Procedurally Generated 3D Worlds for Video Games

By Callum Wilson

Supervisor: Dr. David Cairns

7th April 2017

Dissertation submitted in partial fulfilment for the degree of
Bachelor of Science in Computing Science & Mathematics

Division of Computing Science and Mathematics
University of Stirling

Abstract

During the development of video games, much time and money is spent creating the 3D worlds which the player explores. This creation is typically done manually by artists and designers. An alternative to this is procedural generation: generating data with algorithms.

This project presents procedural generation and some typical techniques to generate data. Then, several modern examples of projects using procedural generation are presented and evaluated.

This project aimed to create a generator capable of generating realistic 3D worlds comprising of: terrain, rivers, lakes, oceans, vegetation, and buildings. This project created a dynamically loading terrain system and a terrain generator which generated data based upon tectonic plate boundaries. Also, Headway was made into creating a rain-simulator which used an agent based method to determine the placement of rivers and lakes.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

I certify that this dissertation reports original work by me during my University project except for the following:

- The noise generation functions used are from *LibNoise for .NET* [1]
- The terrain chunking system is based upon a video tutorial series by Sebastian Lague [2]
- The rendering engine used was Unity 3D [3]

Signature **Date**

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. David Cairns, for his patience and help during this project. It would not have been possible without you.

I would like to thank my parents and my little brother, and the rest of my family and friends, for all their hard work and support. I love you all.

Finally, I would like to thank all the staff who have taught and/or helped me during my time here at the University of Stirling. You have all given me something I can never repay.

Table of Contents

List of Figures

1 Introduction

1.1 The Problem

In video game development, a lot of time and money is spent creating 3D worlds for the player to experience. These worlds, or maps, are usually created by hand by artists. This process can take weeks, or even months depending on the scale and complexity of the map. Not only that but this also relies upon human imagination, something which has its limits. This project will explore procedural generation as a potential solution to these problems.

1.2 What is Procedural Generation?

Procedural generation is the process of generating data algorithmically rather than manually. In video games, procedural generation can be used to generate: textures (2D images that are mapped onto 3D game objects), music, 3D object models, 3D terrain meshes, and more. This is sometimes referred to as 'Procedural *Content* Generation'.

Generators are usually deterministic: using a set of parameters, a generator will create content specific to those parameters. Running the generator with those parameters will always produce the same output.

Procedural generation can be online or offline. Online generation would be done during run-time on the user's (player's) machine. Offline generation would be done during development on the developer's machines. There can be hybrids of the two, for example the main map data such as terrain information could be generated offline, then the decoration of the map (placement of vegetation, for example) could be done online during play or before play whilst the game loads.

1.2.1 Issues with Procedural Generation in Video Games

When designers create worlds for video games they must consider the rules of the game, otherwise, the worlds can be boring, or in worst cases game-breaking. The obvious examples would be: creating an intractable dungeon, or an unbalanced multiplayer map (where one team has an advantage because of their placement in the map). When using procedural generation, it is especially important to keep this in mind.

Consider a video game where the player is a character in a procedurally generated fantasy world. In this fantasy world, there are many villages and towns, each with their own simple economies. The player starts in one of the villages, except the player's village is trapped by unfavourable terrain (such as mountains and woods). Being a low-level character it may be difficult for the player to escape this village and explore the rest of the world. The player must

therefore spend an inordinate amount of time trying to overcome the first area of the game. During this time, the player may give up entirely and quite the game.

This is a simplified version actual games, but it represents a major issue. How can developers make sure that the world of a video game and the mechanics of that video game work together if the world is not known?

It is not always feasible to test generated data for these issues, and in some cases, impossible. Nor is it always possible to test for 'fun-ness'. A developer might think to get a team of testers to keep replaying generated maps, but this is certainly no guarantee for the other possible outputs of a generator.

Some of these issues may not be an issue at all, consider Borderlands, a first-person-shooter (FPS) game with procedurally generated weapons. Not all weapons are good, or even useful and some are downright awful. However, because the player could choose whether they picked the weapons up or not, it was not an issue. These weapons also fit into the games story, and given the game's satirical and wacky themes, they fit quite nicely [4].

1.3 Why use Procedural Generation?

There are many benefits to using procedural generation. This project focuses on saving man-hours, which would be spent designing worlds, by replacing world designers with algorithms and allowing for automatic world creation. Considering the amount of time that goes into creating these worlds, it is no meagre thing.

Designers need not be taken out of the picture completely. Procedural generation can be used in tandem with designers by taking the role of inspiration. Worlds can be created automatically, and then designers can modify these worlds however they wish. This method is especially useful because not only does it reduce total man-hours but also avoids some of the common pitfalls of unplayability. This is avoided because the designer effectively checks and rectifies any issues which the world might have.

Procedural generation does not have to be used to generate everything in a world. For example: consider *SpeedTree*. "*SpeedTree* is a powerful toolkit used to create 3D animated plants and trees for games, animations, visual effects and more." [5]. This toolkit provides generators to generate vegetation in videos games, something which will likely not affect gameplay but will provide a pleasing aesthetic. It would be tedious for a designer to hand craft all that vegetation, especially if the world area is large. Note, *SpeedTree* is not only used in video games but also in CGI for films.

Procedural generation can also act as a form of compression when the algorithms are deterministic. This compression is useful not only during transmission of the game (via network

or physical media) but also when the game is installed on the player's machine. Rather than transmitting/storing all the data, you only need to transmit/store the algorithm and the input parameters.

1.4 Scope and Objectives

This project aims to create a program capable of generating finite 3D maps, using multiple procedural generation techniques, from user-defined parameters e.g. world size and a seed value. These maps will consist of terrain, water (rivers, lakes, oceans), vegetation decoration (trees and plants), and cities (possibly with road between them). These features will be generated so that they conform to real-world logic and are, therefore, somewhat realistic.

Following this, the project aims to provide a learning experience for game developers who are implementing their own procedural generators. It will also provide concrete examples of how these techniques can be used and drawbacks associated with them.

1.5 Achievements

This project has created a program capable of generating terrain based upon layered noise and a simple tectonic plate boundary system. This project tried multiple different processes which may be of use to future developers when creating their own generators.

This project also successfully implemented a dynamic level of detail (LOD) chunked terrain system to load and display the generated maps efficiently.

1.6 Overview of Dissertation

In section 2, a brief introduction into procedural generation in the scope of video games will be given. A basic technique to generate terrain data will be described as an example. Then, several games, which use procedural generation, will be described, analysed, and have conclusions drawn from them.

In section 3, multiple prototypes which were developed for this project will be described and assessed. Along with these, reasoning will be given for why they were initially designed. The final application and its results will be presented, as well as an overview of how it was implemented.

In section 4, conclusions will be drawn on the project and procedural generation in video games. More specifically, this will include a summary of the project, an evaluation of the created application, and possible ideas for future work.

2 State-of-The-Art

2.1 Introduction

There are many video games that use procedural generation, early examples include *Rogue* (1980) [6], which generates simple 2D dungeons constructed of ASCII characters, and *Elite* (1984) [7], which generates simple universes. More modern examples include *Borderlands* (2009) [8], a FPS game which uses procedural generation to generate enemies and equipment.

In this section several modern video games will be discussed that use procedural generation to generate complex 3D maps, some of which use it to generate galaxy's worth of starts and planets. An example of how to generate terrain procedurally will also be discussed.

2.2 Using Procedural Generation to Generate Terrain

There are a wide range of procedural generation techniques, to aid in the understanding of the section ahead, some terminology/common tools will be defined and then a simple technique of generating terrain data will be shown.

2.2.1 Data Generation

There are many ways to generate data procedurally. In this section, several techniques and common terms will be defined.

2.2.1.1 Seed

A seed is typically a 32-bit integer value which is used by generators, in some way, to generate data. The idea of a seed value is important in procedural generation because the same data is outputted for the same seed. This allows for a kind of compression of data.

2.2.1.2 Noise

In this context, noise refers to a sequence of pseudo-random numbers. A noise function, therefore, would be a function which given an input, returns a pseudo-random number. The functions are classed as pseudo-random because if they are passed the same parameters, they give the same output. These functions usually use a seed value to ensure this.

Commonly, noise is used to generate textures and comes in many forms. Perhaps the most useful type of noise is coherent noise. Coherent noise is a 'smoother version of noise'. It is smooth in the sense that a slight change in the input value will result in a small change in the output value. A significant change in input, however, will result in a random change in the output value. When graphed, this looks like a smoother curve than non-coherent noise.

Noise generators can have many settings for specifying characteristics of the data they output. Some of these include: frequency, octave count, and persistence. However, not all will be present for different noise functions.

2.2.1.3 Fractal Methods

The term ‘fractal’ refers to a curve or shape where each part exhibits the same statistical character as the whole. In layman’s terms, it means that something looks similar whether you look at the whole thing, or small section. See Figure 1 for some examples of fractal shapes. In video games, fractals can be useful to generate terrain, specifically mountain ranges, and trees. Both of which exhibit fractal characteristics. It is important to note that some examples of noise are fractal. One such type of noise is Perlin noise.

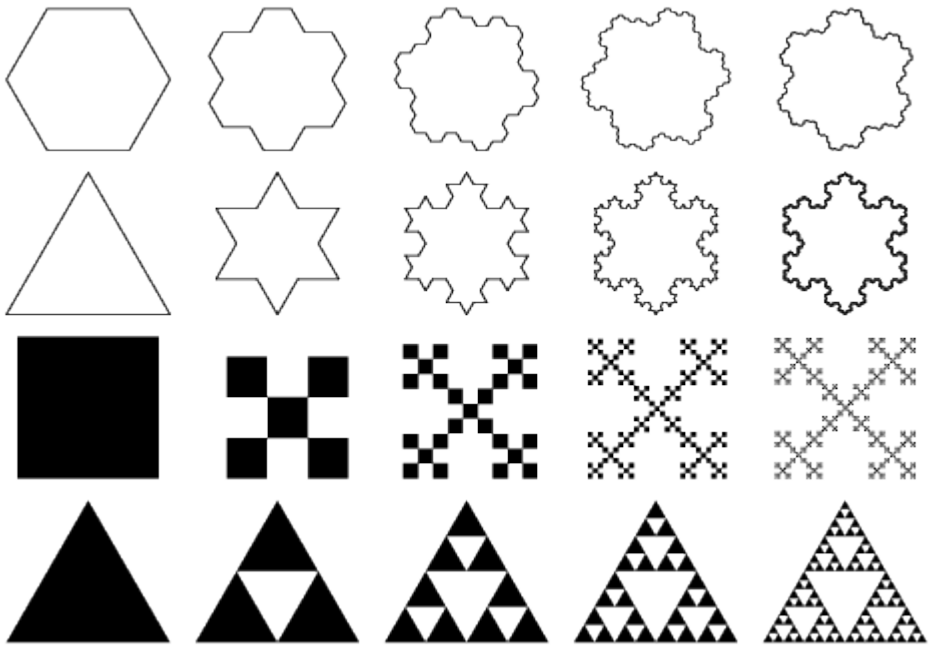

Figure 1 Example of Fractal Shapes [9]

2.2.1.4 Offline vs. Online Generation

Online and offline generation refers to the time at which data is generated. If generation is online, it occurs on the user’s (in this case, the player’s) machine [4]. Offline generation, however, occurs on the developer’s machine before transfer to the user [4].

2.2.1.5 Constructive vs. Generate & Test Methods

It can be said that there are two classes of data generation methods. A constructive method is one which generates data and is then done with it [4]. On the other hand, a generate & test method would generate data and then applies a test (or tests) to the data to ensure it meets some

requirements. If the data does not meet the requirements, then some/all of it is discarded and more data is generated [4]. This is repeated until sufficient requirements are met.

2.2.1.6 Deterministic vs. Stochastic Methods

Some generation methods produce the same output for the same input, others do not. Methods which do produce the same output are deterministic methods. Those which do not are classed as stochastic methods. However, it is important to note that this does not include the seed value [4]. Seed values always generate the same data.

2.2.2 A Basic Example

Typically, games use coherent noise in terrain generation. Coherent noise is a smoothed-out series of pseudo-random numbers (based upon a seed value) layered over each other. This noise function can be used to define map data.

The most famous example of coherent noise is Perlin noise (see Figure 2 below). Pseudo-random numbers are useful because, using the same seed, the same number series can be generated, meaning that there is no need to store (or transmit) the data. As well as a seed value, generators can be designed to take other parameters as input which define desired characteristics of a map.

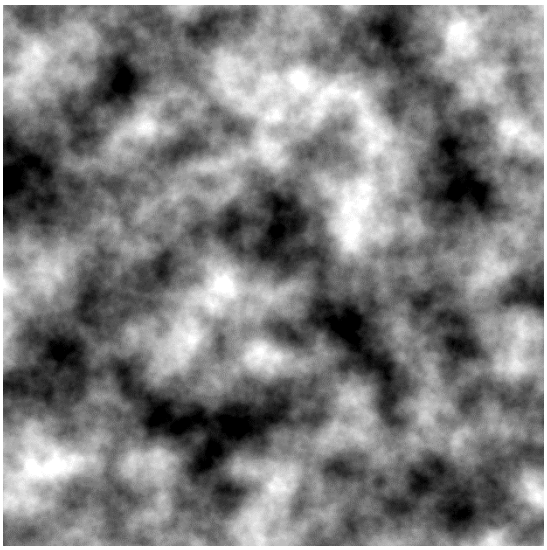

Figure 2 An example of Perlin noise

A basic generator could use a Perlin noise function, with a specified seed value, to determine the height (z value) of terrain at any given point (in the x-y plane) which is to say:

$$PerlinNoise_{seed}(x, y) = z$$

Using these height values a 3D mesh can then be generated to represent the terrain. In practice, Perlin noise functions output values in the range $[-1, 1]$ so the value generated must be mapped into the desired height range e.g. $[0, \text{maxHeight}]$.

This method has a few drawbacks: terrain can be boring, there can be no overhanging terrain, and terrain is not life-like (in the sense that it is not based upon physical principles), and it only defines terrain, there is no decoration. A developer would layer this method with other techniques to generate a more complex map consisting of terrain, vegetation, and man-made structures.

2.3 *Minecraft* (Mojang AB)

2.3.1 Description

Perhaps the most popular video game using procedural generation today is *Minecraft* [10] with over 106,859,714 copies sold [11]. *Minecraft* uses online generation, generating complex voxel-based volumetric worlds which contain a wide range of biomes [12]. Each of these biomes have different and unique terrain styles, can contain different flora and fauna, and contain different structure types. The worlds also contain various water features (oceans, lakes, rivers), cave systems, mine shafts, villages, ruins and ancient temples.

Not only are there biomes, but three distinct dimensions within *Minecraft* worlds, these being: The Overworld (earth-like world); the Nether (hell-like world); and the End (a floating island dark region). The Overworld worlds are huge; the playable area in *Minecraft* is ~60,000,000x~60,000,000 blocks with a world depth of over 4000 blocks (as of *Minecraft* version 1.10.2) with each block occupying 1 cubic meter of space. To support such a large world, the terrain is loaded in chunks as the player explores. This massive scope is allowed because of procedural generation, especially when one considers that *Minecraft* was initially developed by one person.

It is difficult to determine exactly how *Minecraft* worlds are generated today, however, earlier versions of the game used a 3D Perlin noise function to generate terrain. Noise values were evaluated in 3D space in intervals of 8 for the horizontal axis, and 4 for the vertical axis for efficiency reasons. These values were then used in a linear interpolation to get a noise value for each block (voxel). This noise value determined what type of block would occupy that space, e.g. a noise value less than 0 would indicate an air block [13].

Worlds generated in *Minecraft* are extremely diverse. Using a biome system with unique generation rules for each one, worlds are full of rolling plains, extreme mountains, deserts, great oceans, dense jungles, and more. However, this comes at a cost. Sometimes the edges where two (or more) biomes meet are usually denoted by a harsh switch as opposed to a gradual transition, see Figure 3 below.

Figure 3: Screenshot of Minecraft showing the boundary between a desert region, a mountainous region, and a forest region.

Minecraft worlds are comprised not only of terrain; villages (and other structures) are also generated in the world. However, sometimes the placement of these villages is not always perfect and they can be generated in the terrain to produce odd results, see Figure 4, Figure 5 below.

Figure 4: Screenshot of Minecraft showing a desert village partially spawned in a sand dune.

Figure 5: Screenshot of Minecraft showing other view of desert village partially spawned in a sand dune.

Worlds can also contain some smaller anomalies such as floating blocks, see Figure 6 and Figure 7 below.

Figure 6: Minecraft screenshot showing two floating dirt blocks (dead centre).

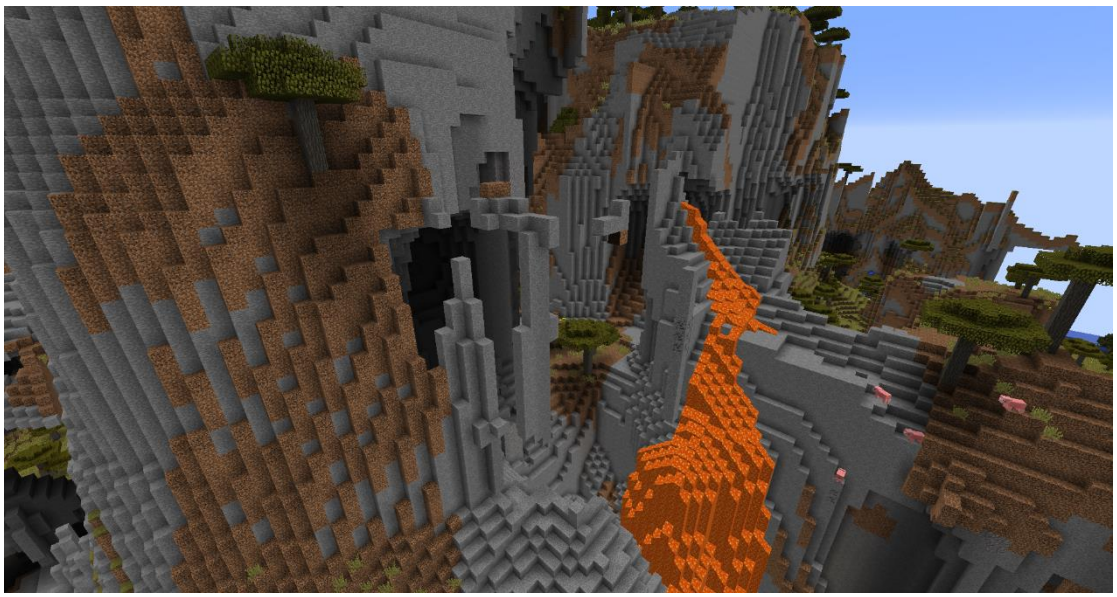

Figure 7: Minecraft screenshot showing floating stone blocks (slightly above dead centre).

In Figure 3 we can also see another weakness of *Minecraft's* world generation: rivers do not always make sense. In fact, for the most part they start and end randomly, unlike in the real world where rivers usually start high up in mountainous regions and flow down to terminate at the ocean or other body of water. A custom generator could attempt to avoid this issue by placing river sources in mountainous areas, and then simulating how the water would flow downwards. However, that method would require knowledge about the surrounding area which, in *Minecraft's* case, is unknown.

Although there are issues with *Minecraft* worlds they are few considering the overall size of the worlds. Any issues of traversability are, in a sense, nullified because of the player's ability to place and destroy blocks at will.

The Nether is a cave like world full of red rock and plenty of lava, see Figure 8. This region is unlike any other location you would find in of the other dimensions. Likewise, the End is also unlike the other dimensions, see Figure 9.

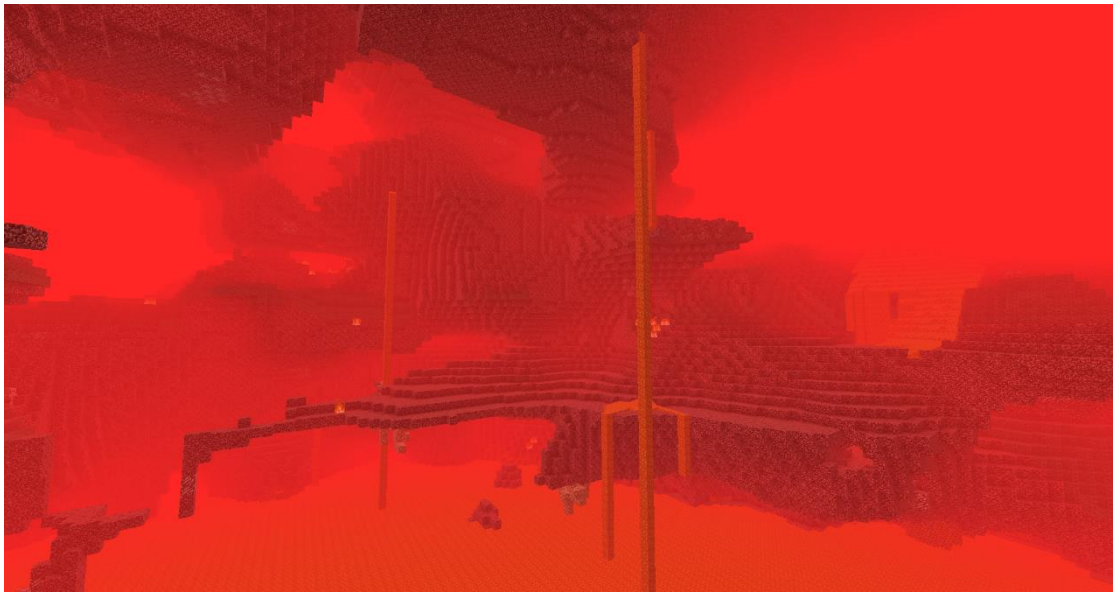

Figure 8: Minecraft screenshot showing the Nether world (player has dark-vision buff).

Figure 9: Minecraft screenshot showing the End dimension (edited to increase brightness).

The variance between these different dimensions and the different biomes suggests a modularly designed generator. The world space is segregated into different biomes and for each biome a specific generator is used, given some input, to generate that regions terrain; place flora and fauna; and place structures.

2.3.2 Conclusion

Minecraft's worlds, whilst not realistic, are interesting, rich, and incredibly variable. They inspire the player to explore and have helped make *Minecraft* become a commercial success, and a global phenomenon. *Minecraft* is an excellent example of how procedural generation can help expand the scope of a game.

2.4 *Sir, You Are Being Hunted* (Big Robot)

2.4.1 Description

Sir, You Are Being Hunted [14] is a FPS (First Person Shooter) stealth game set in a stylised British countryside on an archipelago of 5 islands (see Figure 10). The player spends their time evading enemies and collecting items to win the game. There are buildings which contain loot, such as weapons and food, and various types of enemies which patrol the islands. Islands contain different types of flora which the player can use for cover, such as tall grass. The game also has docks on the islands which act as teleports the other islands. These maps are generated online, just before entering the game.

Sir, You Are Being Hunted was developed by a team of 3 developers, and is a perfect example how procedural generation can expand the scope of a project with few members.

Figure 10: Screenshot of an example island [15].

2.4.2 Generator Description

To generate maps *Sir, You Are Being Hunted* uses Voronoi diagrams (see Figure 11) to generate cells [15] which are of a specific region (e.g. coast, field). A multi-octave Perlin noise function is used to determine the heights of the corner points of each Voronoi cell, then another layer of a 'much more fine-grained Perlin noise' (most likely meaning noise with a higher octave

count) is applied to the terrain to improve the look of it [15] by adding some roughness to the terrain. For each cell, a specific generator (for the region of the cell) is used to generate the contents of that cell [15], like *Minecraft's* biome system. In *Sir, You Are Being Hunted* there are many regions such as walled fields, villages, churches, forests/copses, industrial, and mountainous regions [16].

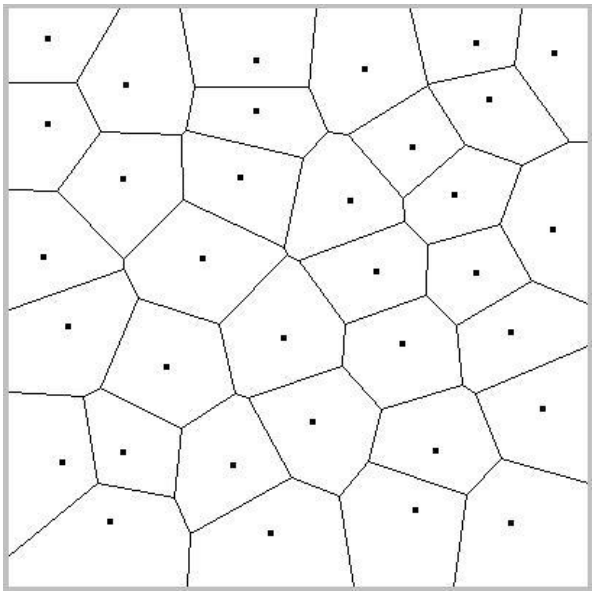

Figure 11: An example Voronoi diagram [17].

To solve the issue of making the boundaries of regions appear realistic, a Brownian motion-type walk is applied to dictate how to mix the two textures together [15] [16]. An example of this texture blending process can be seen in the lower left-hand corner of Figure 10, where a beach texture is blended with grassy textures in a random manner.

Depending on the region of the Voronoi cell, the generator places various 3D models (props) and loot. For example: if the cell is a walled field the region-specific generator will place wall models along the perimeter of the cell, leaving gaps for gates [15] [16].

These region-specific generators can be complex, the generator for village regions controls the layout of houses, gardens (and which type of garden), and roads [15]. There is also a generator for village names. It mixes substrings of different types, specifically:

$$Village\ name = Phrase\ A + Phrase\ B + Subtitle$$

Where Phrase A could be from the set {Little, Great, Upper, Bishops, …}. Phrase B could be from the set {Bury, Hampton, Cove, Dangling, …}. Subtitle could be from the set {On

the end, market town, in the wold, …}. Thus, a possible output could be: “Little Cove market town”. Obviously, duplicate names are banned [16].

After these cells are generated there is a phase in which the regions are evaluated against a set of rules to remove such inconsistencies such as villages underwater [16].

A complicated issue is the placement of roads. Big Robot used a process in which the map was modified to fit the placement roads as oppose to a generate and test type solution. After deciding what two locations are to be joined, the cells between them are cleared, parallel cells are placed to force straightedges alongside roads (with some programmed ‘wiggle’). Although this is fast, it does create trivial solutions i.e. the roads are almost straight lines between locations, see Figure 10. However, within the scope of the game this is a boon as it provides a way for the player to navigate.

In early versions of *Sir, You Are Being Hunted* there were issues with the placement of docks. Docks, which are used to teleport the player to another island, could spawn in land-locked lakes (see Figure 12) and by sheer cliffs [18]. If the dock spawns in a land-locked lake, although this goes against the logic of the game, the player still use it and keep playing the game. If the dock spawns by a sheer cliff that players game can potentially be finished as one of the games mechanics is that when the player enters the water a sea monster attacks and kills them after a period to stop them swimming out to see. This could trap the play at that dock, unable to enter that island and which is necessary for the player to complete the game.

Figure 12 Land-locked dock in Sir, You Are Being Hunted [18]

2.4.3 Conclusion

Sir, You Are Being Hunted is very good at generating maps which fit the concept and style of the game, which is to say: very good at generating maps with a stylised British countryside feel. The generator's success is partially tied to the fact that the maps are finite in nature, as opposed to *Minecraft* which requires the possibility of infinite generation. This finiteness allows the generator to view the map rather than as a set of discrete areas which only affect their neighbours. Therefore, this allows a more cohesive map.

This idea of finiteness is not trivial. There are many aspects of game world which require a global knowledge of the map. An example would be roads or rivers. To know the path of a road or river you need to know where it starts, ends, and what is in between. If the river or road start and end in the same chunk of the map that is being generated, then there is not a problem. However, it is preferable that they could span whole regions as they do in the real world. In *Minecraft's* case rivers fail because they are based upon noise as opposed to a logical reason e.g. rivers flowing down from mountains to the ocean.

2.5 *No Man's Sky* (Hello Games)

2.5.1 Description

No Man's Sky [19] is a FPS space exploration simulation game. The player can pilot a spaceship to travel between planets and solar systems. On planets, the player must gather resources to upgrade their equipment and fuel their ship. Whilst there is no win state for the game, the aim is to travel to the centre of the galaxy. However, the player is generally encouraged to simply explore the galaxy. This is important to note because as previously discussed, it is important to keep in mind the driving goal of a game when designing the worlds for it. In *No Man's Sky,* it is appropriate to say that the worlds must encourage exploration, the player must want to move forwards otherwise the game will become boring and the player will stop playing.

The scope of *No Man's Sky* is larger than both *Minecraft* and *Sir, You Are Being Hunted*. *No Man's Sky* aspires to generate a galaxy worth of solar systems down to the plants on the planets.

2.5.2 Generator Description

No Man's Sky uses a layered generation process. Initially the positions of the solar systems in the galaxy are known. Each solar system uses its position as a seed value for generating data about that solar system, e.g. number of planets or space stations. Relevant data (e.g. distance from the sun) and the seed gets passed to the 'planet generator' which generates the planet meshes and data pertaining to the ecosystems of that planet. This planet data is then

passed to the creature generator which, depending on the ecosystem of the planet, generates creatures that live there [20].

Terrain information is voxel based [20]but still uses a mesh-style continuous terrain (see Figure 13 to Figure 20) unlike *Minecraft* which uses voxels and builds terrain from cubes. First the surface of the planet is partitioned into region types (e.g. mountains, plains) using Voronoi/Worley noise [20]. Then the surface is built up using a similar method to *Minecraft*; a function is evaluated at certain points (in 3D) in regular intervals to determine the density at that point which specifies terrain information [20] like *Minecraft*. This allows for over hangs and floating islands which a simple height map based approach does not allow [20].

The terrain generator in *No Man's Sky* aims to achieve an eroded affect [20]. As each voxel does not know any information about its neighbours, Perlin noise is used to determine erosion effects at a given point [20]. This also has the nice bonus of being more efficient as calculating erosion is as simple as evaluating a Perlin noise value, rather than simulating weathering and, therefore, needing to consider area and neighbouring voxels.

Caves are also present in *No Man's Sky* (see Figure 20), they are generated using Perlin worms which are also used to generated actual worm-like terrain structures [20].

Textures are blended well between different materials (e.g. rock and grass) and different regions. This texturing seems to be based upon a planet sized texture map (using the information which used to specify regions) mixed with some local blending method [20].

Next a set of 'decoration' (trees, rocks, other plants) objects are generated based upon planet data and are placed on the planet [20]. The generation of decoration objects is done by deforming a set of prefabricated models (created by artists). In the example of trees, the artists would create a set of basic tree shapes. Then the generator would texture the tree based upon the planet data and then would deform the branches, for example: elongating upper branches and creating a bend in the trunk. These trees would then be planted around the planet in clusters, mixing trees of various sizes and other plants [20].

Although planets are finite spaces each voxel is ignorant of its neighbours i.e. the generator does not consider any other part of the planet to generate its data which allows for data to be generated in real-time at the cost of life-like cohesiveness.

2.5.3 Conclusion

No Man's Sky generates some interesting planets with great inter-planetary variation. Figure 15, Figure 16, and Figure 18 shows creatures on different planets. Figures Figure 13, Figure 15, and Figure 17 show the variation in plant life.

Figure 13: Screenshot from No Man's Sky [21].

Figure 14: Screenshot from No Man's Sky [21].

Figure 15 Screenshot of a planet in No Man's Sky [22].

Figure 16 Screenshot of No Man's Sky [23].

We can see from Figure 13, Figure 14, Figure 15, and Figure 19 that a wide range of planets are generated. However, the terrains on those planets are the same across the planet. Considering that planets are generated at run-time and specific terrain is generated in real-time it is impressive what *No Man's Sky* can generate. However, planets are generally uninteresting (after a few hours of play time), they are mostly 'samey' barren rocks with a spattering of flora and fauna. This does seem to fit the scope of the game which is supposed to limit the amount of more interesting planets to make them more important to the player [20].

Figure 17 Screenshot of No Man's Sky showing a strange plant.

Figure 18 Screenshot of No Man's Sky showing a creature.

Figure 19 Screenshot of No Man's Sky showing odd terrain.

Figure 20 Screenshot of No Man's Sky showing the entrance to a cave.

2.6 *Outerra*

2.6.1 Description

Outerra [24] is a planetary game engine. It uses procedural generation methods to provide arbitrary resolution of terrain. This is done by using fractal methods which take in parameters defined by elevation and climate data. These methods then output terrain data to fill the gaps in the predefined world-data. These fractal methods have been designed so that they mimic natural processes to try and generate believable terrain [25].

The results are strikingly like real-world scenes, see Figure 21 and Figure 22.

Figure 21 Screenshot of Anteworld showing a fjord

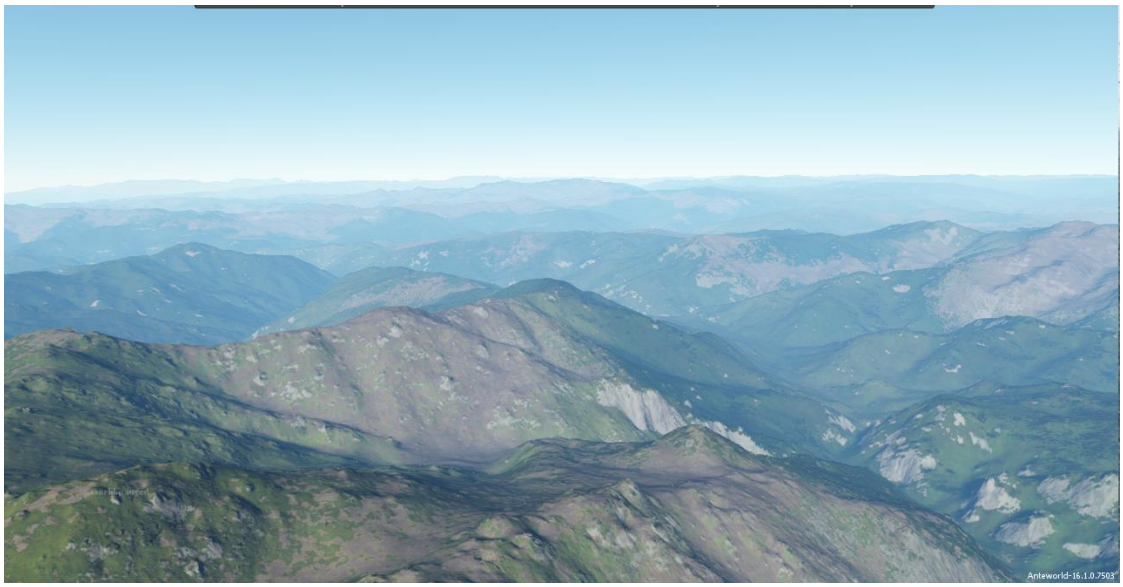

Figure 22 Screenshot of Anteworld showing a mountainous region

Figure 23 shows a distant rock-face, Figure 24 shows the same rock-face but up close. The effect of this arbitrary resolution (due to procedural generation) is very powerful and heavily contributes to the realism which is possible in *Outerra*.

Figure 23 Screenshot of Anteworld showing a rock-face at a distance

Figure 24 Screenshot of Anteworld showing a rock-face up close

Figure 25 Screenshot of Anteworld showing a river carving its way through terrain

2.6.2 Conclusion

Outerra is a perfect example of how procedural generation can have a relatively small role to play in games, and still produce fantastic results. In *Anteworld* [26], which is a demo of *Outerra*, the Earth has been recreated at a "true-to-life scale" [26]. In this demo, it easy to see the realism which is achieved, in part, by procedural generation techniques.

2.7 *Elite: Dangerous* (Frontier Developments)

2.7.1 Description

Elite: Dangerous [27] is a massively multiplayer space adventure video game in which the player takes control of a spaceship and explores our galaxy, the Milky Way [27]. Procedural generation is used to generate ~400 Billion stars using first principles at a 1:1 scale [27] over a very long simulated time. Planets are also procedurally generated, around the stars, from data about the solar system such as the star itself and the available matter [28]. These planets are generated using geological principles, such as tectonic plates, to provide a "realistic and scientifically accurate planet." [28]. Not only terrain is generated, even atmosphere is generated to determine how light would interact with it and thus determine how the planet would look.

Planets, space stations, and other properties are generated by the player's machine [29] using seed parameters. These seed parameters are the same for all players, ensuring that the game world is consistent for everyone.

2.7.2 Conclusion

Elite: Dangerous provides a realistic, 1:1 scale, galaxy to explore with ~400 Billion stars. This amount of content could only be possible with procedural generation, especially when one considers that some details (planets, properties, space stations) are generated as needed on the player's machine. This massively reduces the storage requirements on the game's multiplayer servers and the amount of data which needs to be delivered to the player's, thus reducing the load on the network.

Planets can be varied, however, given the number of planets there are a lot of similar looking planets, see Figure 26 and Figure 27. This fits the objectives of the generator, to generate a realistic galaxy, as space does contain a lot of similar looking rocky planets.

Figure 26 Screenshot of Elite: Dangerous showing the player exploring a rocky looking planet [30]

Figure 27 Screenshot of Elite: Dangerous showing the player exploring another rocky looking planet [30]

Figure 28 Screenshot of Elite: Dangerous showing the player viewing a pock-marked planet from orbit [30]

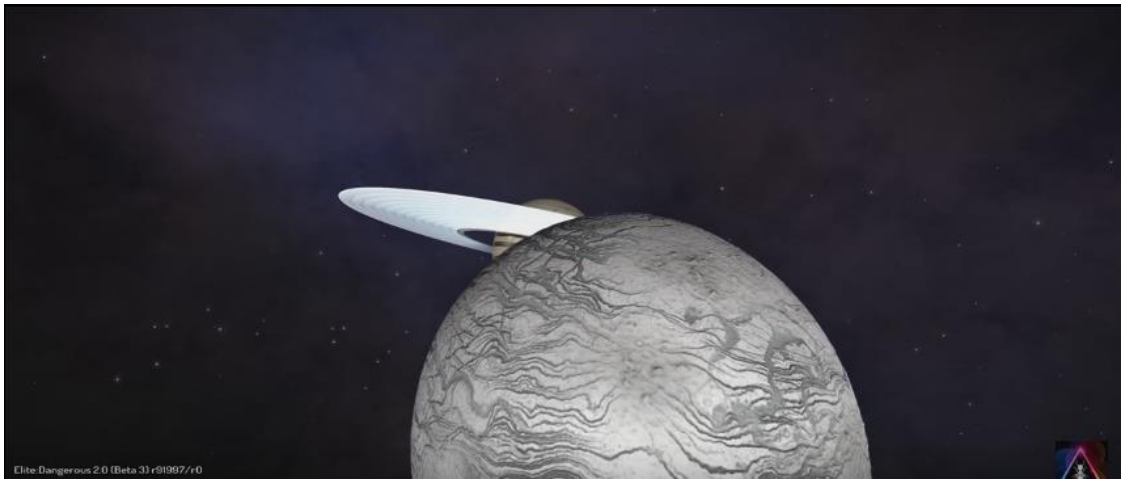

Figure 29 Screenshot of Elite: Dangerous showing the player viewing an ice-covered planet from orbit [30]

Figure 29 shows an example of an extremely interesting ice-covered planet generated by *Elite: Dangerous*. The amount of surface detail is very high, as shown by Figure 30.

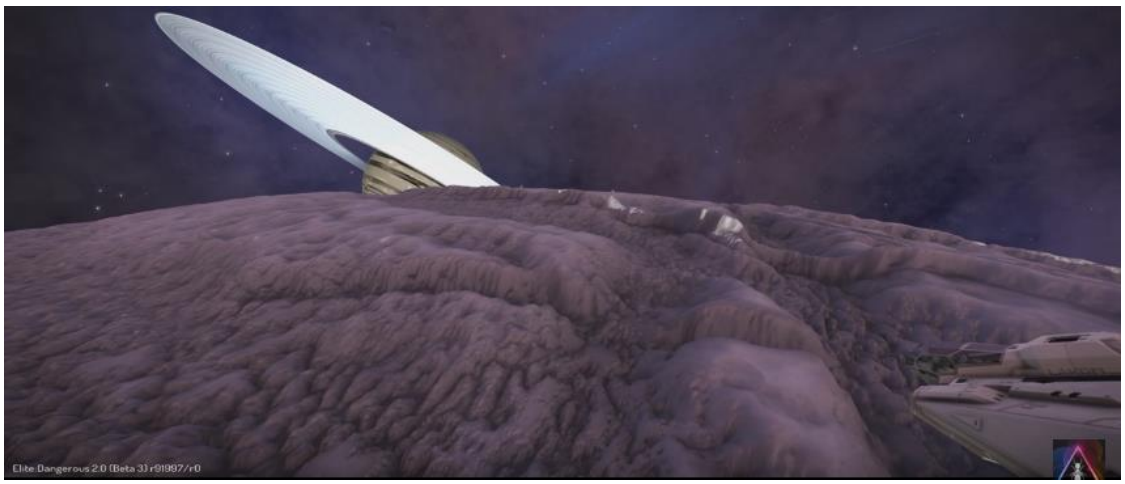

Figure 30 Screenshot of Elite: Dangerous showing the player flying over the surface of an ice-covered planet [30]

This provides further evidence that real-world physical laws can be used to dramatically increase the quality of worlds for video games.

3 Technical Description

3.1 Requirements

At the simplest level, the system needs to be able to generate world data and then display that world data so the user can explore it. The system also needs to be able to handle user input to allow them to fly around generated maps. It is clear, then, that there must be at least four significant systems, one system concerned with converting generated data into a render-able form, another system concerned with data generation, a system concerned with taking in user input and acting accordingly, and a rendering system.

The following sections will contain a further analysis of each of these systems.

3.1.1 Data Generation

There are many requirements for the worlds generated by the system. A world consists of a terrain mesh (or meshes), rivers, lakes, oceans, and vegetation. These worlds need to be interesting, physically based, generated quickly (or rather, not slowly), and reasonably diverse, both between different maps and within same map

3.1.2 Data Conversion

Data needs to be converted into an appropriate form to be rendered. For example: terrain data needs to be converted into mesh data, and tree placement data needs to be converted into billboards/tree models.

3.1.3 User Input Handling

The user needs to be able to control the camera, exit the application, and input parameters which are then used to generate the world.

3.1.4 Rendering

Data needs to be rendered in 3D, efficiently, across as many platforms as possible to increase the number of people who can benefit from this work.

3.2 Design and Evaluation

This project used an Agile style development process. With the goals previously mentioned in mind, a list of tasks was compiled. The first task was to choose a rendering system. This would dictate how data would be generated, for example: what language would be used. Once this was done, it was possible to design the data conversion along with the data generation systems. These two were done in conjunction as they are dependent systems. It is required that the data

generated can be converted and rendered if one wishes to see how well a world generator performs. Likewise, to test data conversion, one must have some data to convert. Initially, the generated data was very simple and more of a placeholder for later developments.

Once basic systems were in place to generate and render data, it was necessary to provide user interaction for the generated worlds. At this stage, the user input handling system was designed and implemented. Initially, the only functionality provided was to allow the user to fly about the world. Later would come a GUI parameter input.

As the project advanced it was necessary to revisit various parts of the systems as bugs were found, or unforeseen requirements were found.

It was known at the start of the project that this style of development was to be used and as such, Git [31] was used to provide source version control. This allowed for fast prototyping without fear of 'breaking' code or developing the system to a broken state from which it could not be fixed.

3.2.1 Choosing a Rendering System

During the preliminary stage of the project, several rendering systems/game engines were evaluated for suitability for rendering generated world data. Given the objectives of the project, there were multiple requirements of the chosen system.

One of the most important requirements was ease-of-use/ease-of-learning. Due to the limited time for the project, the chosen engine needed to be quick to learn and easy to use to minimise the time spent on implementing designed systems and algorithms. This is also beneficial to any developers who wish to understand the created system.

Another important consideration is ease of cross-platform compilation. Given that the project was intended as a learning experience, it was beneficial that it could be easily compiled on a wide range of systems by users.

A similar consideration is popularity of the chosen engine. It was beneficial, to developers, that a widely-used engine was used as it would minimise the extraneous learning (e.g. learning the programming language of the engine) that the developer would have to do when trying to understand the created system.

Licensing is another consideration. An open-source license is not necessarily required, however, the ability to use the candidate system for free is extremely beneficial when considering that the project aims to be accessible to as many users/developers as possible.

3.2.1.1 Irrlicht

Irrlicht [32] is an open source 3D rendering engine written in C++. It was initially researched as an option because it was advertised as being fast, relatively easy to understand, and easily cross-compiled. Perhaps the most important feature was a premade terrain class which dynamically adjusted the LOD of the mesh depending on camera position. However, this proved inadequate as the maximum size of the terrain was 512x512. An attempt was made to try and increase this size but it was deemed that it would take too much time to implement. *Irrlicht* also has a relatively low popularity when compared with the other candidates.

3.2.1.2 OGRE3D

Like *Irrlicht*, *OGRE3D* [33] is a cross-platform 3D rendering engine written in C++. It differs from *Irrlicht* in that it is more regularly maintained. However, it is larger than *Irrlicht*, and somewhat more difficult to learn, despite having excellent and extensive documentation. Cross-compilation of *OGRE* is possible; however, it was more difficult than Irrlicht.

3.2.1.3 OpenGL

OpenGL [34] is an open source, cross-platform, rendering API. *OpenGL* would have been used to create implement a small, basic game engine. This would have been almost ideal as it would have been designed to fit the generation systems perfectly. For example, early designs of the system allowed for infinite maps, which would have required a level-streaming system.

A considerable negative to this approach is that it would have required a lot of time to implement. In fact, writing a game engine could be a project in and of itself.

3.2.1.4 Unreal Engine

Unreal Engine [35] is a widely-used industry-standard game creation tool. It has extensive documentation and learning material. It is highly efficient, supports easy cross-compilation (at the click of a button), and offers free versions for anyone to use available on Windows, Linux, and Mac OS X. Users can use either C++ with *Unreal's* own API, or *Unreal's* Blueprint visual scripting system to create applications.

Unreal is widely used by large game studios, small independent studios, and even hobbyists.

3.2.1.5 Unity 3D

Like *Unreal*, *Unity* [3] is widely-used industry-standard game creation tool. It has extensive documentation and learning materials. It is also highly-efficient and supports easy cross-compilation. There are free versions for anyone to use and it is available for Windows and OS X machines. Users use C# to write code along with Unity's own API to interface with the engine.

Unity was chosen over *Unreal* simply because it was easier to learn and the author was already familiar with creating applications with it.

3.2.2 First Prototype

An initial prototype was developed to test the suitability of *Unity*, and to develop a terrain system capable of efficiently rendering a large amount (> 2,000x2,000 points) of height data.

3.2.2.1 Description

This prototype consists of a terrain rendering system and a placeholder Perlin noise generator to generate terrain data. The terrain system split the terrain data into chunks, these chunks have meshes at different level of details for use at different distances-to-the-viewer.

A basic terrain texture was also generated; colours were determined based upon height of the individual points. This was done because it added some sense of scale to the worlds and made it easier to see the terrain.

3.2.2.2 Issues

During development, there were a few issues. The edges of adjacent chunks meshes did not line up exactly. This was because the meshes did not overlap in data so of course the height values would be different. The solution used was to change the height values between adjacent meshes to be a midway height.

3.2.3 Second Prototype

This prototype was developed as an attempt to provide faster generation of data and provide a method of inspecting generated data, in a 2D form, before rendering.

3.2.3.1 Description

This prototype consisted of two applications: A *Unity* application which loaded terrain data from a heightmap file, rendered the world, and allowed the user to explore said world; and then a second small C++ application to generate the data and provide a GUI to input parameters for data generation, and to view the data.

The GUI for the second application was implemented using *FLTK* [36], a small, cross-platform, C++ GUI library. The GUI allowed input of parameters such as: seed, map dimensions, and then noise settings such as: frequency, persistence, and lacunarity. The GUI also allowed the user to view the data in a 2D form, see Figure 32. This application also outputted the generated data to a heightmap file. This file stored data in a binary format as show in Figure 31.

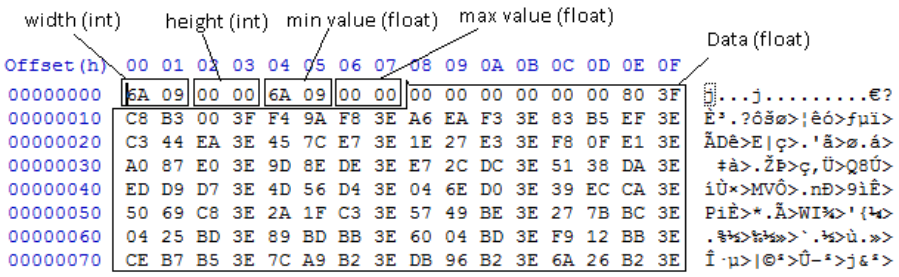

Figure 31 File format of heightmap files

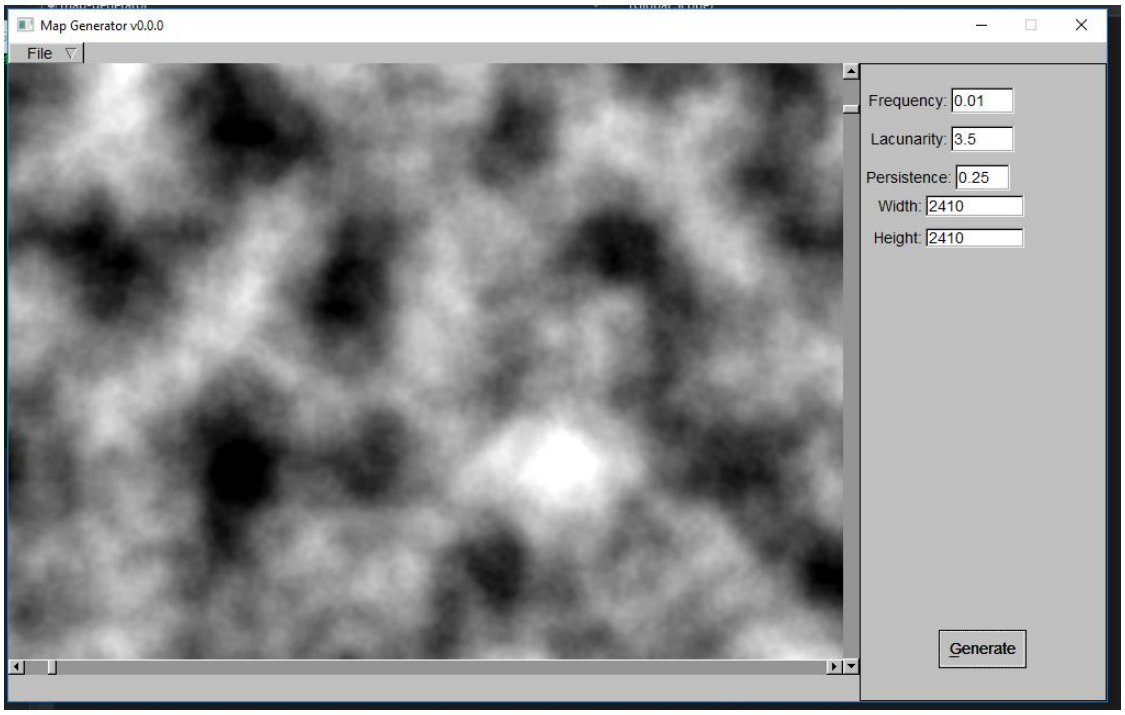

Figure 32 GUI for second prototype

3.2.3.2 Issues

The applications themselves worked fine, however, it was judged that the resources required to develop two applications would have been too great for the scope of the project. There are also difficulties in building the GUI application for cross-platform.

There were also issues when trying to read in a file in the *Unity* application. It was necessary to provide an explicit path to the file as *Unity* does not keep its resources in the same folder structure across different platforms.

3.2.4 Proposed Server/Client Prototype

The aim of this prototype was to provide a way to generate infinite maps, without having the generation done by *Unity*. This was thought to provide faster data generation, as this prototype was not constructed, it is unknown whether this would have been true

3.2.4.1 Description

This prototype would have consisted of two applications: a server application which would generate world data and streamed it to the second application, which would have been a *Unity* application, whose task was to render the generated data and allow the user to explore it.

3.2.4.2 Issues

As with the previous prototype, this required development of two applications. Networking, even locally, can cause difficulties which can be avoided by keeping everything in a single application. Cross-platform compilation would most likely have proven difficult.

3.2.5 Final Application

The final application was intended to use everything learned from previous prototypes to fulfil the project's objectives.

3.2.5.1 Description

The final application makes use of the previous developer terrain rendering system. However, this application also handles the generation of world data.

Generation makes use of multiple generator functions and of a rudimentary tectonic boundary system. There is also a work-in-progress rain-simulation process.

First, tectonic boundaries are generated. This is done by pseudo-randomly generating a set of starting points on the edge of the map. Next, a random angle is generated which determines the overall angle of the boundary line. Then, from the starting point a line of fixed length is generated (pointed in the direction of the generated angle), the start coordinates are moved to the end of this line and the process is repeated until the line meets the edge of the map. This yields a list of connected lines travelling from one edge of the map to another, see Figure 33.

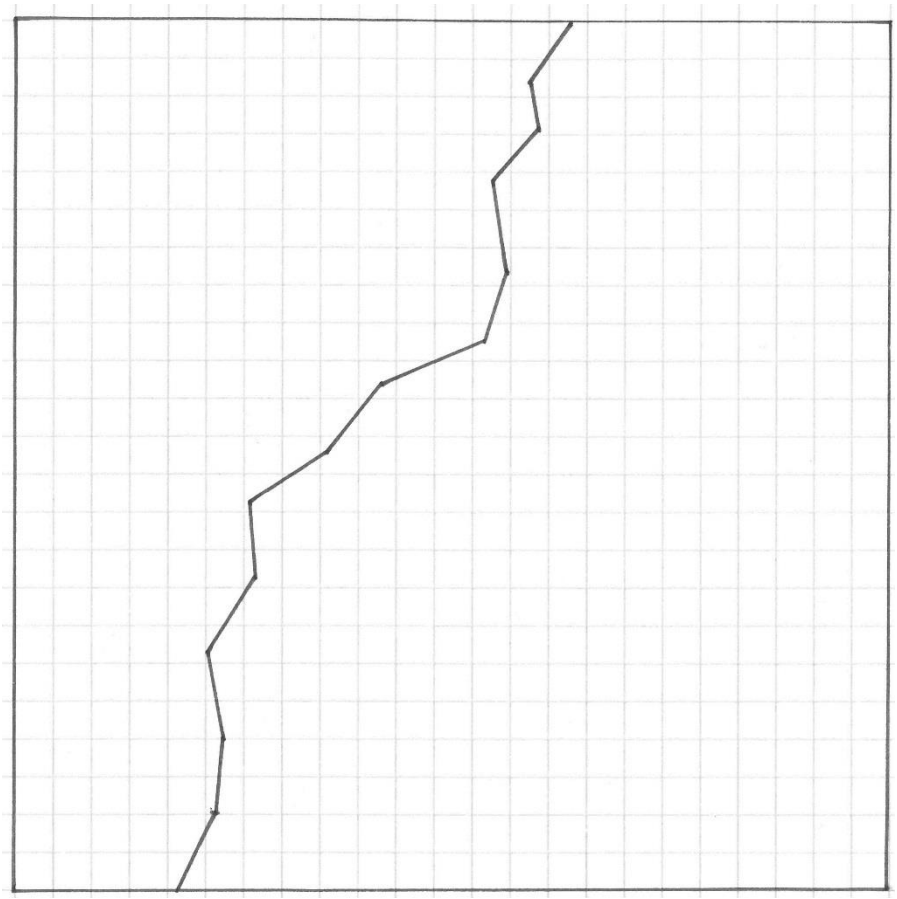

Figure 33 Example of a tectonic boundary

With boundaries created, terrain can now be constructed. The terrain is represented by a 2D array of floating point numbers representing the height of the terrain at that point. The height of each point in the array is determined by combining three noise values:

- A Base Perlin noise value to specify a rolling hills base
- A fine-grained Perlin noise value to add some more fine-grained noise to the smooth base
- A Fractal noise value for adding mountainous regions to the map

In practice, these three values are combined using the following equation:

$$height = w_{xy} \times fractal\ value_{xy} + fine\_perlin\ coefficient \times fine\_perlin\ noise\ value_{xy} + base\ perlin\ value_{xy}$$

The fine Perlin noise value is multiplied by a constant coefficient which is used to control the strength of the effect it has on the terrain. This coefficient was determined by trial and error.

The Fractal noise is multiplied with a weighting coefficient, w_{xy}. This coefficient is determined by calculating the smallest distance to a boundary line. This means checking all the lines in the boundaries, skipping those which cannot be connected to the point with a perpendicular line. However, it is important to check the points proximity to end points of the line as to not incorrectly ignore points in the shaded area show in Figure 34, which is outside of the boundaries of both line A and B.

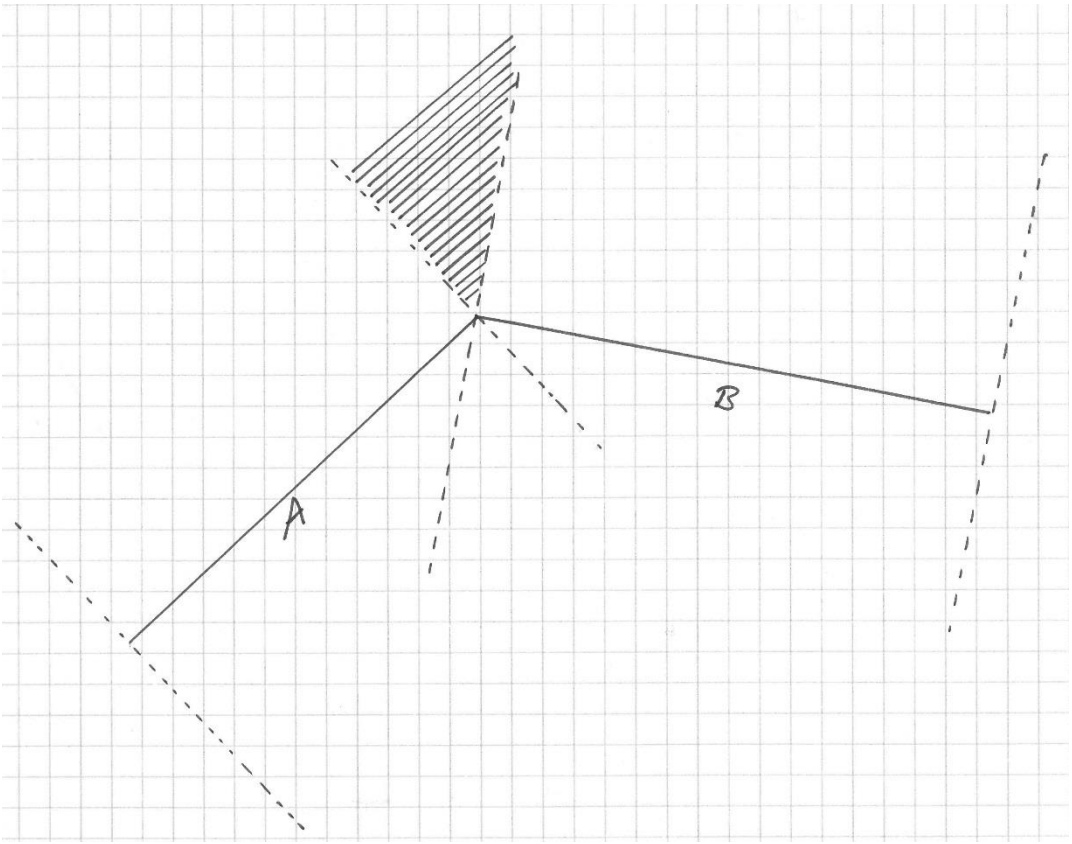

Figure 34 Diagram showing Boundary Dead spot. Dotted lines show the boundary of the lines. The shaded area denotes the area in which the points are neither in the range of either line A, or B.

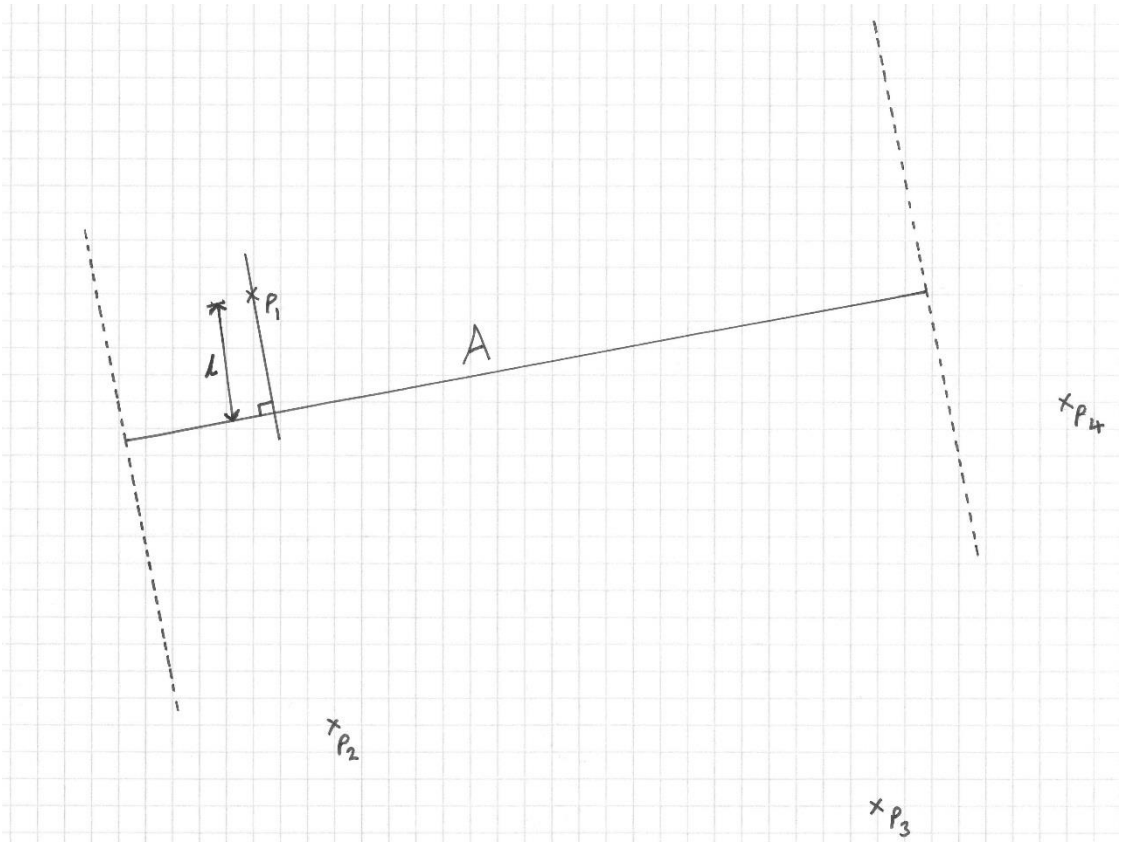

Figure 35 Diagram depicting scenarios when calculating weights for points in the map

As mentioned above, each a weight is calculated for each point in the map. In Figure 35 a line, A, with equation $y = \alpha x + c_1$, which is a segment of a boundary, can be seen, along with points p_1, p_2, p_3, p_4 which are 4 scenarios which can occur when calculating heights for points in the map. The dashed lines depict the outer-edges of the domain for which the line is valid (only points which lie on a perpendicular intersecting line are within range of the line).

Consider point p_1, to calculate the distance between the line A, the closest point on A to the point must be determined.

This is done by calculating the perpendicular intersecting line, denoted A_i (with equation $y = \beta x + c_2$), between A and p_1, and then satisfying the equation. The perpendicular gradient, β, remains constant for all points and thus need only be calculated once: at the creation of the line A. β is the reciprocal of α, which is to say:

$$\beta = -\frac{1}{\alpha}$$

The intersect, c_2, however, is unique for each point and is calculated with the following equation:

$$c_2 = y_1 - \beta x_1$$

Where x_1 and y_1 are the coordinates of the p_1.

With the perpendicular intersecting line now known, the only thing left to work out is what point on the line A is closest to p_1. This is done by equating the two lines A and A_i and solving for x:

$$\alpha x + c_1 = \beta x + c_2$$

And then rearranging for x:

$$x = \frac{c_2 - c_1}{\alpha - \beta}$$

With the x-coordinate known either the equation for A or A_i can be used to calculate the y-coordinate. The distance between two points, (x_1, y_1) and (x_2, y_2), is then found using the following equation:

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Now the distance is known (denoted as *l* in Figure 35), the weight must be calculated. Each line segment has a range of affect, essentially a maximum distance below which a point is affected by the boundary. The closer a point is to a boundary, the stronger the effect of the boundary, the further away, the weaker the effect. This can be modelled with the following equation:

$$weight = \frac{(distance - maximum\ range)}{maximum\ range}$$

The values of weight range from 0 to 1 (inclusive).

Now, considering Figure 35 again, specifically p_2, p_3, p_4. p_2 is further away from the line than p_1 and will therefore have a smaller weight. p_3 is outside of the range of line A and therefore will have a weight of 0. p_4 is within range of the line but is not within the boundaries of the line, however, its weight will be calculated using its distance from the end of the line.

The rain-simulation system was intended to generate rivers and lakes in a realistic manner. The premise is that by placing many raindrop agents at random places on the map, and then simulating their path downhill (as water will always flow downwards), it would be possible to determine where rivers and lakes would likely occur.

River placement would be determined by looking for areas where multiple agents travelled. Lakes, on the other hand, would occur where many agents got trapped by 'bowl' shaped terrain.

3.2.5.2 Issues

After implementing the boundary system, there was a bug which meant that points in the obtuse-angle region where two lines meet, see the shaded region in Figure 34, were classified as not in

range of either line. This was soon fixed by checking each points proximity to the end-points of lines.

The boundary creation method does not always create believable or interesting boundaries. These failed boundaries can either be small, such as the one seen in Figure 36, or they can be mostly comprised of a straight line.

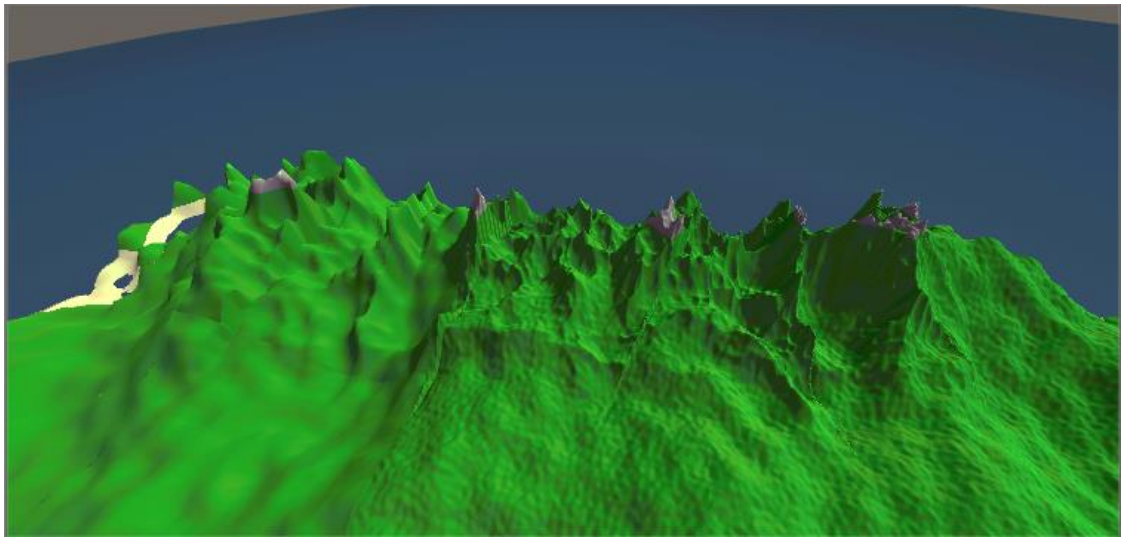

Figure 36 Screenshot of an incorrect boundary. The area here is the corner of the map

The raindrop agents have issues with navigating the terrain because the fine-grained Perlin noise adds a lot of local minima to the data. A fill-value was added to combat this. The fill-value would make the agent artificially higher than it was; this value would increase every iteration the agent spent stationary. However, it did not work as intended and seemingly caused agents to travel up hills.

3.2.5.3 Results

Generated terrain looks like it might occur in real-life, however, due to the lack of decoration it does look boring.

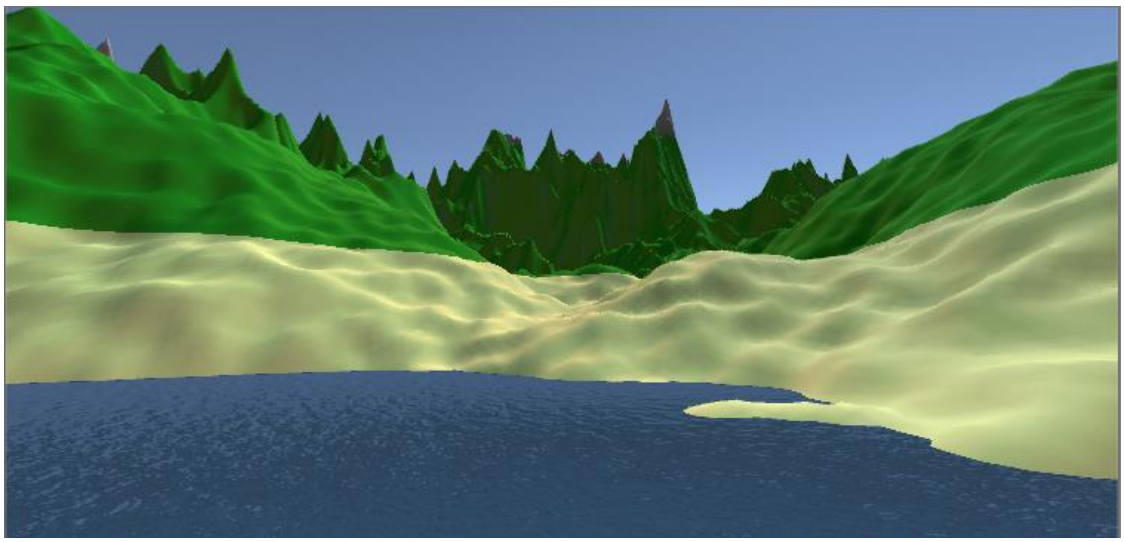

Figure 37 Screenshot of mountains seen from sea-level

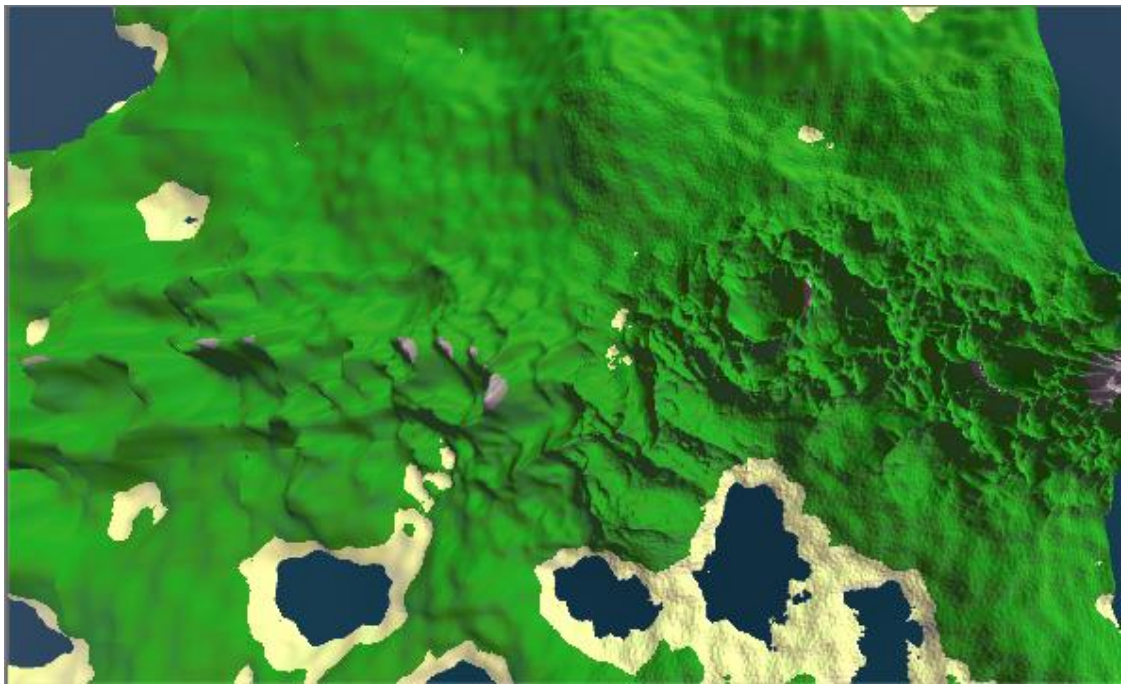

Figure 38 Screenshot showing how boundaries can snake across the terrain

In Figure 38 a boundary can be seen snaking across the terrain in a random manner. Looking from right to left, it is possible to see the effect of camera distance on the LOD of the meshes. The meshes on the right are at full level of detail, whereas on the left they are at their lowest level of detail.

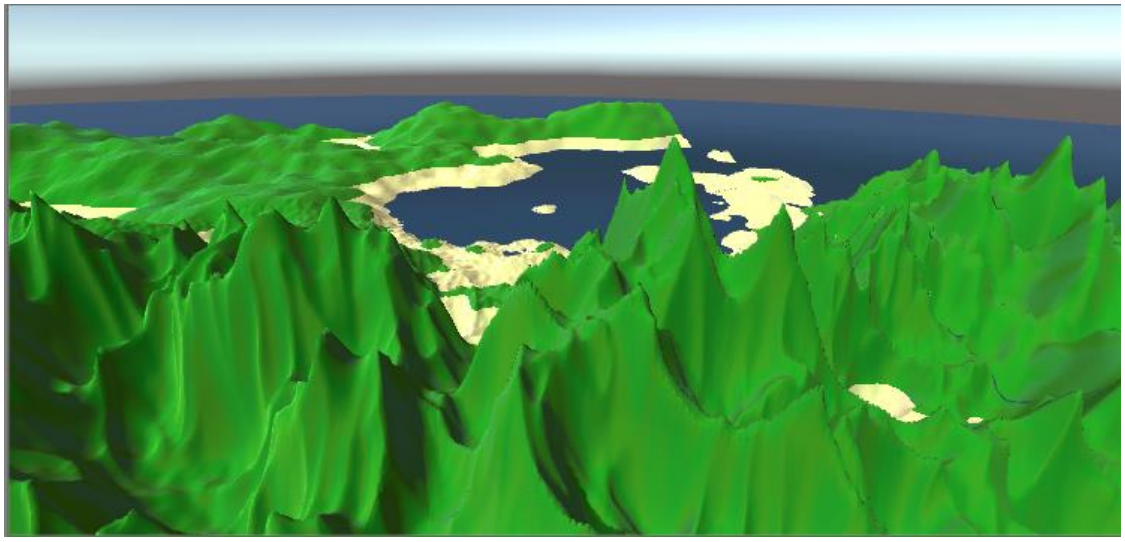

Figure 39 Screenshot showing mountains with normal octave count of 4

Figure 39 shows a typical mountain range generated by the generator. Currently, the octave count of the fractal noise, used to generated the mountains, is set to 4. By increasing the octave count to 8 the amount of 'detail' on the mountains also increases. See Figure 40.

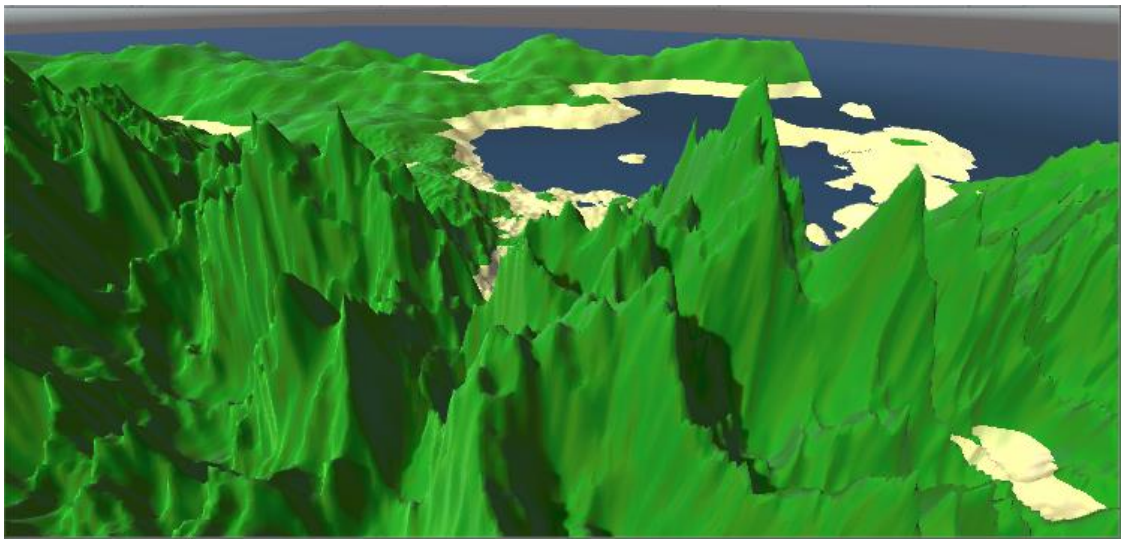

Figure 40 Screenshot of a typical mountain range with an octave count of 8

Conversely, by decreasing the octave count to 1, a reduction in detail can be seen. See Figure 41.

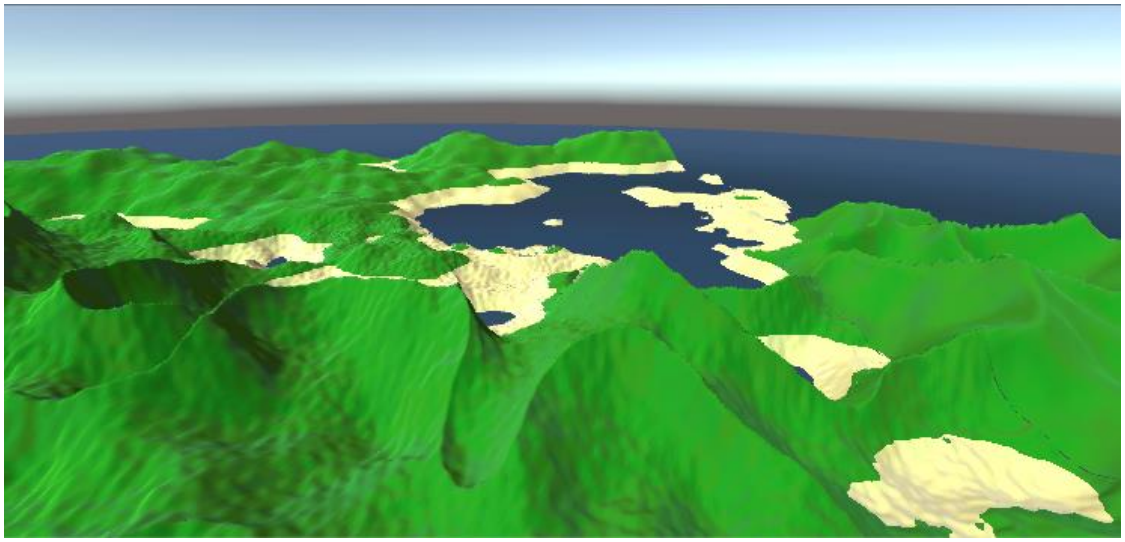

Figure 41 Screenshot of a mountain range with an octave count of 1

Figure 41 shows how useful these noise generators can be: with a few tweaks, the above ridges could look like sand dunes instead of a mountain range.

3.2.5.4 Efficiency

Shown below are average timings (ten for each size) at different sizes of maps with one boundary.

Table 1 Times for creation of worlds at various sizes

Map Dimensions (points x points)	**Average boundary creation time (ms)**	**Average height creation time (ms)**	**Total time (min:sec) [rounded to the nearest second]**
3615x3615	349	217,237	3:38
2410x2410	260	91,918	1:32
1205x1205	146	20,349	0:21
482x482	86	3,078	0:03

These timings show that the created generator is slow at creating moderately sized maps. It is likely that few players would wait over three minutes to load the game they are playing. It is also important to keep in mind that these timings are for terrain only, most games would require other components to their worlds. These components can only add to the generation time.

3.2.5.5 Testing

There are two principal areas to test in this project: code correctness and map quality. However, due to time constraints, no formal testing was carried out. The application was tested, by the developer, by repeatedly generating maps. This had the benefit of providing a way to calibrate the noise functions (by adjusting frequency, octave count, lacunarity, etc.) and also to check that meshes were correct and boundaries formed correctly.

Code correctness

Code correctness would be tested in the usual manner, unit testing input vs outputs to make sure the system does not critically fail.

Map quality

Map quality, on the other hand, is rather more complicated. It is not immediately apparent how to test, or even what to test for.

First, it is necessary to determine what characteristics a map must exhibit. This will largely be dependent on the game the map is being created for. In this project, maps should mimic real-life.

Next, it is necessary to construct appropriate tests. This is perhaps the most difficult part. Some requirements will likely be qualitative, e.g. is a map fun? Is a map interesting? It is not an easy task to test for these values in an objective manner. It is, of course, possible to test for these using humans. In fact, most studios will put their games through play tests at multiple stages of development. These play tests not only test for bugs, but test the mechanics (rules) of the game to see if it is fun and how difficult it is, and also test the artistic content of the game. An issue of using humans is that they need to be paid, which can be costly. An alternative could make use of AI methods to design agents to test maps. For example: if one were developing a maze generator, they could create an agent which attempts to solve said mazes.

As previously stated, in this project, the aim of maps is to mimic real-life. A good test would be to somehow compare generated maps with areas in the real world. This could possibly be done by comparing screenshots of the game to photographs of places in the real world, or, by taking terrain data of locations in the real world, and somehow comparing that with terrain data generated by this project.

Another difficult problem is determining how to interpret the results of the above tests. What ‘fitness’ value is acceptable? This could perhaps be calculated Mathematically, e.g. a map may need to be 95% ‘like’ the real world, or perhaps the threshold could be calibrated by the developer by examining results and accepting or declining worlds and thus determining a lower bound for the fitness value.

4 Conclusion

4.1 Summary

4.1.1 Procedural Generation

Previously, procedural generation played a relatively minor role in video games, past games used it to create content but given the simplicity of the games themselves, and the technology at the time, the results were limited.

With the advent of much more powerful processors, procedural generation can be used for more complex tasks. Procedural generation can greatly expand the scope of a video game as well as its re-playability. This coincides with the recent resurgence of 'indie' studios, who characteristically have few team members and few resources. Procedural generation has been especially useful to these types of developers.

Procedural generation also has uses in supplementing human imagination: artists can generate complete (or partially complete) worlds and modify them as necessary.

There are many modern video games utilising procedural generation, most notably, *Minecraft, No Man's Sky, and Elite: Dangerous*. All of which are mainstream hits. There are also non-game uses for procedural generation, such as *SpeedTree* which has seen use in CGI for feature films.

Procedural generation need not be used in large amounts, *Outerra* provides an example of how procedural generation techniques can be used to extend existing data to provide heightened realism.

4.1.2 This Project

This project has created a generator capable of generating simple terrain based upon physical processes. This generator shows the power of procedural generation; using simple methods, believable terrain can be easily generated using only two size parameters and a seed value. This project has also implemented a dynamic terrain system, allowing for the rendering of created data in *Unity*, a popular game engine/creation program used by many professional studios as well as hobbyist and indie studios.

4.2 Evaluation

This project made a good start but did not fulfil all its objectives. This project aimed to create a generator capable of generating worlds with terrain, water features, vegetation, and buildings. Only terrain was achieved. However, given the development cycle and resources available to

even small game teams, the lack of progress for a one-man team in the allotted time can be forgiven.

The generator created utilises a simple tectonic-plate model to create more believable terrain. This was moderately successful and indicates that, given more time, it is a viable path to take when designing a generator. The generator does, however, have issues of reliability, sometimes the boundaries created are almost a single straight line, or they end and terminate within a short distance.

This generator uses a fixed number of points, at fixed intervals. This is good when computing heights, but when it comes to displaying the terrain it can cause issues. For example: if the terrain is very steep between a small number of points, the mesh looks very sharp and unrealistic. This can be combatted by using fractal methods (or others) to 'fill in the gaps' like the process *Outerra* uses, or by increasing the resolution of data generated at the initial stages.

The time taken to generate worlds is longer than desired, it is likely that most players would be annoyed by the amount of time taken to generate worlds, especially given the lack of content in generated worlds.

This project does show that layering simple techniques provides good output. It also shows that even a simple physically based model can provide a convincing effect. However, it is not as convincing or complex as the worlds generated by *Elite: Dangerous* or *No Man's Sky*. Both games produce varied terrain, and perhaps most importantly, complete (a complete planet) terrain. Whereas this project only creates sections of terrain.

4.3 Future Work

The generator created here only uses converging boundaries. This can be extended by adding other types of boundaries, such as diverging, and creating the appropriate generator to model said boundaries.

Future works could also add more realism to the boundary simulation, this could be done by simulating actual tectonic plates rather than just boundaries. Tectonic plates could be assigned a height and velocity. These, in turn, can be used to determine the type of boundaries. Each boundary would have its own generator for generator specific terrain to that boundary, i.e. diverging boundary terrain would be characterised by a rift valley.

However, before adding to the generator it would be best to optimise it to decrease generation time, especially if the generator is meant to be used for online generation (on the player's machine).

The simulation of climate is a possible next step. Climate not only affects terrain, erosion, and rivers, but also dictates vegetation and the material-look of the terrain (e.g. rock vs. sand). This could possibly follow on to adding a Biome system like that of *Minecraft.*

The addition of vegetation, textures, and buildings adds great value to worlds from the player's perspective. Adding a biome system, like *Minecraft*, would greatly increase how interesting generated worlds are. Adding textures and biomes does introduce issues when differing biomes meet; it is not trivial to blend these to provide a seamless transition. However, as shown by *Sir, You Are Being Hunted* (with regards to the transition between beach and any other texture), depending on the type of transition, easy, albeit specialized, solutions can be found.

If buildings were present, the addition of roads would provide added realism. They would also provide an interesting problem: it would require the calculation of paths between buildings/cities. This would be done as to minimise cost (length of road) and the effect on the terrain, just as it might be done in real-life.

Testing procedural generated worlds is critical given how large a part of games they are. Developing appropriate tests is dependent on the game for which the maps are created, however, more work is required in this area. It could be required that maps must be traversable, this could be tested by developing agents to walk across the map, testing if there are any points at which they get stuck or cannot access.

5 References

[1] gharen, “Home page for LibNoise for .NET,” [Online]. Available: https://libnoisedotnet.codeplex.com/. [Accessed March 2017].

[2] S. Lague, “Youtube playlist titled "Procedural Landmass Generation (Unity 5)",” [Online]. Available: https://www.youtube.com/playlist?list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3. [Accessed September 2016].

[3] “Home page for Unity 3D,” Unity, [Online]. Available: https://unity3d.com/. [Accessed March 2017].

[4] G. N. Y. k. O. S. C. B. Julian Togelius, “Search-based Procedural Content Generation: A Taxonomy and Survey,” *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, ,* vol. III, no. 3, pp. 172-186, 2011.

[5] “SpeedTree Homepage,” Interactive Data Visualization, Inc., [Online]. Available: http://www.speedtree.com/. [Accessed March 2017].

[6] “Wiki page for Rogue (Game),” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Rogue_(video_game). [Accessed October 2016].

[7] “Wiki entry for Elite (Game),” Wikipedia, [Online]. Available: https://www.en.wikipedia.org/wiki/Elite_(video_game). [Accessed October 2016].

[8] “Game page for Borderlands video game,” Gearbox Software, [Online]. Available: https://borderlandsthegame.com/game/borderlands. [Accessed October 2016].

[9] E. W. Weisstein, “Fractal,” Wolfram Research, Inc, [Online]. Available: http://mathworld.wolfram.com/Fractal.html. [Accessed April 2017].

[10] “Home page for Minecraft,” Mojang AB, [Online]. Available: https://minecraft.net/en-us/. [Accessed March 2017].

[11] AB, Mojang, “Minecraft Sales Figures,” 22 9 2016. [Online]. Available: https://mojang.com/2016/06/weve-sold-minecraft-many-times-look/.

[12] “Biome Wiki entry,” Minecraft Wiki, [Online]. Available: http://minecraft.gamepedia.com/Biome. [Accessed October 2016].

[13] M. Persson, “Terrain Generation part 1,” [Online]. Available: http://notch.tumblr.com/post/3746989361/terrain-generation-part-1. [Accessed October 2016].

[14] “Home page for Sir, You Are Being Hunted,” Big Robot, [Online]. Available: http://www.big-robot.com/tag/sir-you-are-being-hunted/. [Accessed March 2017].

[15] Big Robot, “Official blog post on the procedural generation in Sir, You Are Being Hunted,” Big Robot, 2 July 2012. [Online]. Available: http://www.big-robot.com/2012/07/02/procedural-british-countryside-generation/. [Accessed October 2016].

[16] T. Betts, “Video of a GDC Lecture titled 'Creating FPS open worlds using procedural generation,” Big Robot, [Online]. Available: http://www.gdcvault.com/play/1020340/Creating-FPS-Open-Worlds-Using. [Accessed October 2016].

[17] R. Pless, “Lecture notes on Fortune's Algorithm and Voronoi Diagrams,” Washington University in St. Louis, [Online]. Available: http://www.cs.wustl.edu/~pless/546/lecture/L11.html. [Accessed October 2016].

[18] D. Stapleton, “Sir, You Are Being Hunted Review,” IGN, 1 May 2014. [Online]. Available: http://uk.ign.com/articles/2014/05/02/sir-you-are-being-hunted-review. [Accessed March 2017].

[19] “Home page for No Man's Sky,” Hello Games, [Online]. Available: https://www.nomanssky.com/. [Accessed March 2017].

[20] H. McKendrick, “Video of a lecture at nucl.ai conference titled 'Building A Galaxy: Procedural Generation in No Man's Sky',” Hello Games, 2015. [Online]. Available: https://archives.nucl.ai/recording/building-a-galaxy-procedural-generation-in-no-mans-sky/. [Accessed October 2016].

[21] A. Smith, “Article titled: RPS Verdit: No Man's Sky,” Rock Paper Shotgun, [Online]. Available: https://www.rockpapershotgun.com/2016/09/07/no-mans-sky-verdict/. [Accessed October 2016].

[22] J. Walker, “Article titled 'No Man's Sky: Why am I still Here?',” Rock Paper Shotgun, 13 September 2016. [Online]. Available: https://www.rockpapershotgun.com/2016/09/13/no-mans-sky-diary/. [Accessed October 2016].

[23] J. E. Webber, “No Man's Sky Review: beautifully crafted galaxy with a game attached,” The Guardian, 12 August 2016. [Online]. Available: https://www.theguardian.com/technology/2016/aug/12/no-mans-sky-review-hello-games/. [Accessed October 2016].

[24] “Home page of Outerra,” Outerra, [Online]. Available: http://www.outerra.com/. [Accessed March 2017].

[25] “Features of Outerra,” Outerra, [Online]. Available: http://www.outerra.com/wfeatures.html. [Accessed March 2017].

[26] “Demo page for Outerra,” Outerra, [Online]. Available: http://www.outerra.com/demo.html. [Accessed March 2017].

[27] “Home page for Elite: Dangerous,” Frontier Developements, [Online]. Available: https://www.elitedangerous.com/. [Accessed March 2017].

[28] J. Williams, “How Planets are Made in Elite: Dangerous,” wccftech, 5 October 2016. [Online]. Available: http://wccftech.com/planets-elite-dangerous-horizons/. [Accessed March 2017].

[29] “Procedural Generation page on Elite: Dangerous Wiki,” [Online]. Available: http://elite-dangerous.wikia.com/wiki/Procedural_Generation. [Accessed March 2017].

[30] ObsidianAnt, “Youtube video titled "Elite: Dangerous Horizons - Deep Space Landings - What's on those Planets? ",” 8 December 2015. [Online]. Available: https://www.youtube.com/watch?v=Dq7sPryJf1s. [Accessed April 2017].

[31] “Home page for Git,” Git, [Online]. Available: https://git-scm.com/. [Accessed March 2017].

[32] “Home page for Irrlicht,” [Online]. Available: http://irrlicht.sourceforge.net/. [Accessed March 2017].

[33] “Home page for Ogre3D,” [Online]. Available: http://www.ogre3d.org/. [Accessed March 2017].

[34] “Home page for OpenGL,” Khronos Group, [Online]. Available: https://www.opengl.org. [Accessed March 2017].

[35] “Home page for Unreal Engine,” Epic Games, [Online]. Available: https://www.unrealengine.com/. [Accessed March 2017].

[36] “Home pafe for FLTK,” [Online]. Available: http://www.fltk.org/index.php. [Accessed March 2017].