a lattice based noise function that uses the corner of lattice cells to define random noise values that are theb interpolated to compute a noise value at the location of the point given. For each result of the noise function, a dot product for each position and gradient vectors must be calculated for each in the terrain. Meaning Perlin Noise scales in complexity 0(2^n) for n dimensions.

Using Ken Perlins original C Implementation and the one given in MyLearningSpace:

To begin with Perlin Noise the application needs a list of permutations that act as the 'random' numbers. These are setup in the initialisation function which is called the first time Perlin is run in the program. This function also sets up our pseudo-random gradients (between -1 and 1) that will be used when the numbers from the permutation are used. The gradients that get setup are then normalised.

The Perlin Noise function used takes in a set of 3 values, the x, y from the heightmap and z. All the variables are then initialised that use the x, y and z. It also gets the pseudo-random number, from the permutation created, for the x boundaries which are used to get the psuedo-random values for the 4 corners of the point on the terrain. It then works out what is called the 'fade' or the 'blend curve' to get create a smoother transition from one point to another. After setting up all the variables in the noise function they are put to use by solving for all of the 3D points needed. E.g top, bottom. The points are then interpolated between the dot products calculated. These values are then uses to create the pseudo random noise that is applied to the height map.

Controls

Keyboard

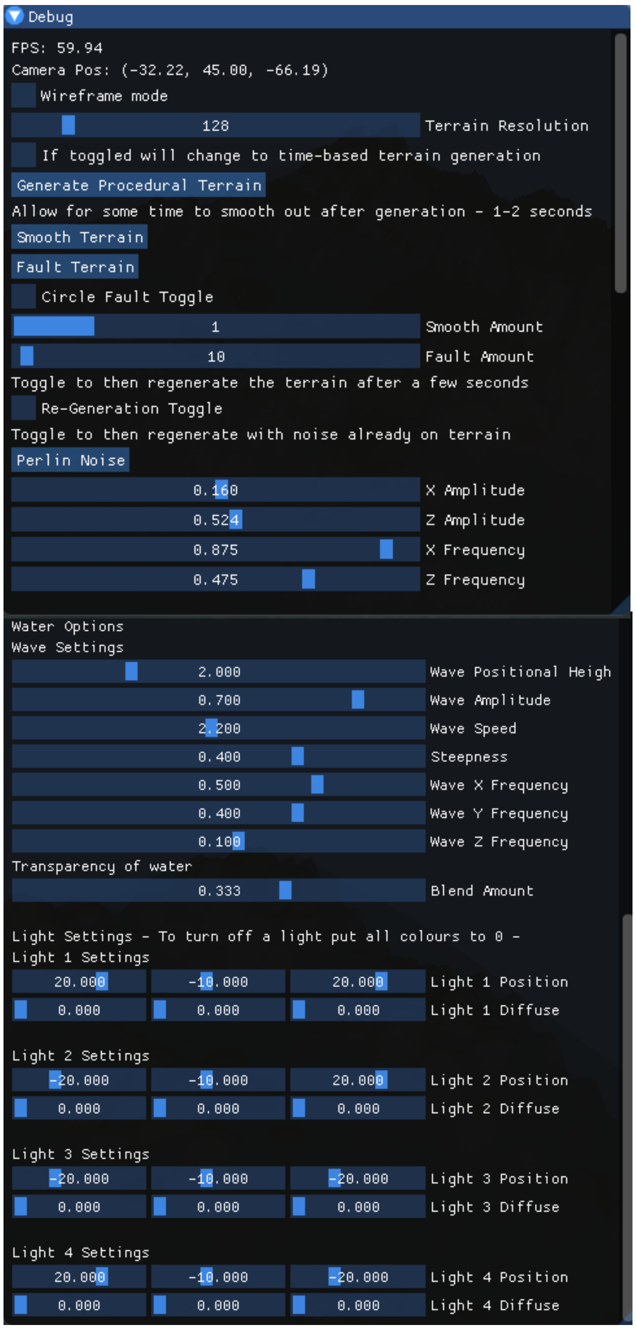
- Move camera – WASD
- Press SPACE to turn camera onto mouse control and back
- Q and E to move camera up and down

ImGui

- Wireframe mode
- Terrain Resolution

(Recommended 128/512)

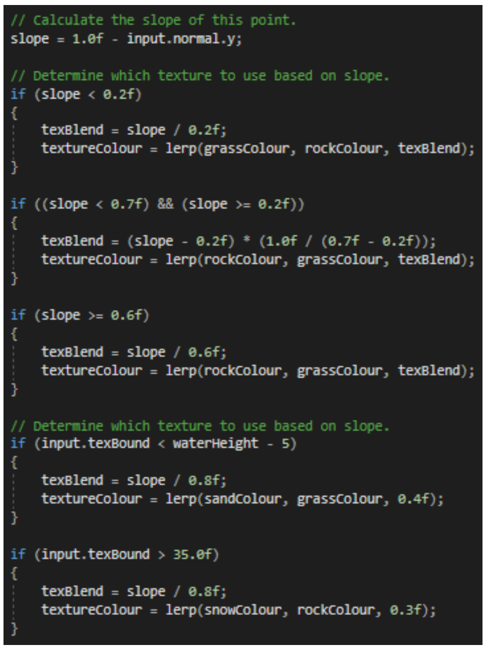
- Time Based Terrain Generation
- Terrain Generation that runs through all techniques
- Smooth out Terrain
- Fault Terrain
- Toggle Circle Faulting
- Change smoothing count
- Change faulting count
- Re-Generate terrain toggle (wait 3s)
- Use Perlin Noise on current terrain
- Change XZ Amplitude/Frequency
- Change Y Position of wave
- Change Amplitude of wave
- Change Speed
- Change Steepness
- XYZ Frequency
- Change Blend Amount of wave
- 4 Adjustable Lights



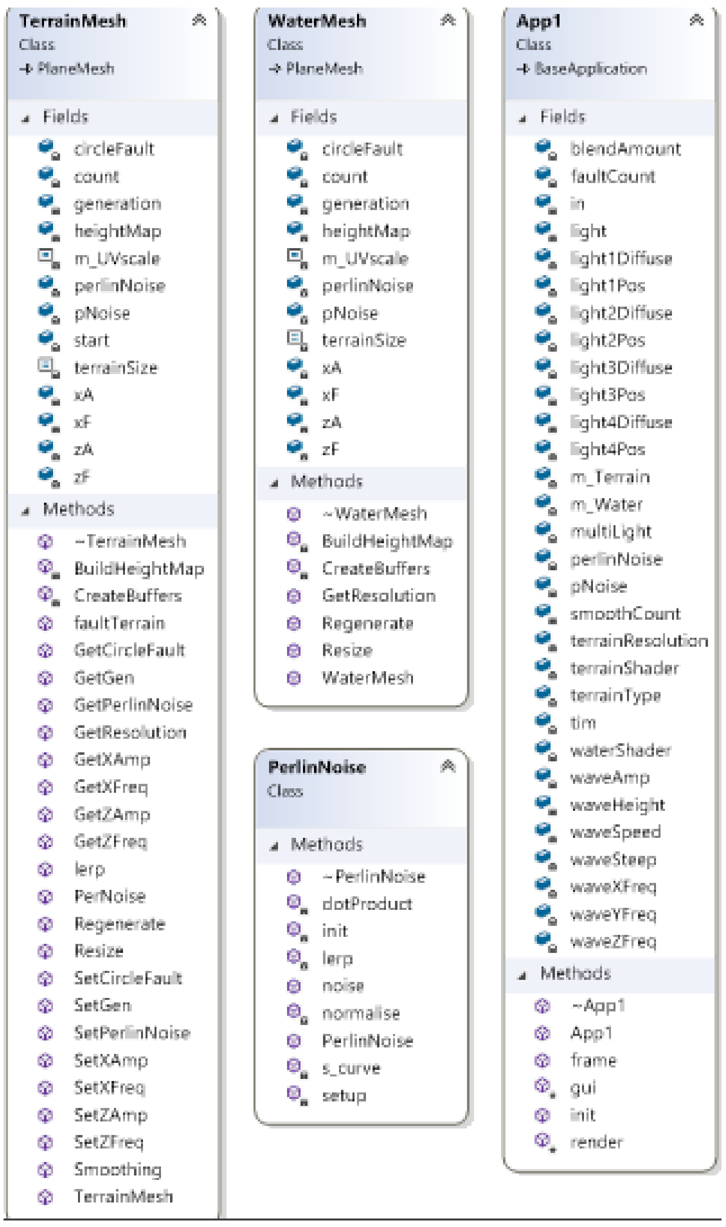
```
Debug
FPS: 59.94
Camera Pos: (-32.22, 45.00, -66.19)
Wireframe mode
128                                        Terrain Resolution
If toggled will change to time-based terrain generation
Generate Procedural Terrain
Allow for some time to smooth out after generation - 1-2 seconds
Smooth Terrain
Fault Terrain
Circle Fault Toggle
1                                          Smooth Amount
10                                         Fault Amount
Toggle to then regenerate the terrain after a few seconds
Re-Generation Toggle
Toggle to then regenerate with noise already on terrain
Perlin Noise
0.160                                      X Amplitude
0.524                                      Z Amplitude
0.875                                      X Frequency
0.475                                      Z Frequency
Water Options
Wave Settings
2.000                                      Wave Positional Heigh
0.700                                      Wave Amplitude
2.200                                      Wave Speed
0.400                                      Steepness
0.500                                      Wave X Frequency
0.400                                      Wave Y Frequency
0.100                                      Wave Z Frequency
Transparency of water
0.333                                      Blend Amount
Light Settings - To turn off a light put all colours to 0 -
Light 1 Settings
20.000      -10.000      20.000            Light 1 Position
0.000       0.000        0.000             Light 1 Diffuse
Light 2 Settings
-20.000     -10.000      20.000            Light 2 Position
0.000       0.000        0.000             Light 2 Diffuse
Light 3 Settings
-20.000     -10.000      -20.000           Light 3 Position
0.000       0.000        0.000             Light 3 Diffuse
Light 4 Settings
20.000      -10.000      -20.000           Light 4 Position
0.000       0.000        0.000             Light 4 Diffuse
```

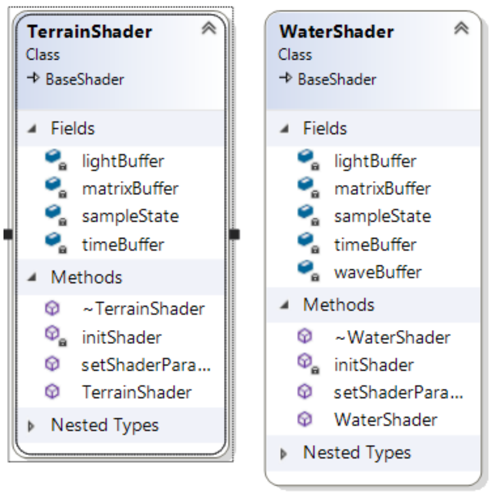
Code Structure

The code is formatted into classes and shaders. Each shader has its own Vertex and Pixel shader attached that gets loaded in.
Mesh Classes are also used which are derived form the plane mesh class.
The shaders contain constant buffers which take in values to be used in the scene and effect the mesh they are attached to. The constant buffers have a specific rule that must abided by which is that when you pack a buffer it must be in multiples of 16 meaning that you must use paddings when you don't fill the space.
The Water Shader is used to move make the water look as if it is a wave. It gets passed delta time and changes the y position of the point on the heightmap to simulate this. It also takes light values so that it is affected by all the light in the scene which is passed from the vertex to the pixel shader.
The Terrain Shader is like the water shader except the terrain's vertices get changed in the mesh class instead of the vertex. The pixel shader is used for changing the terrain textures at different heights/slopes.
Code for the texture changing based on height/slope: -



```
// Calculate the slope of this point.
slope = 1.0f - input.normal.y;

// Determine which texture to use based on slope.
if (slope < 0.2f)
{
    texBlend = slope / 0.2f;
    textureColour = lerp(grassColour, rockColour, texBlend);
}

if ((slope < 0.7f) && (slope >= 0.2f))
{
    texBlend = (slope - 0.2f) * (1.0f / (0.7f - 0.2f));
    textureColour = lerp(rockColour, grassColour, texBlend);
}

if (slope >= 0.6f)
{
    texBlend = slope / 0.6f;
    textureColour = lerp(rockColour, grassColour, texBlend);
}

// Determine which texture to use based on slope.
if (input.texBound < waterHeight - 5)
{
    texBlend = slope / 0.8f;
    textureColour = lerp(sandColour, grassColour, 0.4f);
}

if (input.texBound > 35.0f)
{
    texBlend = slope / 0.8f;
    textureColour = lerp(snowColour, rockColour, 0.3f);
}
```





The Perlin Noise algorithm has its own class that gets instantiated in the terrain mesh class to be used. There, the noise function gets called to be applied to the terrain's height map.

In the Terrain Mesh class Getters and Setters are used for values that are changed consistently. e.g. amplitude and frequency. This class is also where the faulting, circle faulting and smoothing functions are stored.

The application class controls the program as it handles the rendering of the scene and the ImGui settings. The textures, lights and shaders are initialised into the application here.

Critical Analysis

The briefs outline describes an application that has procedural content generation as a live visual or functioning component and that it should have the ability to effectively visualise the procedural content within. It also states that the application should eb built using DirectX 11.

The application created has the above requirements due to the terrain being created is produced using multiple algorithms such as Faulting, Circle Faulting, Smoothing and Perlin Noise. ImGui has been used so that most of the parameters to do with the generation of the terrain and the water is modifiable meaning that there is a lot interaction within the application. Allowing the terrain to changed and created however is wanted.

There is a button to generate a representative random procedurally generated terrain so that it easy to demonstrate.

Code Efficiency

Perlin Noise could have been ported to be used inside the vertex shader to take the load off the CPU and the same for the terrain generation.

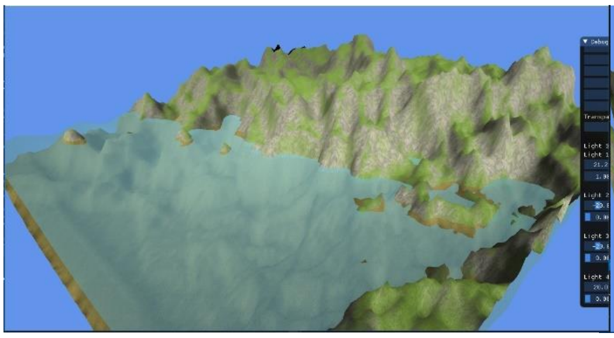
Reflection

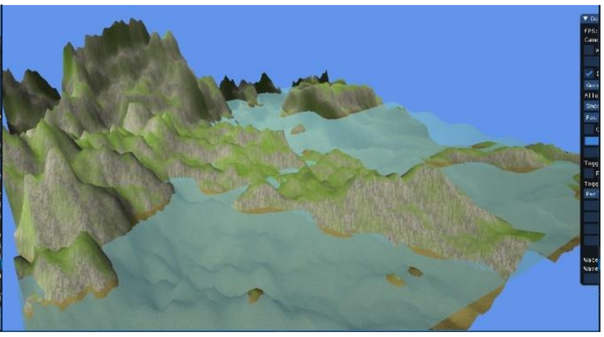
From the start a lot has been learned from working with procedurally generated content. Mostly by playing about with Perlin Noise and Faulting to understand the inner workings. Using shaders has been a learning curve that proved difficult due the debugging nature of them but has been proven a useful skill to learn.

Trials and Tribulations

Randomness

Using random numbers to fault – Using just the ‘rand()’ and ‘RAND_MAX’ made a pseudo random change in the terrain and would change in the same order of numbers e.g. waveHeight would go in an order of 2->16->20->26 etc. After some research, including <ctime> to the application helped with the randomness by using ‘srand’ but then the faulting was affected. As the fault lines would be placed in the same spot until the time was updated, this has been kept in the application as it can create interesting terrain despite the faulting error.


w/out srand()


w/ srand()

Also, the application uses high-order bits for randomness in places where it looks for a random number between 2 values.

e.g. `(waveHeight = -2 + (float)(15.0 * (rand() / (RAND_MAX + 1.0)));)`

This randomness is used for things like circle size when faulting, wave height and finding the random value to start the fault line on.

Perlin Noise

Adding Perlin Noise was a challenge that proved difficult on top of understanding Perlin noise itself I kept getting an error about variables that weren’t initialised. The problem was solved when the variable was taken outside of the header class and turned into a global variable in the cpp file.

ssReferences

Faulting Algorithm – Lighthouse3D

http://www.lighthouse3d.com/opengl/terrain/index.php?impdetails

http://www.lighthouse3d.com/opengl/terrain/index.php?fault

Circle faulting

http://www.lighthouse3d.com/opengl/terrain/index.php?circles

Better Randomness – Stack Overflow thread

https://stackoverflow.com/questions/1912199/better-random-algorithm

Ken Perlin’s Perlin Noise C Implementation – NYU.EDU

https://mrl.nyu.edu/~perlin/doc/oscar.html