Procedural Methods

**CMP305**

|  |  |
| --- | --- |
| Student Name | Keiran Millar |
| Student Number | 1502338 |

Contents

[Introduction 3](#_Toc512949215)

[Controls 3](#_Toc512949216)

[Camera 3](#_Toc512949217)

[Terrain 3](#_Toc512949218)

[Perlin Noise 3](#_Toc512949219)

[Fault Lines 5](#_Toc512949220)

[Clouds 5](#_Toc512949221)

[Slope Based Textures 5](#_Toc512949222)

[Little Extras 6](#_Toc512949223)

[Height Map Generation 6](#_Toc512949224)

[Smoothing 6](#_Toc512949225)

[Post Processing 6](#_Toc512949226)

[The Little Mechanic That Couldn’t 6](#_Toc512949227)

[Conclusion 7](#_Toc512949228)

# Introduction

The requirements for this project was to create a program that showcases different examples of the procedural methods we have learned in the lectures. The project also required a post processing effect and an interactive camera, so we could see the 3D effects of the methods created. The application made use of such techniques as, height map generation, faulting, Perlin noise, smoothing and slope-based textures to achieve these effects.

These techniques were created in the Rastertek[[1]](#endnote-1) DirectX11 framework using a flat plain and a range of user controls (see below) to create the desired effects. The interactive camera is created in the Rastertek framework.

# Controls

## Camera

W-Move Forward

S-Move Backwards

Arrows- Look Around

Page Up-Move Upward

Page Down-Move Downwards

B-Blur

## Terrain

Space-Add Perlin Noise To the Terrain

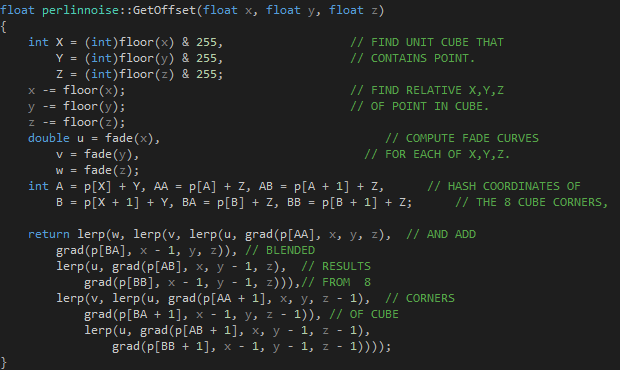
P-Smooth The Terrain

F-Apply Fault Lines

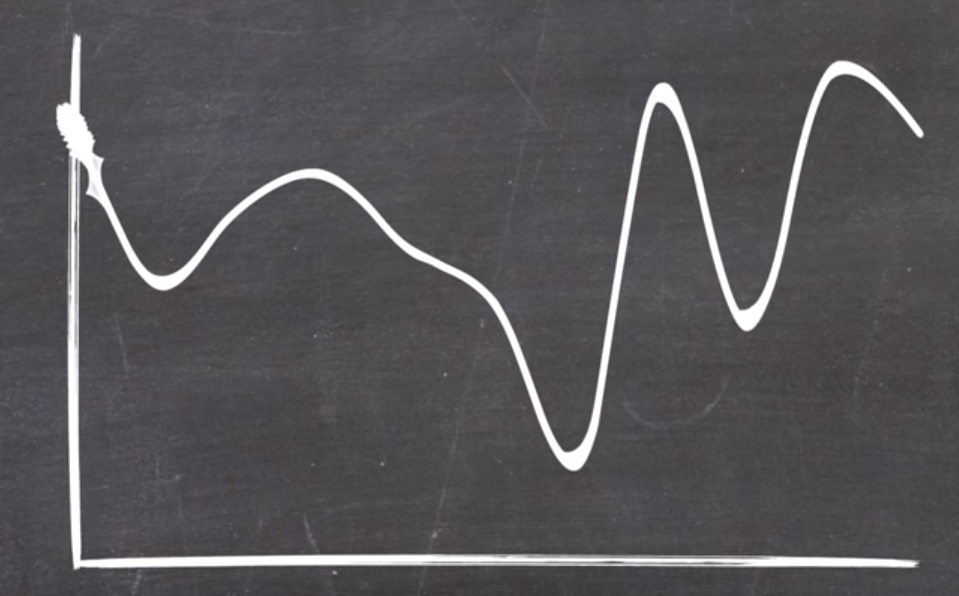
V-Apply Fault Circles

# Perlin Noise

Perlin noise is a way of generating seemingly random numbers that appear more natural than convention random numbers. It is achieved by using the function below that was created in a tutorial in class.



To improve on this however, we had to make use of octaves and Fractional Brownian Motion (FBM). This is done by making multiple loops through the GetOffset function and adding the numbers together. Further improving the numbers generated and giving a more realistic output as seen below:



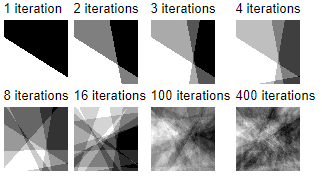
Left: Shows one pass through Perlin Noise. Right: Shows 3 Octaves of Perlin Noise.

Credit: The Game Theorists[[2]](#endnote-2)

There is also a project in the solution that will output a .ppm to show off Octaves in use.

# Fault Lines

To calculate fault lines, I used the Lighthouse[[3]](#endnote-3) algorithm to randomly generate a set number of points, then create a vector between these points. It would the go through every point in the terrain, generate a vector between every point and the random points and will do a cross product between these. If the y component is greater than 0 the height map is incremented by a pre-determined displacement variable, if it isn’t greater than 0 then the heightmap is decremented by said value. This creates a divide along the terrain, with enough iterations it can create an interesting terrain as can be seen below:



In my terrain class, every time you press F or V it will loop through this process 50 times. 50 seemed to be a good compromise between performance and amount of iterations between presses.

As well as fault lines, I also implemented fault circles, where it generates a set of random points on the terrain and will increment the terrain in a circle around these points. It gives off an easy way to generate more interesting terrain and give more changes in height for the terrain.

# Clouds

To create a more realistic scene, a sky dome was used that is drawn in before anything else and then everything is drawn on top of it, in theory the user should only see the dome but turning off the z buffer allows for the everything to be drawn on top of the dome, creating what seems to be a sky to the world. The cloud texture has a number added to it which creates motion, I changed this number to make it take in a number from my Perlin noise class to give a more natural motion to the clouds instead of having the clouds move at a linear speed.

# Slope Based Textures

When creating a landscape based on procedural methods it is inevitable that there will be points with a large gradient between them. To make the terrain more appealing, slope-based texturing can be applied to allow a larger gradient to have a different texture from flat areas. This can be used to show grass or snow on the ground, but it won’t appear on a “cliff face” as it is less likely to see it appear on those surfaces. This can be taken even further to consider other things, for example you could use this to make beaches and deserts or fields, depending on what type of are they are in.

# Little Extras

## Height Map Generation

To generate a heightmap for the terrain, two different routes can be taken. One is using a premade heightmap that is usually crafted to look appealing when on a terrain and the other is generating one using an algorithm. The problem with generating one is that it doesn’t always come out as you want, for example at the start of the project I would just use random numbers between 0 and 9 as the height for each point, but this created a very jaggy and unplayable terrain. The method I used instead is to use one that came with a Rastertek tutorial as it allowed for an interesting base for my algorithms to manipulate.

## Smoothing

Smoothing is done by taking the average height of a positions 8 neighbours. These would allow for jaggy terrain to have a smoother gradient between them. However, this process is parasitic in nature and if done enough it will cause the terrain to even out and become near flat. If a position is on the edge of a plane then only its applicable neighbours are taken into account as the top left point will only have 3 neighbours, otherwise the terrain will use coordinates that are not valid.

# Post Processing

In the application, there is a blur function that is started by pressing the B key. This blur function does not work as intended due to there being a problem with my ortho matrices in the camera class. I have tried for days and got help off of all my university classmates to try and fix this issue but for some reason it does not work. The problem is with my ortho matrix so it will not render on the screen for the user. To show that my blur does work I have used the view matrix that I know works. This obviously only displays the texture in the world but it does show off that my blur works as intended. To see this, you can restart the application and press the B key without moving, the blur texture will be in front of you and to the left.

# The Little Mechanic That Couldn’t

To begin this project, we were given a very wide scope with what we could create. I am very interested by cars, motorsport and everything of that nature. So, naturally, that’s where my mind went to and how I could create a project around it. I had the idea of creating a procedurally generated racetrack and displaying it on the plane. I took inspiration from a game called Dirt 4, in this game you never race the same track twice. Every stage is generated using there “My Stage” feature[[4]](#endnote-4), this is made using a complexity and length value.

To see how I could possibly achieve this I did a little research on how to achieve this desired result. Along my research, I came across this post[[5]](#endnote-5) which showcases a very useful method to achieving these results. The steps can be boiled down to these steps:

1. Generate Random Points.
2. Sort them into (anti)clockwise order
3. Create a Convex Hull of these points
4. Add midpoints and offset if needed
5. Separate points
6. Fix the angles if necessary

Included in the application is a file named racetrack.cpp which will show my attempt at creating the racetrack generator. I created a Convex hull of randomly generated points but no matter how many different approaches I took when separating the points, I couldn’t manage to get them to function properly. Even creating basic midpoints gave unreliable results that where undesirable. In the end I decided it best to focus my time on more complex methods and apply them to the terrain but in the future, I do plan on coming back to this and finishing the project as not only would it interesting to tackle, I think it could be very cool if done right.

# Conclusion

In conclusion I find that this module has been a very interesting experience, it has opened up a world of programming that I am very interested in its many applications. To improve my application, I would like to have spent more time on the racetrack idea, it could have even been placed into the terrain I have created using my other functions. A feature I had worked on was the Rastertek water tutorial[[6]](#endnote-6), sadly I could not get it working in time but if I had managed to, it would have been very easy to further improve on my texturing. It could have had another texture for when it is near the water level, for example this could be a sand texture to give even more realism to the application.

However, instead of talking about what ifs and what could have been, I am proud of the features I have created. The terrain can reliably showcase a number of different procedural generation techniques that I have learned over the course of this module.

Also, annoyingly, this module has given me more appreciation for the movie Frozen[[7]](#endnote-7)

1. Rastertek Website: <http://www.rastertek.com/> [↑](#endnote-ref-1)
2. Game Theorists: <https://www.youtube.com/watch?v=LzCOq2WFsSk> [↑](#endnote-ref-2)
3. The Fault Algorithm, Lighthouse: <http://www.lighthouse3d.com/opengl/terrain/index.php?fault> [↑](#endnote-ref-3)
4. Dirt 4, Codemaster: <https://www.racedepartment.com/threads/dirt-4-a-look-at-your-stage-track-generation-tool.133664/> [↑](#endnote-ref-4)
5. Gamasutra, Generating Procedural Racetracks: <https://www.gamasutra.com/blogs/GustavoMaciel/20131229/207833/Generating_Procedural_Racetracks.php> [↑](#endnote-ref-5)
6. Tutorial 16: Small Body Water, Rastertek: <http://www.rastertek.com/tertut16.html> [↑](#endnote-ref-6)
7. Frozen (2013), Disney: <https://www.imdb.com/title/tt2294629/> [↑](#endnote-ref-7)