**Paul Noonan T 00166981**

**Procedural Terrain Generation**

Table of Contents

[**Chapter 1: Introduction** 2](#_Toc469264037)

[**Chapter 2: Procedural Generation in Games** 2](#_Toc469264038)

[2.1 Introduction 2](#_Toc469264039)

[2.2 History of Procedural Generation 2](#_Toc469264040)

[**Chapter 3: Procedural Generation Algorithms** 5](#_Toc469264041)

[**3.1 Perlin Noise** 5](#_Toc469264042)

[**3.2 Diamond - Square** 8](#_Toc469264043)

[**3.3 Voronoi Diagram** 9](#_Toc469264044)

[**Chapter 4: Methodology & Design** 11](#_Toc469264045)

[**4.1 Introduction** 11](#_Toc469264046)

[**4.2 Research Findings** 11](#_Toc469264047)

[**4.2 Research Question** 11](#_Toc469264048)

[**4.3 Proposed Solution** 11](#_Toc469264049)

[**4.4 Design** 12](#_Toc469264050)

[4.4.1 Vision Doc 12](#_Toc469264051)

[4.4.2 Functional spec 13](#_Toc469264052)

[**Sprint One** 16](#_Toc469264053)

[Sprint One 16](#_Toc469264054)

[Task 1 16](#_Toc469264055)

[Task 2 16](#_Toc469264056)

[Task 3 17](#_Toc469264057)

[Task 4 17](#_Toc469264058)

[Images 18](#_Toc469264059)

[**Sprint Two** 19](#_Toc469264060)

[Sprint Two 19](#_Toc469264061)

[Task 1 19](#_Toc469264062)

[Task 2 20](#_Toc469264063)

[Task 3 22](#_Toc469264064)

[Bibliography 24](#_Toc469264065)

# **Chapter 1: Introduction**

--Leave until the end

# **Chapter 2: Procedural Generation in Games**

## 2.1 Introduction

Procedural generation is a method of creating data algorithmically rather than manually. The process has been used for many years in the gaming industry starting in the 1980’s. This introduces a randomness factor into the game that cannot be achieved any other way. This gives the player less predictable gameplay which increases “a game’s replayability since the content will be different on each play through.” (Lee, 2014)

Manually creating large landscapes, terrains and environments can take up a large amount of time. With the use of a procedural generation algorithm these large areas of land can be generated in moments which “create a game that has a near infinite number of unique areas to visit” (Williams, 2015). Images, animations, models and sounds are among the other things that are procedurally generated in games.

Games created these days take up a large amount of storage space. Procedurally generating parts of the game the file size of the games is drastically reduced while still keeping a high level of detail in the game worlds.

## 2.2 History of Procedural Generation

Procedural generation in games can be traced back to games as early as 1980. Due to the limitations of memory in computer systems in these years, they simply didn’t have enough memory to store large sets of information about game worlds like they do today. So instead of having small games with static content developers started to implement procedural generation techniques to keep memory usage low but still have large games. The Sentinel supposedly had 10,000 different levels stored in only 48 or 64 kilobytes.

In 1983 a dungeon crawler called “Rogue” or “Rogue: Exploring the Dungeons of Doom” was released. Rogue was a simple game where the player had to explore a dungeon and “fight your way through the Dungeons of Doom and return with the fabled Amulet of Yendor” (Epyx, 1985). All of the dungeons, monsters and treasure placements were all procedurally generated as the player traversed from one floor to the next. Rogue generated its dungeons one room at a time. Starting with one room placed randomly on a grid, it then creates another room on another grid. It then slides these grids together until the rooms fit tightly together. To procedurally create their content, they used nothing more than a random number generator.

In 1996 Blizzard released the first Diablo game, a game series that is still popular today. Diablo 1 took procedural generation a step further then just level generation and started to use it to generate in game items. Blizzard used Pseudo Random Numbers (PRN’s) to randomly assign an item a drop rate, modified, quality and stats. Generating items procedurally this way became hugely popular after Diablo introduced it, being used in some of the biggest games of the 21st century. The most well-known example of procedural item generation in a video game is held by Borderlands which holds a world record for it. “Borderlands, which uses a procedural system to randomly generate loot, holds the Guinness World Record for the most guns in a video game: a whopping 17.75 million.” (Yin-Poole, 2012)

When memory was no longer an issue most developers moved away from procedural generation as they could handcraft all of their game levels and not worry about storage space. Procedural level generation up to this point was primarily used for generating dungeons or closed environments. This was until  [Markus "Notch" Persson](https://en.wikipedia.org/wiki/Markus_Persson), a Swedish game developer, took on the challenge of creating an open world 3D game in which players could roam freely in any direction they choose. He created a game which is still in the top 10 most played games of today with over 40 million unique players each month, he created Minecraft. Released in 2009 into its aloha stage, featured entirely procedurally generated worlds that players could explore. Creating worlds based off a seed, there were endless possibilities of worlds to explore. Minecraft used Perlin Noise calculations to generate a realistic environment. “Mountains are always rocky and sprinkled with snow, while the low lands are typically full of grass and trees.” (Fingas, 15) Minecraft merged several layers of Perlin noise, one on top of the other, and then smoothed between them to create their unique landscape. Elements within the world are also procedurally generated such as forests, villages, vegetation and dungeons. Minecraft procedurally generates the world as the player moves through it, storing all information about the world the player has previously visited. As the world is so massive storing all that information does require a lot of memory in some cases. The game itself has incredibly low requirements in storage taking up only 120 megabytes’ compared to other games in its genre taking up 8 gigabytes minimum. If a player explored the entire Minecraft world it would require 858306884.765625 gigabytes or 0.8 of an Exabyte to store all of the information. However, to explore the entire world it has been estimated to take approximately 3 trillion hours . . . 4,390,586 lifetimes to explore the Minecraft map. You might think it’s a lot but if you take earth it is “500 million square kilometres, which is a lot. Until you learn that Minecraft has a surface area of Four. Billion. Square. Kilometres.” (Fallon, 2015)

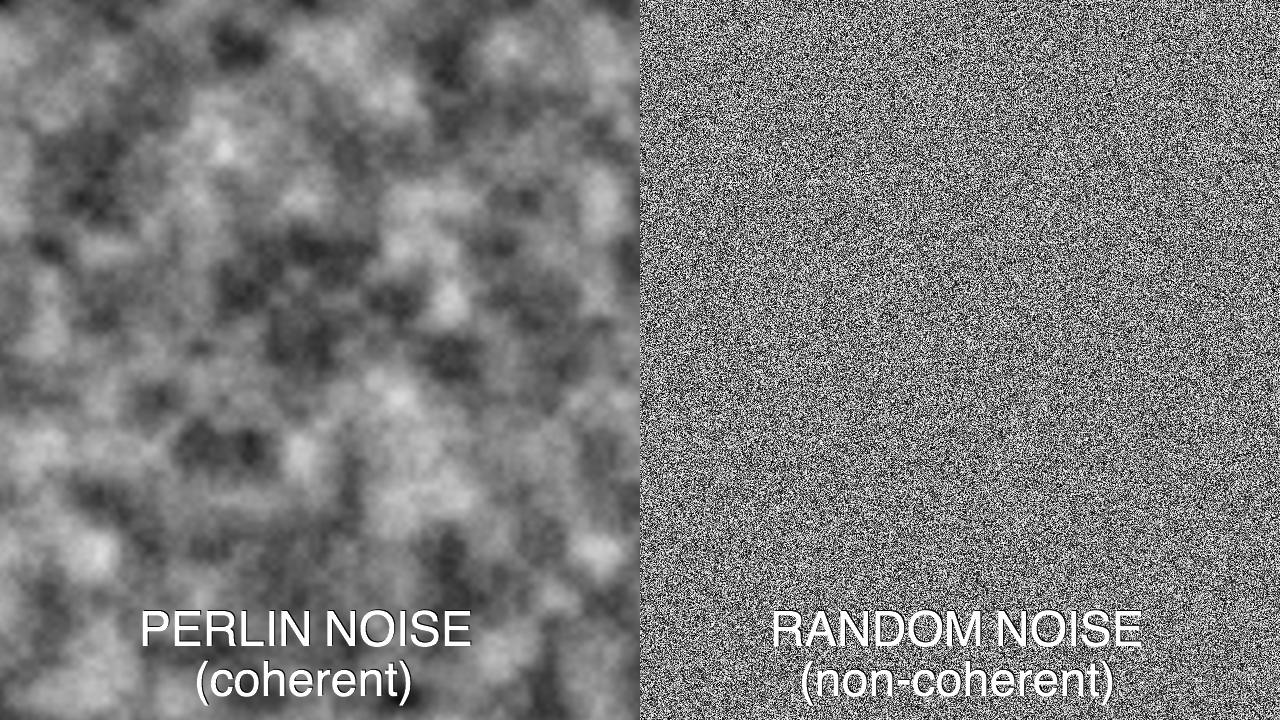
After Minecraft was released, because it was such a hugely popular games, a lot of companies tried to copy them, procedurally generating their own worlds. No game really stood out from the crowd until Hello Games announced a project they have been working on called “No Man’s Sky”. Released in 2016, No Mans Sky took procedural generation to a whole new level entirely. Rather than generating a single world they created an entire galaxy. “The game presents a traversable cosmos of unimaginable scale: 18 quintillion life-size planets” (Parkin, 2015). Each planet the player visits will have its own characteristics, its own flora and fauna. Hello Games procedurally generated almost everything in the game. Plants, Animals, Climates and Encounters. No Mans Sky took a huge step forward for procedural generation based games in the industry.

# **Chapter 3: Procedural Generation Algorithms**

There are many procedural generation algorithms used in games today. Many factors have to be considered when picking the best or most robust algorithm. One of the first discovered or created algorithms for procedural generation was “Perlin Noise”. Other algorithms used today include “Diamond-Square” and “Voronoi Diagram”. The above algorithms will be explained here.

## **3.1 Perlin Noise**

Perlin noise was developed in 1983 by Ken Perlin. In the original Tron movie in the 1980’s, Ken Perlin designed Perlin Noise to procedurally texture objects with CGI. In 1997 he received a Technical Achievement Award from the [Academy of Motion Picture Arts and Sciences](http://www.oscars.org/) for his work.

Noise is a completely random series of numbers. It can commonly be seen on TV screens when there is no signal or no channel is selected. The screen would become fuzzy and in a random pattern each pixel would flash a colour on the greyscale between black and white. This type of noise is the basis for any type of procedural generation. If you specified a range for the noise, let’s say 0 – 1, each pixel on the screen would be assigned a value and from that value they would obtain a colour. 1 would be white and 0 would be black, everything in between would be a mix between the two. This completely random noise can be organised by using Perlin noise. This organises the random noise into a pattern that is mathematically calculated.  (Marks, Perlin Noise -vs- Random Noise, 2012)

It is mainly used in games today to randomly procedurally generate content such as fire effects, water, clouds and terrain. It has developed over the years and now “It’s fast. It’s very little code to produce decent results.” (RedBlobGames, 2016)

The Unity Engine has a math formula “public static float PerlinNoise (float x, float y);” which generates a 2D PerlinNoise map. This, like in a lot of other game engines, hide the math from the developers allowing them to focus on manipulating the data the formula generates. When it comes to generating a landscape, the 2D PerlinNoise that is returned is usually treated as a height map for the generation. With very little manipulation of the data developers can, within minutes, generate simple hills or mountains.

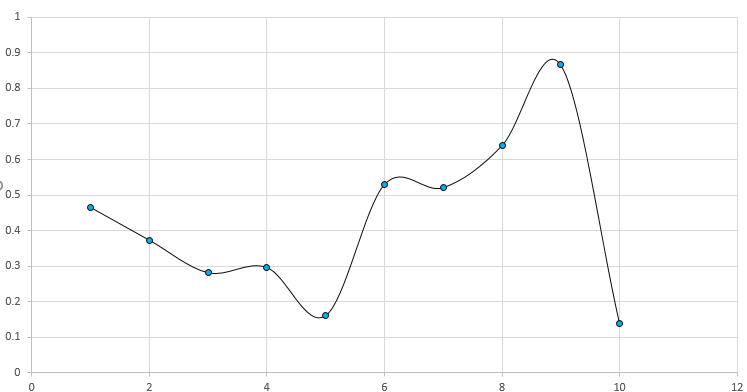
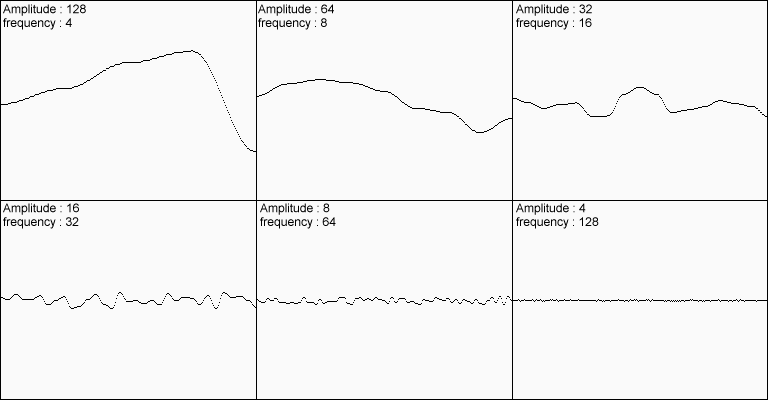
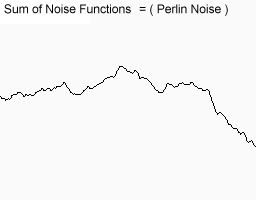


Chart 3.1: The above chart shows an example of the results of the unity PerlinNoise function.

This simple function does create a height map that can be applied to a terrain although it is very unrealistic looking. To create a more realistic terrain several of these Perlin noise functions should be layered on top of each other. Each one of these layer is called an octave. After each octave has been completed the amplitude and frequency will be changed for the next loop. Once all the values have been generated, the values at each position of the height map are combined. This should result in the creation of a more realistic looking terrain. Here is an example of this combination:

Image 3.2: Six Octaves of Perlin noise with Frequency being doubled and Amplitude being halved with each iteration.

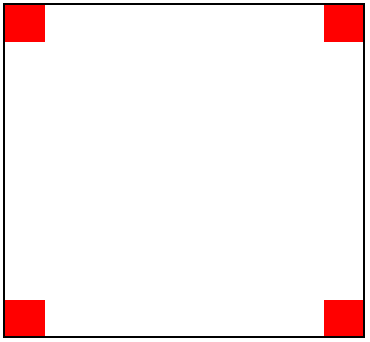
  
Image 3.3: The Six Octaves of Perlin Noise from Image 3.2 combined into one.

The graph in image 3.3 shows a much more realistic terrain as the mountainous shape has rugged areas added to it. This process can be repeated many times to add higher levels of detail. Games such as Minecraft and Terraria that have advanced procedural generation layer several of these Perlin Noise functions on top of each other.

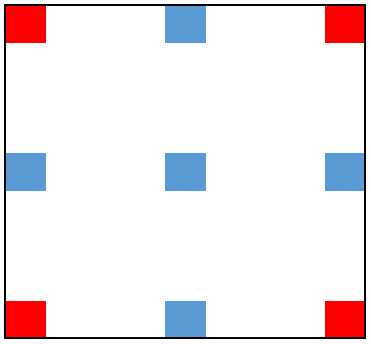
## **3.2 Diamond - Square**

The Diamond-Square algorithm is a method for generating height maps. It was first introduced in 1982 at SIGGRAPH by  [Fournier](https://en.wikipedia.org/wiki/Alain_Fournier), [Fussell](https://en.wikipedia.org/w/index.php?title=Don_Fussell&action=edit&redlink=1) and [Carpenter](https://en.wikipedia.org/wiki/Loren_Carpenter). The Diamond-Square is a refinement of the midpoint displacement algorithm and is broken into two parts. The ‘Square’ generates a boxlike landscape made up of easily identified squares while the ‘diamond’ improves these results as including both diamond and square shapes into the generation. It requires a lot less manipulation compared to Perlin noise which makes it “it's simple and produces something very visual” (Foster, 2015)

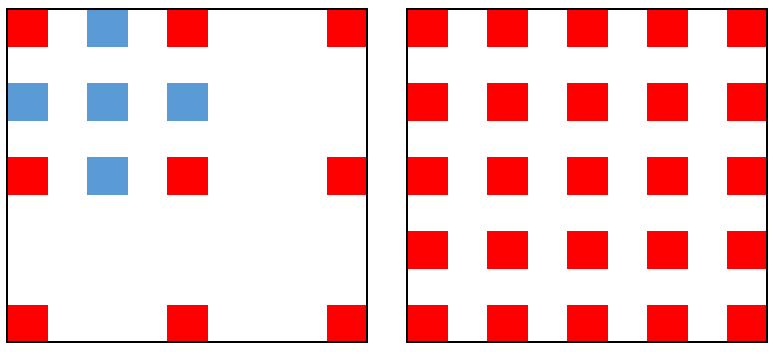
The algorithm starts with a 2D array of a width and height 2ⁿ + 1. The four corners points   
of the array must be set to initial values for the diamond and square steps to work. Each corner can have a set height value attached to it, this is called the square step. The height these corners have can be predefined or can also be procedurally generated.

  
Image 3.4: This the start of the Diamond-Square algorithm, displaying four corners.

Next phase the average of the four corner points is calculated and a random amount of noise is added or subtracted, this is the diamond step. The noise value is a randomly generated value between the minimum and maximum height of the height map.

  
Image 3.5: This is the ‘diamond step’ of the algorithm. The blue represents the new points defined by the algorithm.

The diamond and the square steps are repeated over and over until the points are as close as is defined by the algorithm. The close the points are the more detail will be evident in the final terrain.

  
Image 3.6: On the left is the next iteration of the diamond step. On the right is with all the diamond and square steps complete

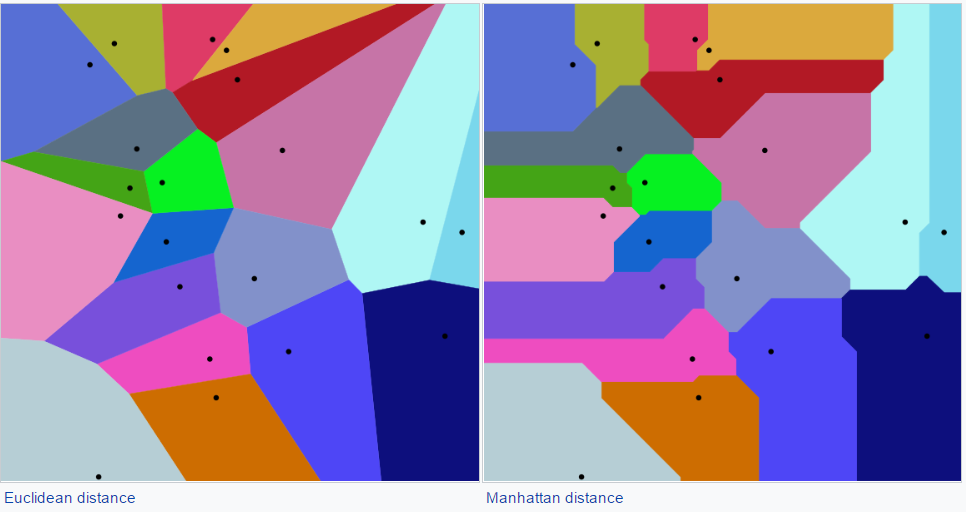
Once all of the diamond and square steps have been completed the terrain is generated from the points. A plane is started from the first point and is connected to the closest 3 points. This process is repeated until all of the points have been connected with planes. This creates the final terrain.

This algorithm is very common today in procedural generation high quality realistic terrain. Terragen, a product from Planetside Software, along with other applications use the diamond- square algorithm because it’s so simple to implement while giving realistic results. The biggest problem is because the terrain is generated between points that are spaced apart it cannot generate overhangs. This can result in very unrealistic terrain in certain environments where overhangs and back sloping terrain is common such as a cliff face. The data can be manipulated to produce these features but at its core it is impossible for the algorithm to generate them naturally.

## **3.3 Voronoi Diagram**

A Voronoi Diagram is the partitioning of a plane into regions based on distance to points in a specific section of the plane. The algorithm is also referred to as Voronoi tessellation, a Voronoi decomposition, a Voronoi partition, or a Dirichlet tessellation. “Even though Voronoi diagrams were first investigated by René Descartes in the 17th century and applied by Dirichlet when exploring quadratic forms, the diagrams were named after Georgy Voronoi” (Wilson, n.d.).

The algorithm is very simple at its lower levels and really only has two steps. The first step of the algorithm places control points, also known as seeds or sites, throughout the plane. These points are randomly placed but they can have predefined locations. Usually each point also has a colour related to it so it can be told apart from the other points easily. Secondly the algorithm goes through every pixel and colours it the same as the colour related to its closest point. This will create polygons of colour surrounding each point. The distance to each point is calculated in one of two ways, both producing different end results. The Euclidean distance:   
or the Manhattan distance: .

  
Image 3.7: This image s how the Voronoi diagram produced by Euclidean and Manhattan distance formulas.

Each of the areas seen in image 3.7 can be assigned a height, separating the areas by elevation. This creates platforms that have clearly defined borders. Each of these borders can be sloped to combine with its surrounding platforms to create a seamless terrain. This results in a terrain that is split up into many different areas and can be used in a variety of ways. The most common use for this type of Voronoi diagram in games is to create biome areas on a landscape. This makes it very easy to distinguish one biome area from another. For example, Blue would be water and green would be grass while pink would be mountains and red desert. Voronoi diagrams are applied like this in different aspects of life to try to establish borders such as:

* Zoology -- Model and analyse the territories of animals
* Robotics -- Path planning in the presence of obstacles
* Geography -- Analysing patterns of urban settlements
* Statistics and Data Analysis -- Analyse statistical clustering  
  (Drysdale, 1993)

# **Chapter 4: Methodology & Design**

## **4.1 Introduction**

In the chapter 2 and 3 procedural generation and algorithms were researched and evaluated. In this chapter a new approach to terrain generation will be explained. This Includes the algorithms and techniques involved in generating geographical features such as mountains, cliffs, fields and plains.

## **4.2 Research Findings**

The initial research focused on procedural generation. This involved two different elements, firstly identifying different procedural generation algorithms and secondly how they were applied to the generation of terrain. To procedurally generate terrain, the algorithms data must be heavily manipulated to construct a realistic terrain with many geographical features. Perlin noise excels at generating large sets of data that can easily manipulated to form landmasses with big physical features such as mountains, valleys and cliffs. However, it is weak when it comes to generating features such as borders, rivers and biomes. This is where the Voronoi diagram is most powerful as it creates sections that can be used as biomes. The borders of the diagrams can also be used as rivers as it creates a realistic flow pattern if all the sharp corners are smoothed out.

## **4.2 Research Question**

* Can procedurally generated terrain look realistic.
* What algorithms are used in games to procedurally generate terrain?
* What are the advantages and disadvantages of procedurally generating terrain?

## **4.3 Proposed Solution**

The purpose of this project is to procedurally generate a voxel based terrain. As the terrain will be built out of voxels, the most suitable algorithm is Perlin noise. This is due to the fact that Perlin noise results can be pre-calculated at each point rather then left up to complete randomness. This gives a large amount of control over the end result of the terrain and allows for greater manipulation of the data to achieve a realistic result. Under the Perlin noise terrain a Voronoi diagram will be defining all borders for biomes and rivers. These two algorithms combined with the appropriate parameters will result in a realistic terrain full of geographical features, climates and biomes.

## **4.4 Design**

4.4.1 Vision Doc4.4.1.1 Project Description

This project will allow the user to procedurally generate a terrain that will go on infinitely on the x and z axis. The terrain will be made out of voxels and in my case the voxels will be cubes. The terrain will be textured and each cube can be interacted with individually.

#### 4.4.1.2 Deliverables / Outcome

The end result will be a terrain that a player can move around on and explore infinitely in any direction. The terrain will be able to be changed by the player by adding or removing blocks from the terrain.

#### 4.4.1.3 Functionality and Technology

My whole project will be created inside of unity with C# being my language of choice to code in. If I create the textures myself I will be using GIMP 2 to make a texture map.

#### 4.4.1.4 Key Features / Components

Moscow Method: M = Must have, S = Should have, C = Could have, W = Wont have

001 Generating a noise map – M

002 Generate a chunk of landscape from the noise map – M

003 Generating a texture for each block – M

004 Combining all of the block textures into one chunk texture - M

005 Allow blocks to be added manually – C

006 Allow blocks to be removed manually - C

#### 4.4.1.5 Data

The data I will be holding and saving will the information about each chunk which will be saved locally. Each chunk will have a position, number of blocks in the chunk and a texture that all will have to be saved.

### 4.4.2 Functional spec

4.4.2.1 Map Generation

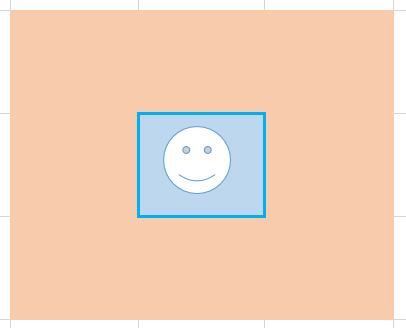
As my project is not directly interacted with by the player most of the work is done by the system. The player will have a choice about some variables the system uses for generation at the start of the game but outside that generation is all handled by the system. This way the player doesn’t have to worry about it and can just enjoy playing the game itself.

To get a final result all of these processes are necessary and must be completed in this sequence.

When the player presses play on the game the system will find the position the player is located in 3d world space. When it has the location of the player it will begin generating the terrain.

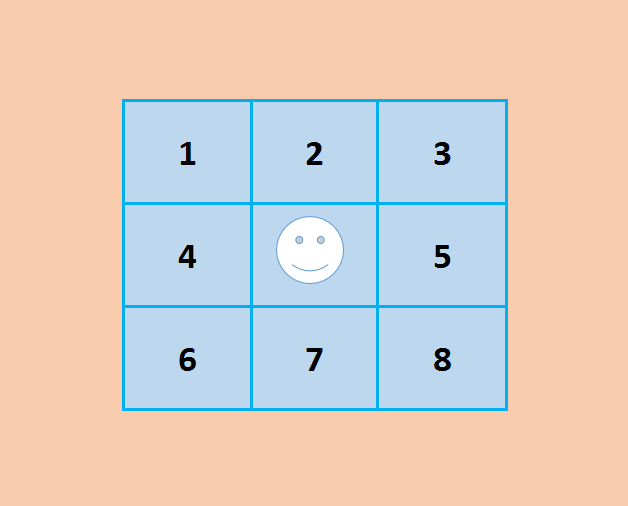
It will start the first chunk of the terrain at the player’s position. When that chunk is complete it will start to generate the chunks around the player in stages until the range limit is reached. This range limit is set by the player to allow them to choose how far they would like to see into the terrain.

Stage 1



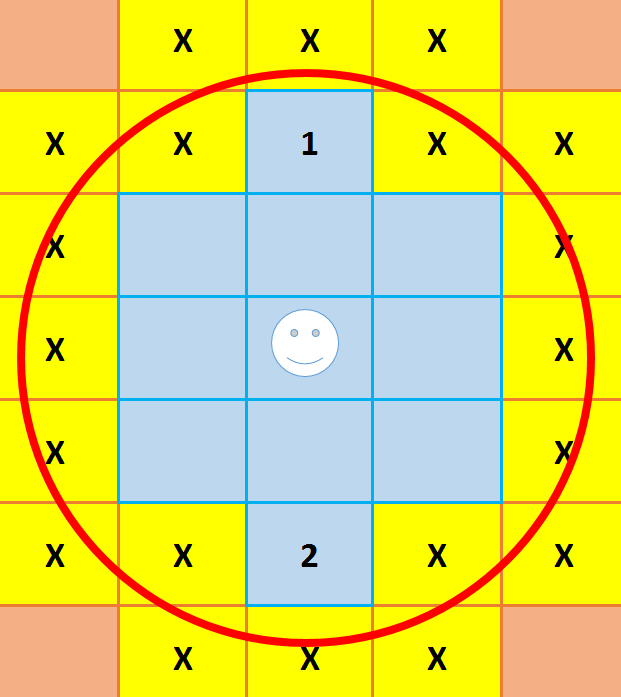
The chunk the player spawned on is generated.

Stage 2



The chunks are generated around the player’s position from 1 – 8 as the system will generate the chunks from left to right always starting at the highest value which is the top left and working to the smallest value in the bottom right.

Stage 3



It generates as far as the players range (red circle) and will not generate a chunk unless it is fully within the circle.

# **Sprint One**

|  |  |  |
| --- | --- | --- |
| **Sprint Number** | **Start Date** | **Finish Date** |
| 1 | 08 / 09 / 2016 | 22 / 09 / 2016 |

|  |  |  |
| --- | --- | --- |
| **Task Number** | **Details** | **Status** |
| 1 | Create a simple Perlin noise function | Complete |
| 2 | Use the noise map to instantiate cubes at noise values | Complete |
| 3 | Colour the cubes with the noise map values | Complete |
| 4 | Apply a colour to each cube depending on noise height | Complete |

# Sprint One

Sprint one lasted for two weeks from the 8th September until the 22nd September. The aim of this sprint was to create a simple noise map. After generating it apply it to a plane and texture it. If that worked then create a grid of cubes and apply the texture to them.

## Task 1

Task one was very simple as the unity engine has its own Mathf.PerlinNoise function. It requires two floats and returns a 2D Perlin Noise. This function was simple enough for me to work with at this stage and had all the functionality I required. The two floats I passed into the function was the width and depth of the map I will be generating, in this case it was 16 and 16 so the function was Mathf.PerlinNoise(16,16);. This returned a list of values between 0 and 1 which I used in task 2.

## Task 2

In this task I wanted to create a grid of cubes equal to the size of the map. I created a class named MapDisplay which would generate this grid of cubes. I created a function called “DrawMap” which required 3 arguments:

* Map Width
* Map Height
* Noise Map

I created a forloop that would loop from 0 to Map Width.

for (int x = 0; x < mapWidth; x++)

{

and then an inner forloop looping from 0 to Map Height.

for (int y = 0; y < mapHeight; y++)

{

At each stage of this loop I would then instantiate a cube at the vector3(mapWidth, 0, mapHeight).

This created a nice square map of cubes which I then used in task 3 to try and apply the noise value to them with colour.

## Task 3

At this stage I had a 2D Perlin noise map and a grid of cubes so in this task I wanted to apply a texture to each cube depending on the noise value at each cubes position. This was a lot easier then I originally thought it would be. As I already had two forloop values specifying each cubes position, as the noiseMap is a 2D float array I could easily get each cubes position in the noiseMap by specifying noiseMap [x, y];. This would return the noiseMap value at each cube position. From here I just grabbed each cubes renderer and created a colour lerp with:

Color.Lerp(Color.black, Color.white, noiseMap [x, y]);

This created a colour between white and black that I applied to each cube. This would visually represent the noise value of each cube.

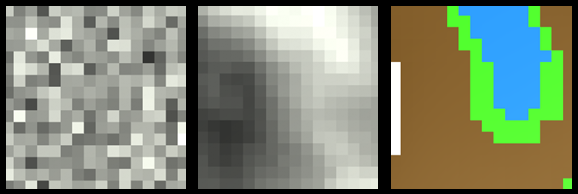
## Task 4

I wanted to take it from black and white to colour. I made a function that applied colour based on height. I made four ‘regions’ that the blocks would be placed in based on their height.

* Water from 0 – 0.2
* Grass from 0.2 – 0.4
* Mountains from 0.4 – 0.8
* Snow tops from 0.8 – 1

Each region had a colour related to it and so it gave the landscape a vibrant appearance. However, it was very obvious where grass stopped and mountains began as it was a horizontal line through the entire map where the colours would change.

## Images

Below is the progression of the development in Sprint one.   
On the left is a noise map displayed on the cubes. It is clear how random the values are without any sort of coherence.   
In the middle is the same noise map generated wither Perlin noise. It is clear that the noise is a lot more coherent, following a pattern nearly.  
On the right is an image of colour applied to the noise. Blue being water and the white is the mountain peak.

# **Sprint Two**

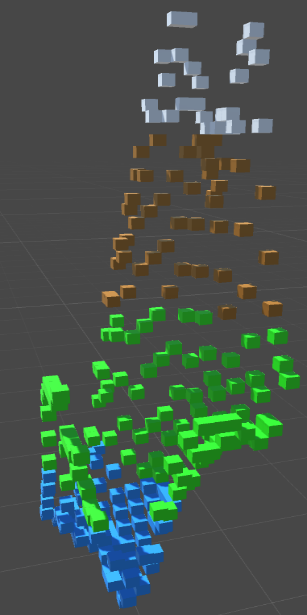
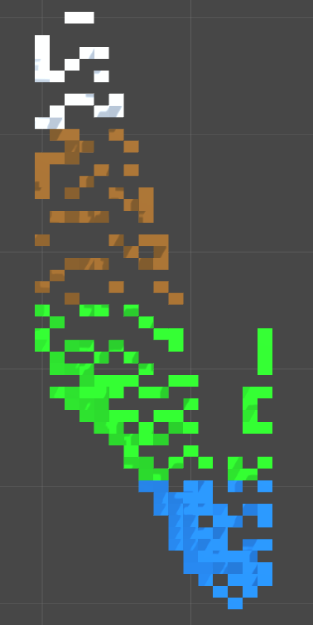
|  |  |  |
| --- | --- | --- |
| **Sprint Number** | **Start Date** | **Finish Date** |
| 2 | 29 / 09 / 2016 | 20 / 10 / 2016 |

|  |  |  |
| --- | --- | --- |
| **Task Number** | **Details** | **Status** |
| 1 | Apply height to the chunks | Complete |
| 2 | Add detail to the Perlin noise | Complete |
| 3 | Generate two chunks side by side | Complete |

# Sprint Two

Sprint one lasted for two weeks from the 29th September until the 20th October. The aim of this sprint was to create a develop the noise generation function further, adding more levels of detail into the noise map. And then finally generate multiple chunks at the same time.

## Task 1

To apply height to each cube of my chunk I took the noise value of each cube and multiplied it by a value. I set my value to 80 and so the maximum height possible in each chunk is 80 as noise is generated between 0 – 1 (80 x 1 = 80). Because my terrain is based on cubes and I wanted each cube to be set on levels I didn’t want this value to be a float and have cubes overlapping height wise. To fix this issue I rounded each number to an integer and then applied this new value to the cube.   
var yValue = Mathf.RoundToInt(80 \* noiseMap [x, y]);  
var newVec3 = new Vector3(transform.position.x, yValue, transform.position.z);  
This code calculates a value and rounds it to the nearest int. I think take this  
value and apply it in the Y-axis of the cube. The results can be seen in the following images in both 2D and 3D. The gap between each block is very obvious and is an issue I planned on dealing with later on in the project.

## Task 2

The main objective of this task was to increase the amount of detail in my Perlin Noise. Up to now I’ve only had the size of a chunk as well as the scale of the noise as variables. To add in more detail to the noiseMap I created five more variables:

**Octaves**: -  
 this is the number of layers of noise we want to layer

**Lacunarity**: -  
 Controls the increase in detail through each octave

**Persistence**: -   
 This controls the impact of each octave on the noise, the increase in octave, the   
 decrease in effect it has.

**Frequency**: -  
 The number of cycles it completes in a given interval  
 Frequency = Lacunarity \* octave

**Amplitude**: -   
 The height from the mean, or rest, value of the function to its maximum or   
 minimum.   
 Amplitude = Persistence \* octave

To apply these variables I first created a for loop looping through each octave. At the end of each loop I would change my amplitude and frequency values. This ensures that each subsequent octave will be more and more detailed but have less effect on the end result.

for (int i = 0; i < octaves; i++)

{  
 float sampleX = (x + octaveOffsets[i].x) / scale \* frequency;

float sampleY = (y + octaveOffsets[i].y) / scale \* frequency;

float perlinNoiseValue = Mathf.PerlinNoise(sampleX, sampleY) \* 2 - 1;

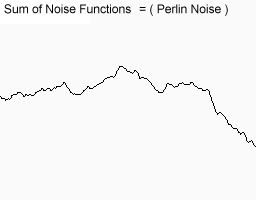
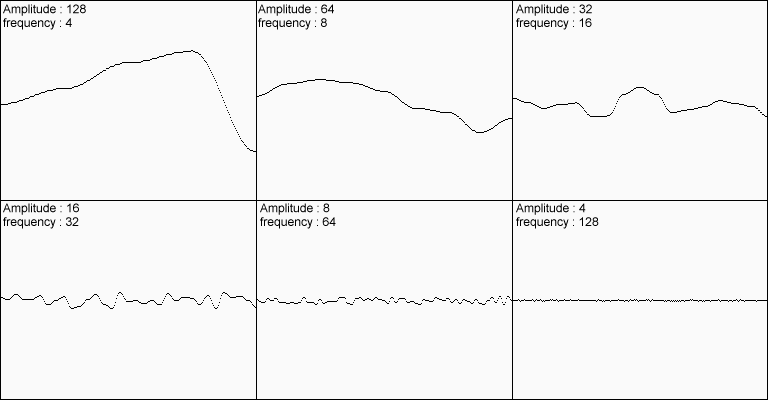
noiseHeight += perlinNoiseValue \* amplitude;

amplitude \*= persistence;

frequency \*= Lacunarity;

}

This code created a noiseMap with increased detail just as I wanted. Below is an example of how the code works. I set Lacunarity = 2 and persistence = 0.5. Through each octave it is clear that the frequency is increasing and the amplitude is decreasing.   
  
The first image is the noiseMap that I was generating before as I only had one octave. It is a very unrealistic terrain. The new noiseMap is all six of the below octaves combined as shown in the bottom image. This is a more realistic outline of a landscape.

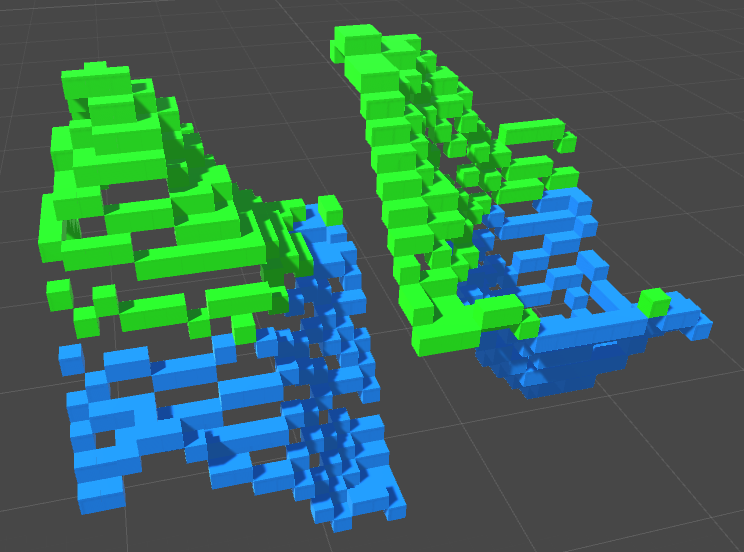


## Task 3

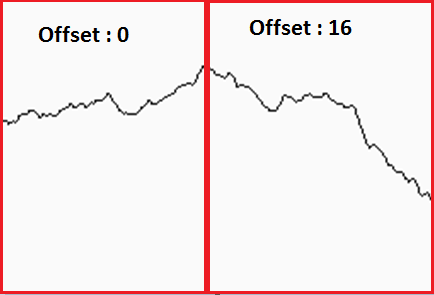
This was my first attempt at generating two chunks next to each other. Before this I was generating a single chunk and editing it.

My first step I just tried to generate twice to generate two chunks instead of one and the first issue was obvious. Both chunks generated right on top of each other so I needed to create an offset. I created a vector2 offset which was the size of a chunk.   
A chunk is 16 x 16 so I know I have to offset each chunk by 16 in the given direction so they are built side by side rather than on top of each other.

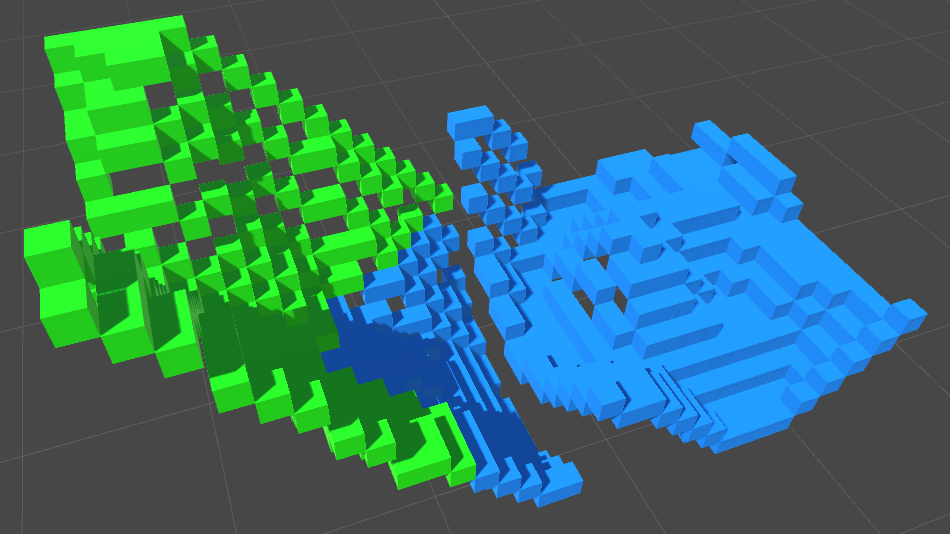
After I created this offset I again generated these two chunks right next to each other and the next issue popped up. Even though I was offsetting the chunk in position chunks seemed to have absolutely no relation to each other. I realised this was due to the fact that the noiseMap was still being generated completely randomly rather than each chunk being offset along the curve.



I added this offset into the noise generating function, adding it to both the X and Y of the curve so that the chunks would be coherent. An example of this offset can be seen in the below image.



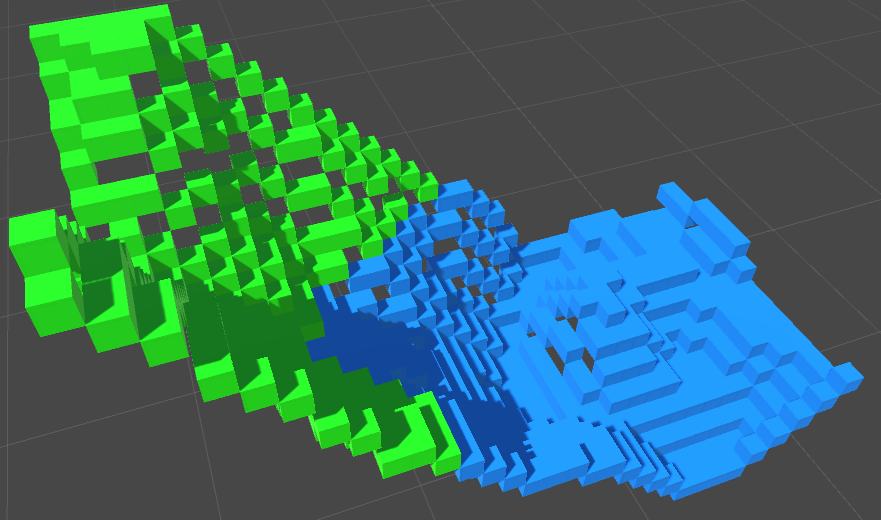
This did make the chunks look like they belonged together however there was still a noticeable gap between each chunk on the y-axis. One chunk would generate slightly above or below the previous chunk.



After doing a lot of code testing I found this issue came from when I normalized the noiseMap I was generating between a min and max which changed from chunk to chunk. I found a work around by debugging several chunks to find their max noise height and min noise height. I found the average of ten random chunks and found that the average minimum was -1.188136 and the maximum was 0.9655656. During the normalization process I then capped the the min and max to these values.

noiseMap[x, y] = Mathf.InverseLerp(-1.188136f, 0.9655656f, noiseMap[x, y]);

This got rid of that noticeable gap and now the chunks connect together seamlessly.



# Bibliography

Alexander, A. J. (n.d.). *What is a "Function Point"?* Retrieved from Alvin Alexander: http://alvinalexander.com/FunctionPoints/node3.shtml

*CMSC 645, Section 6060, Advanced Software Engineering, Summer 2002*. (2002). Retrieved from CSEE: http://www.csee.umbc.edu/~mgrass2/cmsc645/function\_point.html

Drysdale, S. (1993, July 19). *Voronoi Diagrams: Applications from Archaology to Zoology*. Retrieved from ICS: https://www.ics.uci.edu/~eppstein/gina/scot.drysdale.html

Epyx, I. (1985). *Rogue*. Retrieved from roguelikedevelopment: http://www.roguelikedevelopment.org/archive/files/misc/EpyxRogueDOSManual/manual.htm

Fallon, S. (2015, May 27). *How Big is Minecraft? Really, Really, Really Big*. Retrieved from Wired: https://www.wired.com/2015/05/data-effect-minecraft/

Fingas, J. (15, March 03). *Here's how 'Minecraft' creates its gigantic worlds*. Retrieved from Engadget: https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/

Foster, J. (2015, January 19). *The Diamond Square Algorithm*. Retrieved from fatvat: http://www.fatvat.co.uk/2015/01/the-diamond-square-algorithm.html

*function point modeler*. (n.d.). Retrieved from function point modeler: http://www.functionpointmodeler.com/fpm-infocenter/index.jsp?topic=%2Fcom.functionpointmodeler.fpm.help%2Fditafiles%2Fconcepts%2Fcon-32.html

Lee, J. (2014, November 26). *How Procedural Generation Took Over The Gaming Industry*. Retrieved from makeuseof: http://www.makeuseof.com/tag/procedural-generation-took-gaming-industry/

Longstreet, D. H. (2012, February). *Fundamentals of FPA*. Retrieved from Software Metrics: http://www.softwaremetrics.com/fpafund.htm

Marks, D. (2012, January 19). *Perlin Noise -vs- Random Noise*. Retrieved from Tumblr: http://digitalmark.tumblr.com/post/16118899843/perlin-noise-vs-random-noise

Marks, D. (2012, January 19). *tumblr*. Retrieved from tumblr: http://digitalmark.tumblr.com/post/16118899843/perlin-noise-vs-random-noise

Parkin, S. (2015, July 12). *No Man’s Sky: the game where you can explore 18 quintillion planets*. Retrieved from The Guardian: https://www.theguardian.com/technology/2015/jul/12/no-mans-sky-18-quintillion-planets-hello-games

RedBlobGames. (2016, September 28). *Making maps with noise functions*. Retrieved from RedBlobGames: http://www.redblobgames.com/maps/terrain-from-noise/

Williams, J. (2015, July 26). *Procedural Generation As The Future*. Retrieved from wccftech: http://wccftech.com/article/procedural-generation-future/

Wilson, J. (n.d.). *Voronoi Diagrams*. Retrieved from Jwilson: http://jwilson.coe.uga.edu/EMAT6680Fa08/Kuzle/Math%20in%20Context/Voronoi%20diagrams.html

Yin-Poole, W. (2012, July 16). *How many weapons are in Borderlands 2?* Retrieved from EuroGamer: http://www.eurogamer.net/articles/2012-07-16-how-many-weapons-are-in-borderlands-2