

Procedural Methods

Hugh Tyson - 1700792

Introduction

This project's aim was to produce a finished application that utilised both procedural techniques and algorithms to create a realistic looking terrain. Developed using C++ in Visual Studio 2017, the application further makes use of the provided DirectX11 framework and ImGui API.

The application begins by rendering a flat plane mesh. The user is then provided the ability to manipulate the mesh by applying multiple procedural techniques. This can be achieved via controls that are provided within the ImGui window. Techniques that can be applied include multiple variations of Perlin Noise such as: Fractional Brownian Motion, Rigid and Inverse Rigid Noise. Furthermore, the application includes other techniques such as: Thermal Erosion, Fault Line Algorithm, Terracing, Terracing and Terrain Smoothing. Finally, the application further allows the user to toggle on/off a plane mesh that simulates water.



Application Design

The Terrain Generator application makes use of multiple classes in the DirectX11 Framework to create the 3D scene. However, the main classes that were developed upon throughout this project were: the *App*, *Terrain Mesh*, *Water Mesh* and *Post Processing*.

The *App* class is responsible for the main logic behind the program. It controls all Input for the application and is responsible for rendering the scene to the window. There are two meshes in this scene: one for the Terrain and one for the Water.

All functions that alter the Terrain Mesh can be called from the ImGui window. This window contains buttons that call function, sliders that manipulate the values that the application uses and checkboxes for changing how the user views the scene e.g wireframe Mode or edge detection.

Rendering in the scene is accomplished through multiple passes. The first pass calls the render function of the LightShader, which uses *light_ps* and *light_vs*. If post processing is enabled, it will impact on how the scene is outputted to the window. If there is no effect then the meshes in the scene will be rendered straight to the window. Otherwise the scene gets render to a texture. This texture is then used in the Post Processing classes EdgeDetection function that utilises the *EdgeDetection_ps* and *edgeDetection_vs*. This then returns a texture that is rendered to the window.

The *TerrainMesh* class contains multiple functions that can be used to alter the terrains height map mesh. The height map is a one-dimensional float array that stores all the height values for the terrain. When any of the functions in the *Terrain Mesh* class are used to alter the height map value the mesh is regenerated. The Generate Mesh function recalculates all the vertex position and normals for the mesh, allowing it to have correct lighting every time it changes. The mesh is textured using three different textures, which changes to four if water is enabled. All these textures are passed into the *light_ps* and are blended together using linear interpolated, based on the slope at this point.

The *WaterMesh* class contains similar functions to the *TerrainMesh* class. It includes the same functions for calculating vertex position and normals for correct lighting. However, its height map is updated every frame, using the 2D Perlin Noise called from the *App Class*



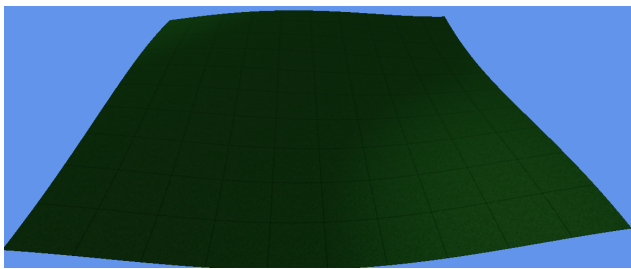
Techniques Used

Perlin Noise

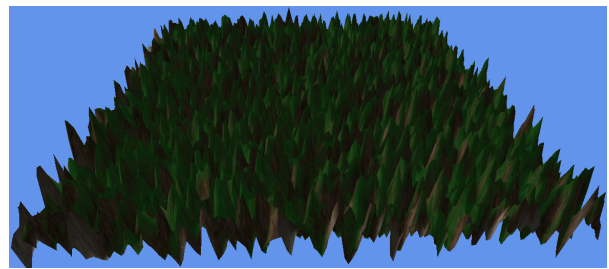
Created by Ken Perlin in the early 1980's while working on the original "Tron" Movie, Perlin Noise can be employed to add a more interesting look to computer graphics. The Perlin Noise algorithm is a method of gradient noise which generates a more naturally ordered set of pseudorandom numbers, resulting in a smoother transition between iterations. The code implemented for this was provided in the module. Furthermore, this code makes reference to Ken Perlin's Perlin algorithm[1].

For this application, Perlin Noise was exercised to achieve the desired finish. In the *CPerlinNoise Class* there are two Perlin Noise functions: one for 1D noise and one for 2D Noise. This application alters the Terrain Mesh by making use of the noise2 function, which generates 2D Perlin Noise. This works by passing in the co-ordinates at the point on the plane, these values are multiplied by a frequency and scale value. The returned

value from the Perlin Noise function is then multiplied by an amplitude value. By multiplying the point by these values, this allows the user more control over the creation and look of the terrain. By changing the frequency, the number of peaks can be manipulated. When using a lower frequency there are less peaks and troughs on the terrain. A lower value will create a more realistic terrain, while a higher frequency is better used to add finer more detail to the mesh. The scale variables are to ensure that the terrain looks consistent no matter what the resolution is. The amplitude value alters the height of the point.



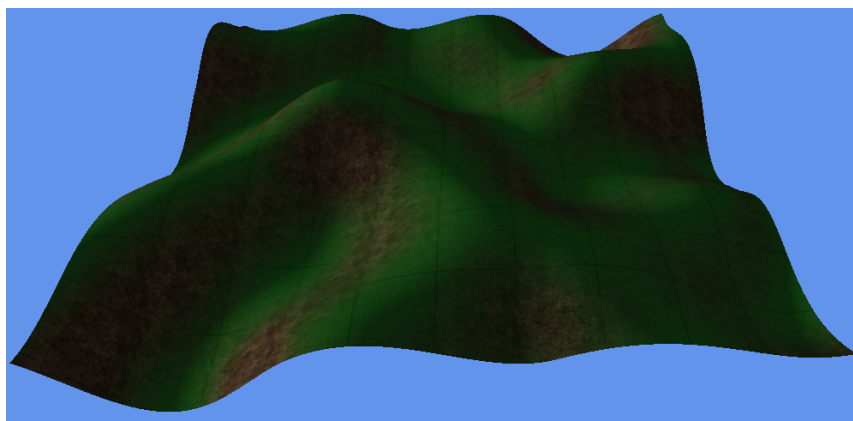
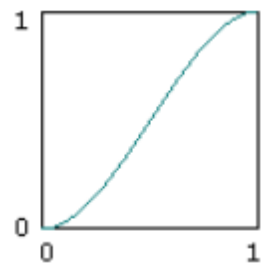
Lower Frequency Perlin



High Frequency Perlin

The 2D Perlin Noise function, works by having a space divided into a lattice in two dimensions. Each integer point (corner) of this lattice then has a 2D gradient assigned to it, and then the noise value is calculated via bilinear interpolation. This is achieved by calculating the position on the ease curve and then using that value to interpolate on the x then on the y using the result from the x interpolation.

Through using the ease curve, it allows the position to have a weighting depending on the four gradients being used. The easing curve is created using the function $f(t) = 3t^2 - 2t^3$, which creates the S-shaped curve shown on the left [2]. The X and Y of the point is then used in the function to determine the sX and sY which are used in the linear interpolations to calculate the Perlin Noise Value.



Perlin Noise created with Frequency of 0.31 and an Amplitude of 21

Fractional Brownian Motion

Fractional Brownian Motion (fBm) is a technique that makes use of the Perlin Noise algorithm to generate more realistic and detailed terrain. It works by adding multiple layers of Perlin Noise on top of each other, while doubling the frequency and halving the amplitude between each octave.

Through the ImGui the user has the ability to change values that effect the Fractional Brownian Motion.

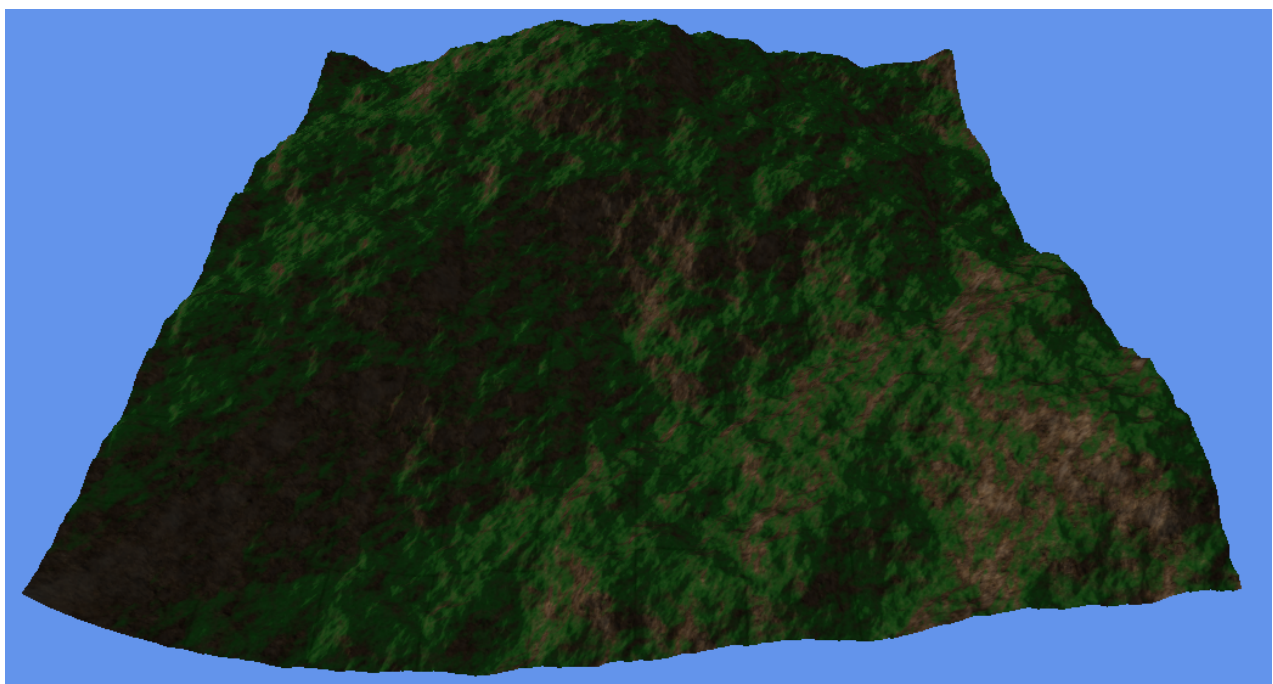
Octaves:

An increase in the amount of octaves will increase the amount of Perlin Noise layers added to the one position. One octave is equivalent to just one Perlin Noise pass on the position.

Amplitude and Frequency:

Employed to create more detailed and realistic terrain. When using the Fractional Brownian Motion function, the user is encouraged to use a higher amplitude and lower frequency to create a realistic looking terrain. As the frequency is doubled and amplitude is halved, finer detail is added to the terrain in the higher octaves.

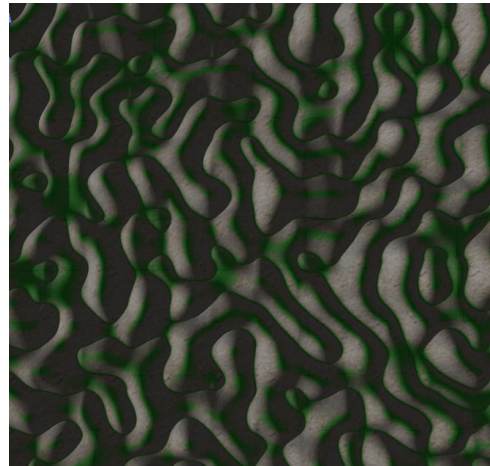
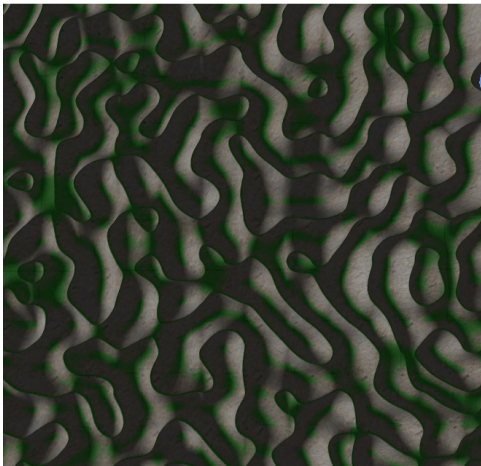
The image below displays an example of Fractional Brownian Motion.



Octaves - 13 Amplitude - 24 Frequency - 24

Rigid Noise

Rigid Noise is technique that uses the Perlin Noise algorithm. Created by finding the absolute of the Perlin Noise value, making all negative values positive, and subtracting it from 1. By getting the inverse of these ridges on the mesh, more realistic terrain can be created. This Rigid Noise value will be added onto the height maps current value, allowing the user to increase the level of detail of the heightmap.



The left image shows the height map before it gets inverted, therefore we can see the ridges are the higher points. The right image is the inverted height map where the ridges are lower and the internal areas within the ridges is pushed up. This method was adapted from online resources [3].

Terracing

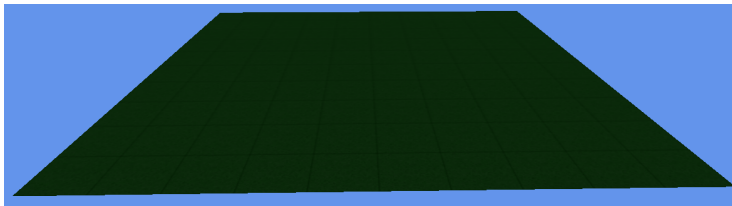
A terracing effect can be created via an already created height map. This technique results in the height map displaying a more step-based difference between values instead of a gradient. To create this effect the height map value is rounded by using the floor function, this value is then divided by a value between 0.1 and 1 to give a different step value. This value can be changed with a slider in the ImGui window.



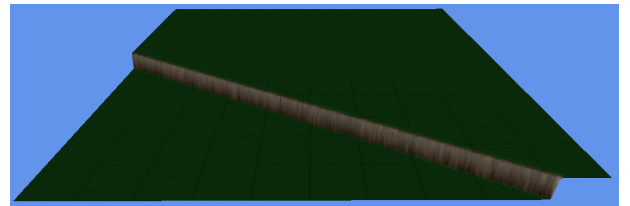
FaultLine Algorithm

```
//calculate the height increase value using a random value  
float height_increase = 5.f - ((float)(rand() % 100) / 100) * 10.f;
```

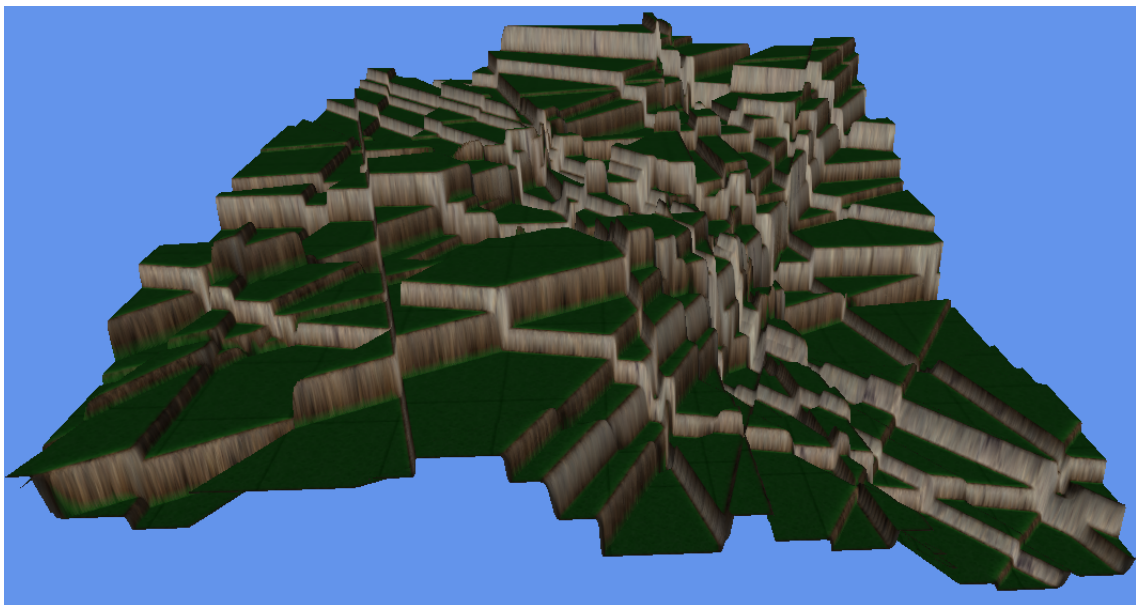
Implementation of the FaultLine Algorithm was an added technique to allow more 'realistic' geological features to be applied to the Terrain Mesh. To begin, 2 random points on the terrain mesh are used to build a vector. The function then loops through every position on the terrain, creating a vector between this position and the first random point. The cross product of these two vectors can then be calculated. This result will be negative for all points on one side, positive on the other and 0 when the point falls upon the line. If the outcomes y component was positive, then a pre-calculated height (see above) was added to the position's height map value. This results in a noticeable difference in the terrain height. This method works best when combined with other techniques to create a realistic terrain.



Flat Terrain



One fault



Multiple faults

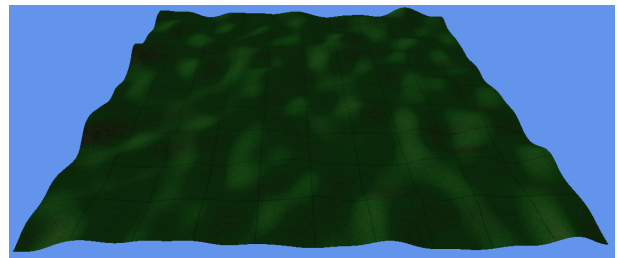
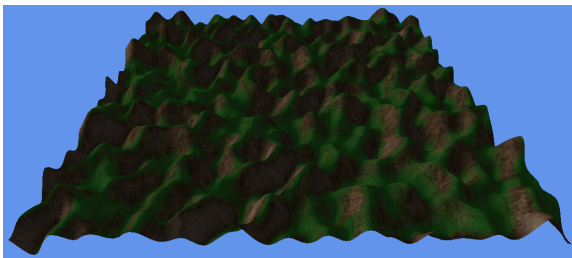
Smoothing

First, the terrain smoothing produces a copy of the height map that has been created. Each point on the terrain is looped over and the average of its Moore Neighbourhood (in total eight surrounding points) is calculated.

h_1	h_2	h_3
h_4	h	h_5
h_6	h_7	h_8

Moore Neighbourhood [4]

This calculated value is then saved to the copy of the heightmap. This is to stop the next few loops using an already altered result to create their new average result.



The left image demonstrates a rough terrain created with Perlin Noise. The right image shows the same terrain after being smoothed.

Thermal Erosion

The thermal erosion algorithm that has been implemented was adapted from the “Realtime Procedural Terrain Generation” paper[4] with assistance from other online resources[5]. The algorithm simulates a method of erosion that can be seen in the real

world. Erosion results in organic material (sediment) breaking down and coming away from the side of a mountain/cliff and being deposited further down the slope.

The algorithm works by calculating the difference in height between the initial point and its Moore Neighbourhood (eight surrounding points). Four variables are used to check the Moore Neighbourhood, these are clamped between 0 and resolution- 1 to ensure the algorithm does not read outside the boundaries of the height map array.

h_1	h_2	h_3
h_4	h	h_5
h_6	h_7	h_8

```
#define clamp(value,minimum,maximum) max(min(value,maximum),minimum)

int x1 = j - 1;
x1 = clamp(x1, 0, resolution - 1);
```

The height difference between all these points are then checked to determine if it is greater than the Talus angle, the angle at which sediment is deposited. If the difference is greater than the Talus angle then this value is added to the total difference. All eight values are also compared to determine which point has the greatest value, this value is then saved to be utilised within the thermal erosion calculation.

```
_copy[(j*resolution) + y1] += c * (max_dif - talus) * (heightDifference[i] / total_dif);
```

All surrounding points are then altered using the algorithm shown above. This simulates the movement of material down the slope. This new value is then saved to a copy of the original height map, which is done to stop future iterations using the new height map value.

Below displays an example of thermal erosion that uses a constant value of 0.5 and a Talus angle of 4/resolution.



Before Thermal Erosion

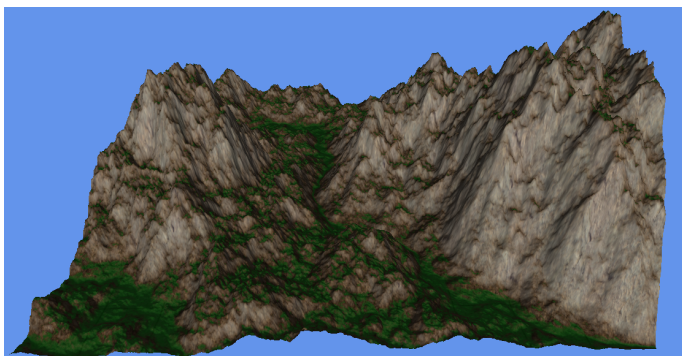


After Thermal Erosion

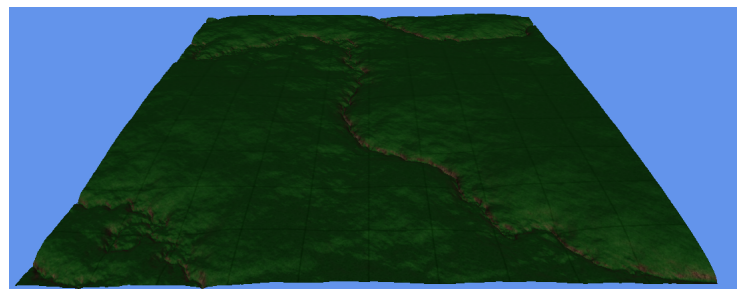
Redistribution

The redistribution function is used to create valley like terrain. This function is only used to alter already created height maps. This method was adapted from Red Blue Game Procedural Terrain Blog [6]. The algorithm works by looping through each position on the terrain mesh, at each point the absolute value of the height map value is taken. This value is then raised to a power. If the power is greater than one then the higher points are increase and the lower points are lowered towards zero. By applying this method first with a higher value then a lower value it creates interesting terrain. This terrain is a particularly excelent technique when applying more detail to with Fractional Brownian Motion and Rigid Noise.

Below, the two images are examples of using the redistribution function. The terrain was created with Fractional Brownian Motion and then manipulated with a power value of 1.5 to create the first terrain. These new heightmap values were then used with a power of 0.3 to create the second terrain.



Power Value of 1.5



Power Value of 0.3

Water

There is a second plane mesh in the application that acts as water in the scene. To produce this effect a height map is updated every frame with 2D Perlin Noise.

Each point on the water mesh is looped over and the x and y position of that position is used in the creation of the new heightmap value. Both values are multiplied by a scale and frequency variable which works the same way as 2D Perlin Noise. However, a time variable is now used to push the Perlin Noise map across the plane. The time variable is also added to the x variable, this is used to create the effect of flowing water.

The water texture has the ability to become transparent. This effect is achieved by enabling alpha blending before the water mesh's information is passed into the *light_vs* and *water_ps*. In the pixel shader the water texture is sampled and then the alpha value is changed to a value between 0 and 1 to change the transparency. The user is given the control to change this. This texture is then combined with the light colour and outputted.

```
float4 textureColor = texture0.Sample(Sampler0, input.tex);  
textureColor.a = transparency_value;
```

Slope Based Texturing

The terrain is textured using slope based texturing, which was adapted from the RasterTek Tutorial on Procedural Terrain Texturing[7].

The aim of this technique was to blend the texture depending on the steepness of mesh at every point on the height map. The *App* class passes three textures into the *LightShader*. **Grass, dirt and rock textures** which will be blended together depending on the slope value that is calculated. The ultimate aim was to elicit a grass and dirt blend where the mesh was flat, a rocky texture where the mesh had a large slope and a mix of dirt and rock textures when in-between.

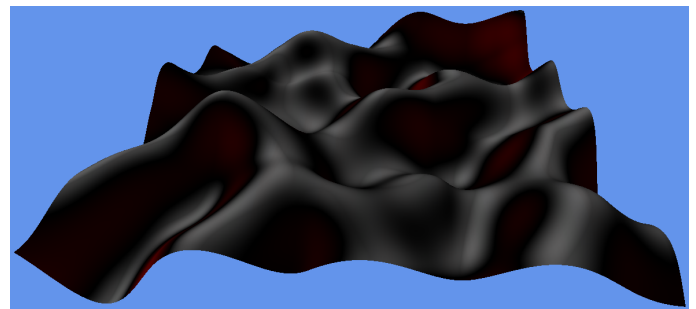
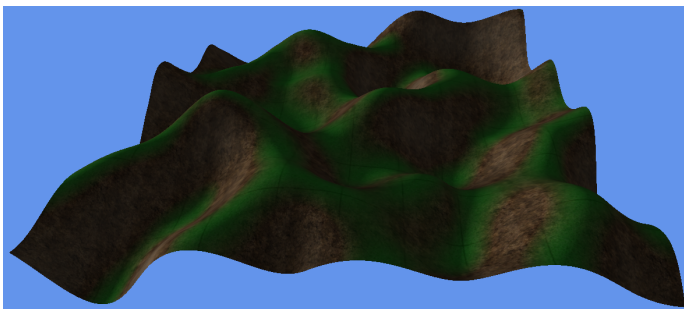
In the *light_ps* all the textures are sampled for that points texture coordinates and the slope value were calculated by subtracting the Y normal from 1. The texture colour was then calculated based on this sloping value. If the slope value was less than 0.2 (flat terrain) then a linear interpolation was made between the grass and dirt texture, with a blend amount of the slope value divided by 0.2. If it was between 0.2 and 0.7 (gentle slope) then a linear interpolation between the dirt and rock value was made with the blend amount being calculated by subtracting 0.2 from the slope value and multiplying it by 1 divided by 0.7 subtracted by 0.2.

```
blendAmount = (slope - 0.2f) * (1.0f / (0.7f - 0.2f));
```

If the slope value was greater than 0.7 (steep terrain) then the rock texture colour would be applied.

Personally, I found that Slope Based texturing looked better than height based texturing as it allowed blending between the textures. The image on the left shows the terrain with the normal textures applied, however this is difficult to see the blending between the textures.

The image on the right uses three textures, a red, black and white texture that makes it easier to see the blending.



If the user has toggled the water to be rendered then the application makes use of height based texturing. If the position on the mesh is underneath the water height value then a linear interpolation between the sand texture and the current texture colour, which is based on the slope, is performed. The *light_ps* further changes the texture of any of the positions just above the water height level, texturing the pixel with the sand texture.

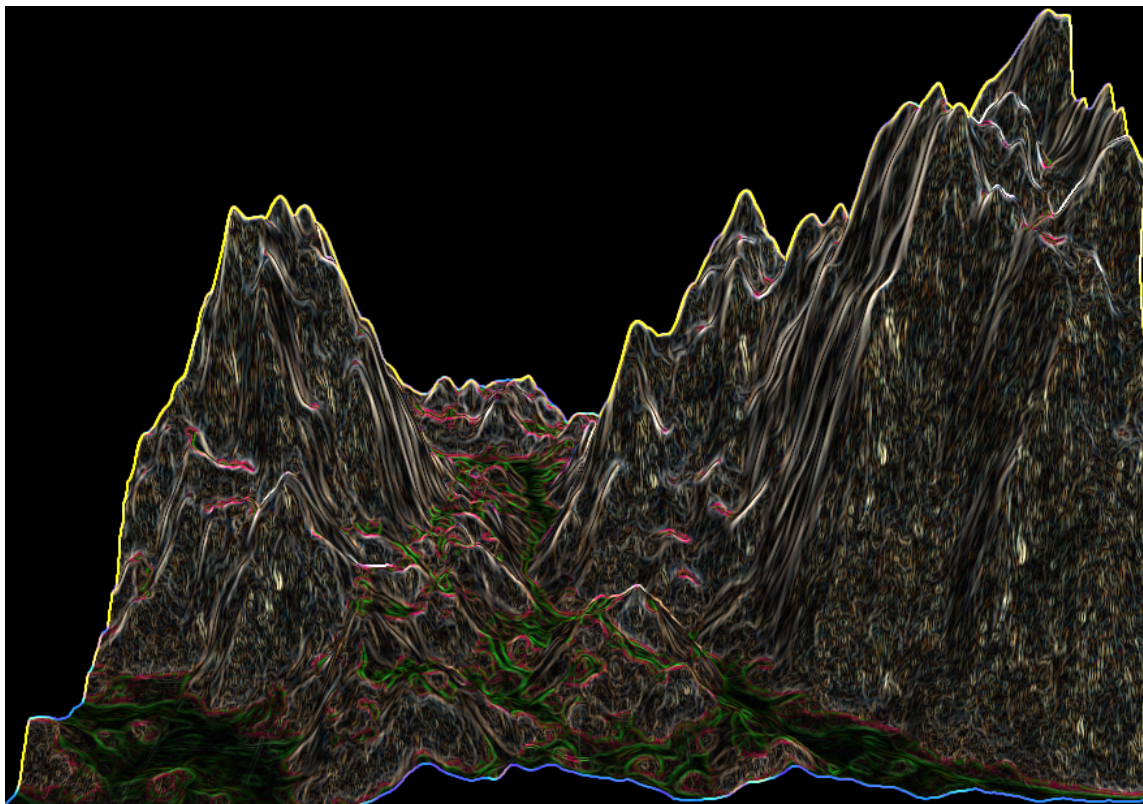
Post Processing

The application makes use of Sobel Edge Detection[8] as a post processing effect. The Edge Detection function in the *Post Processing Class* receives a *RenderTexture* of the scene that is being created. This image is then passed into the *Edge_Detection_ps*. The current pixel and its surrounding eight neighbouring pixels are sampled. The vertical and horizontal Sobel convolution kernels are then created, based off of the values from the sampled textures.

-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

The output colour is then calculated by square rooting both the square of these values together. This texture is then outputted to the screen.



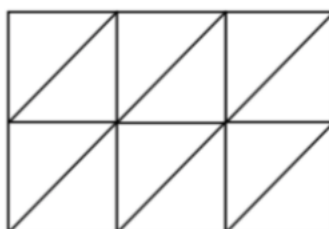
Critical Evaluation

When I started this project, I aspired to create an application that could alter a mesh in order to produce a realistic looking terrain. Upon evaluating the final outcomes, I have a great sense that I have achieved what I set out to accomplish. However, that is not to exclude the different methods and techniques that could have been implemented to improve this application.

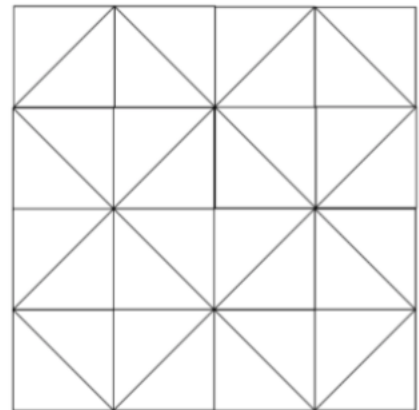
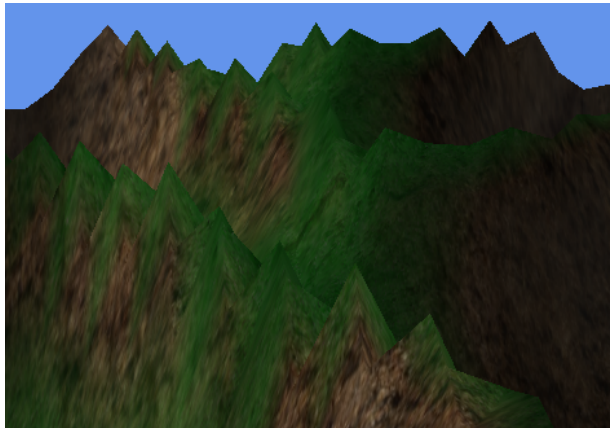
First, in hindsight I would have liked to implemented a different style of erosion to combine with Thermal Erosion, particularly the use of Hydraulic Erosion. As I understand the technique, the Hydraulic Erosion would simulate water flowing down a slope, picking up loose sediment as it flows and then carrying it down the slope and depositing it at low points in the terrain. From researching implementation of this algorithm, I understand that droplets would be generated that would move down the terrain picking up sediment as it moves down a steeper gradient while depositing sediment on flatter terrain. If implemented, this technique could have added to my terrain by building upon its comparison to real life phenomenon.

Second, the addition of clouds to the environment would have also brought more realism to the simulation. Clouds could be achieved by manipulating the texture on a plane using the 2D Perlin Noise function that was used in the terrain and water implementation. However, instead of using the x and y value of terrain the UV co-ordinates of the texture would have instead been used. The deltatime variable would still be used as the 3rd value and be used to push the values along the x axis to create realistic moving clouds.

Furthermore, it would have been beneficial to change the mesh patter of the terrain mesh object. In the current application the terrain mesh is not optimal for creating a height map and currently the application makes use of the standard terrain mesh that was provided for us.



Sometimes when using this mesh, it may result in the terrain having jagged edges. Personally, I felt that this jagged appearance lacks the realism I was trying to accomplish and so I would change this to a quilt type mesh that would be more appropriate for the application.



Upon reflection, I would argue the main criticism of the application is the time it takes to load high detailed terrain, while at a higher resolution. This drawback is due to the application primarily running on the CPU. In order to improve on this aspect, it would be worthwhile changing the application to utilise the DirectX11 pipeline to a greater effect. In its current capacity, it is only used for more advanced texturing and applying post processing effect. But, if the application was to be rebuilt it would make use of different techniques, therefore improving the running of the application. Different techniques could be used such as, using a computer shader for all calculations on the terrain. Furthermore, making use of the tessellation stage in the pipeline which would alter the level of detail based on the distance from the camera and complexity of the piece of terrain.

This application that was developed was compared to another Terrain Generator application, created by Tomas Hudak [9], who had developed a GPU based application. The applications were compared, based on the time it took to generate a mesh that had eight octaves of Fractional Brownian Motion passed over it, at different terrain resolutions.

GPU Times

Resolution	Time taken (s)
255	0.0076
1024	0.06
2049	0.245

CPU Time

Resolution	Time taken (s)
255	0.28
1024	4.46
2049	17.58

As can be observed from the times in the tables above the GPU is vastly quicker at generating Terrain than a CPU is.

Controls / User Guide

Camera Controls

Key	Function
W	Move forward
S	Move Backwards
A	Move Left
D	Move Right
Mouse Movement	Look Around

The following table displays all goals

ImGui Component	Function
Wireframe Mode - CheckBox	Toggles Wireframe mode on/off
Render Terrain - Checkbox	Toggles Render of Terrain Mesh on/off
Terrain Resolution - Slider Int	Changes the terrainResolution variable. Limits between 0 and 1024
Regenerate Terrain - Button	Calls Regenerate function in Terrain Mesh class - uses terrainResolution variable
Flatten - Button	Calls Flatten function in Terrain Mesh class
Smooth - Button	Calls Smooth function in Terrain Mesh class

Faultline - Button	Calls Faultline function in Terrain Mesh class
Terrain Frequency - Slider Float	Changes frequency variable. Limits between 0.01 to 0.5
Terrain Amplitude - Slider Float	Changes amplitude variable. Limits between 5 and 45
Perlin - Button	Calls PerlinNoise function in Terrain Mesh class - uses amplitude and frequency variables
Rigid Noise - Button	Calls RigidNoise function in Terrain Mesh class - uses amplitude and frequency variables
Rigid Noise - Button	Calls InverseRigidNoise function in Terrain Mesh class - uses amplitude and frequency variables
Terracing Octaves - Slider Float	Changes terracing_octaves variable. Limits between 0.1 and 1
Terracing - Button	Calls Terrace function in Terrain Mesh class - uses terracing_multiplier
Brownian Octaves - Slider Int	Changes brownian_octaves variable. Limits between 1 and 30
Brownian - Button	Calls BrownianMotion function in Terrain Mesh Class - uses brownian_octave, amplitude and frequency variable
Redistribution Power - Slider Float	Changes power variable. Limits between 0.1 and 1.5
Redistribution - Button	Calls Redistribution function in Terrain Mesh Class - uses power variable
Thermal - Button	Calls ThermalErosion function in terrain Mesh Class
Use Colour - Checkbox	Changes what texture is used in the light_ps
Enable Water - Checkbox	Toggles Render of Water Mesh on/off
Water Frequency - Slider Float	Changes water_frequency variable. Limits between 0.01 and 0.5
Water Amplitude - Slider Float	Changes water_amplitude variable. Limits between 1 and 4
Wave Speed - Slider Float	Changes speed variable. Limits between 0 and 4
Water Height - Slider Float	Changes water_height variable. Limits between -5 and 7
Transparent Value - Slider Float	Changes transparent_value. Limits between 0.1 and 1
EdgeDetection - CheckBox	Toggles use of EdgeDetection Post Processing on/off

Conclusion

To conclude, I have genuinely enjoyed both the challenges and accomplishments that I have experienced when developing this project and I am satisfied with the application that has been produced. Despite my initial struggle implementing basic procedural algorithms and my lack of experience of how the framework worked, through working through problems and using online resources for help, I became immersed in the development of this application and creating methods that resulted in more realistic terrain.

Upon reflection, I feel working and creating this application has vastly increased my knowledge and awareness of procedural techniques and how they work. I further feel that with the insight I have gained throughout this project, I have developed an interest in researching procedural algorithms.

References

- [1] - Perlin, K. (2019). *Ken's Academy Award*. [online] Mrl.nyu.edu. Available at: <https://mrl.nyu.edu/~perlin/doc/oscar.html> [Accessed 5 Oct. 2019].
- [2] - Mzucker.github.io. (2019). *The Perlin noise math FAQ*. [online] Available at: <https://mzucker.github.io/html/perlin-noise-math-faq.html> [Accessed 6 Dec. 2019].
- [3] - noise, P. and Záborský, M. (2019). *Procedural Terrain with ridged fractal noise*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/36796829/procedural-terrain-with-ridged-fractal-noise> [Accessed 12 Nov. 2019].
- [4] - Anon, (2019). [online] Available at: <http://web.mit.edu/http://web.mit.edu/cesium/Public/terrain.pdf> cesium/Public/terrain.pdf [Accessed 21 Nov. 2019].
- [5] - Craig, J. (2019). *Thermal Erosion*. [online] Gutgames.com. Available at: <http://www.gutgames.com/post/Thermal-Erosion.aspx> [Accessed 29 Oct. 2019].
- [6] - Redblobgames.com. (2019). *Making maps with noise functions*. [online] Available at: <https://www.redblobgames.com/maps/terrain-from-noise/> [Accessed 3 Dec. 2019].
- [7] - Rastertek.com. (2019). *Tutorial 13: Procedural Terrain Texturing*. [online] Available at: <http://www.rastertek.com/dx11ter13.html> [Accessed 26 Oct. 2019].
- [8] - Homepages.inf.ed.ac.uk. (2019). *Feature Detectors - Sobel Edge Detector*. [online] Available at: <https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm> [Accessed 14 Nov. 2019].
- [9] - student, Tomas Hudak.