Infinite Terrain Generation Using Voronoi Diagrams

THE UNIVERSITY OF WAIKATO

ENGG492-Y – Honours Research and Management Project

10/16/2017
Supervised by Bill Rogers
Andrew Leach – ajl36 -1137001

Abstract

Games today which contain worlds spanning thousands of kilometres of terrain (environment) hold a burden on the computers resources in terms of computation and storage, especially when the terrain is generated at run-time.

It is a desirable attribute for a game to handle terrain of this size (for use in simulators or exploration based games), while needing no long-term storage on the computer and therefore removing any bottlenecks of storage or network usage (by sharing terrain in a multiplayer context).

To remove any obvious regularities in the generated terrain, the direct use of Voronoi Diagrams will be explored as the base structure of the terrain in an infinite context. As Voronoi Diagrams are inherently meant for finite area construction, the possibilities to remove this limitation will be investigated.

This report outlines the approach of generating an infinite (only limited by numeric precision) terrain generator at run-time that requires no long-term storage, is applicable for game-usage, and generated a realistic environment.

Table of Contents

1 – Introduction

Procedural Terrain Generation, referring to the generation of the playable surface within a games world, is a topic of great interest as creating a terrain which has the ideal balance between performance, realism and interest is a near impossible task to achieve. Techniques such as Perlin Noise [1] and Fractals [2] have become kings in today's age, popularized heavily by their hybrid usage in such games as "Minecraft" [3], "No Mans Sky", "Just Cause", "Elder Scrolls II" and others [4]. These games make use of proven techniques, yet lack a repeatable and identical deterministic nature in their generation. These games have very large terrains, spanning thousands of kilometres and due to the level of detail of the terrain the storage requirements on the computer is intensive. This puts heavy strain on system resources, in both requiring the storage of the generated terrain as well as in a multiplayer context, the sharing of the generated terrain over the network which may cause bottlenecks.

It is easy to overlook the need for larger storage for a games terrain as the fact that hardware capacity progresses so quickly, which has caused a stagnated approach to address the issue. Having a deterministic terrain generator, in the sense that any segment of the terrain can be re-generated and be identical each time it is generated, allows for very large or "infinite" terrains to be generated without the need for any storage on the computer.

The terrain in games however must contain interesting geological features, such as large flat areas for building and battles, mountain ranges to explore, and a variety of contrasting environmental features to keep the terrain interesting over a long period of exploration. The reason for the need to generate very large worlds is to be able to incorporate many players, and allow for various modes of transport: such as fast flying planes, or teleportation to far away areas.

Thus, the motivation is to explore the possibility of creating a terrain generator which has the desirable properties of being deterministic while maintaining the larger scale properties (terrain features which extend much further past the users view).

The aim is to create a terrain generator which allows the theoretical infinite construction of connected terrain while having the desirable deterministic property and being able to generate a unique and vast terrain holding realistic and large-scale structures. This terrain must contain geological areas which apply directly to game-usage, such as large flat areas, lakes and mountain ranges. To add variety to the base structure of the terrain and add a more realistic feel, irregular polygons will compose the smallest units of this generated terrain.

This report will outline the process undertaken during the project, the successes and failures of a variety of techniques tested, and the issues encountered with both the framework and language of choice for the development of this project.

2 – Related Work

2.1 – Grid-based Terrains

Grid-based terrains are the bed-rock of many large-scale terrain generation algorithms, such as in "Minecraft" [3]. Grid-based terrain means the systematic breakdown of terrain in to small, identifiable and regular shaped chunks which create the larger world. These are usually squares or hexagons, and are extremely popular to implement as it makes it easy to identify and manipulate. However, although simple to implement, it has drawbacks if not correctly implemented as it can lead to visible artifacts being formed in the terrain if each individual shape is too large; or if it is too small, performance starts to suffer.

2.1.1 – Irregular Grids using Voronoi

In recent years, investigation in to the use of non-regular shapes as the basis for grids has occurred for terrain generation, specifically, the use of non-regular polygons [5]. The common application for generating these non-regular polygons come from a diagram known as a "Voronoi Diagram".

Voronoi Diagrams are generated from the input of a non-zero set of points of size **N** and will produce **N** irregular polygons, with a single point inside each polygon [6]. Figure 1 shows a standard Voronoi Diagram output, containing **N** red dots (the input set of inputs) surrounded by **N** irregular polygons.

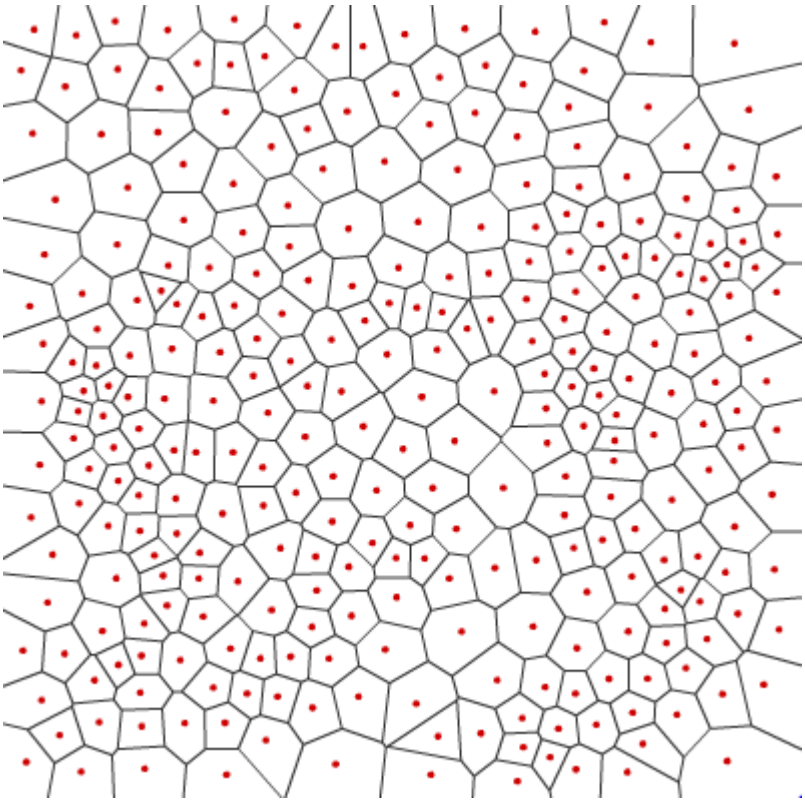

Figure 1 - A Voronoi diagram, with the red dots representing the set of points as input [5]

The art of constructing this diagram can be done in O(nlogn) time, and O(n) space (both worst case bounds) assuming 2D scenarios, where 'n' is **N** (the size of points passed in). These bounds are achieved using Fortunes Algorithm [6], which performs a line-sweep over the points to generate the

diagram. As Voronoi Diagrams are inherently built off random numbers, it is possible to generate it deterministically if the same numeric seed value is used given a deterministic pseudo-random number generator. The use of deterministic pseudo-random generators is a powerful tool as given the same *seed* it will always generate the same sequence of numbers [7]. This seed is a numeric integer.

Figure 2 shows how the line sweep is performed during the execution of Fortunes Algorithm over a set of points. It works by creating parabolas based on each of the points towards the sweeping line (left to right in this instance) and then generating the polygons based on these parabolas. It is clear that, the polygons which are already created on the left-hand side will be unaffected if the points on the right are now changed. This is a property of Fortunes Algorithm which may be exploited to extend the generation to larger areas.

Although Fortunes algorithm is the most efficient algorithm for producing 2D Voronoi Diagrams, there exists no published algorithm that allows the arbitrary and seamless addition of neighbouring Voronoi diagrams in any direction. This means that no practical “infinite” Voronoi diagram algorithm exists, and that the standard algorithms are designed to be produced in a finite area.

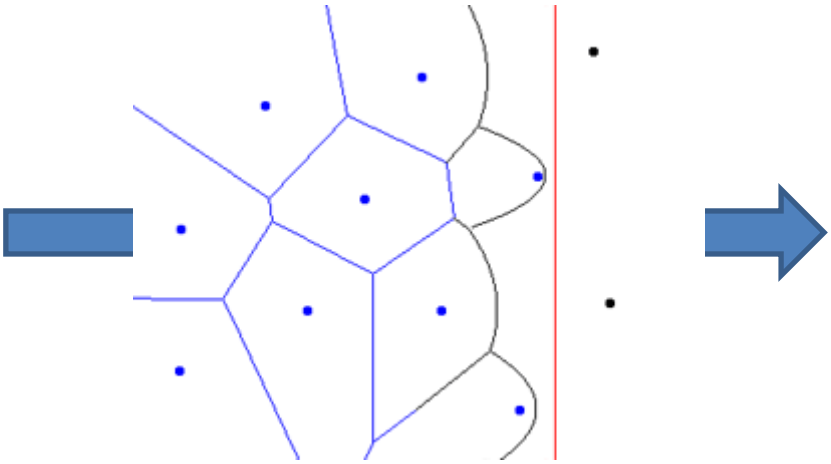

Figure 2 - A picture showing the line sweep performed to produce irregular polygons during Fortunes Algorithm

The idea here for using irregular polygons, opposed to regular shapes (such as squares) is that it removes any obvious pattern in terrain generation, by adding extreme variation in the basic structure of terrain. The drawback is the added complexity in implementation and handling of the irregular shapes.

Voronoi Diagrams have potential to be used as the bed-rock to a terrain generator, as this would produce an alternative basis for terrain. Investigation for this project would have to be made in to potentially extending this to be used in an infinite fashion and to check the results of using irregular polygons as the basis of the terrain.

2.1.1.1 – Polygon Regularity

Further alteration can be done in affecting how the irregular polygons are formed during the Voronoi diagram construction using Fortunes Algorithm [6]. Often erratic and very sharp changes between polygons form undesirable effects within a terrain [5] and therefore work can be done to make these more regular while still retaining uniqueness. There are two well-known ways to create

more well-formed polygons: Lloyds Relaxation Technique; and by averaging out the distance between points in the input set.

Lloyds Relaxation is a technique used specifically in conjunction with Voronoi Diagram construction, with the purpose being of creating more regular polygons [8]. It works by doing **K** iterations (with a higher number producing more regular results) of the Voronoi Diagram algorithm of choice, each time using the centroids of each produced polygon in the previously generated diagram as the basis for the set of input points, which then is used to create another diagram with (the old diagram is discarded). . Generally K is very small (**less than 5**) and therefore does not significantly influence the running time. A direct contrast between using Lloyds and not (Figure 3) show drastic results with as little as two iterations, given the same input size. It is worth noting that Lloyds will retain a deterministic nature assuming the same seed is used to repeat the process.

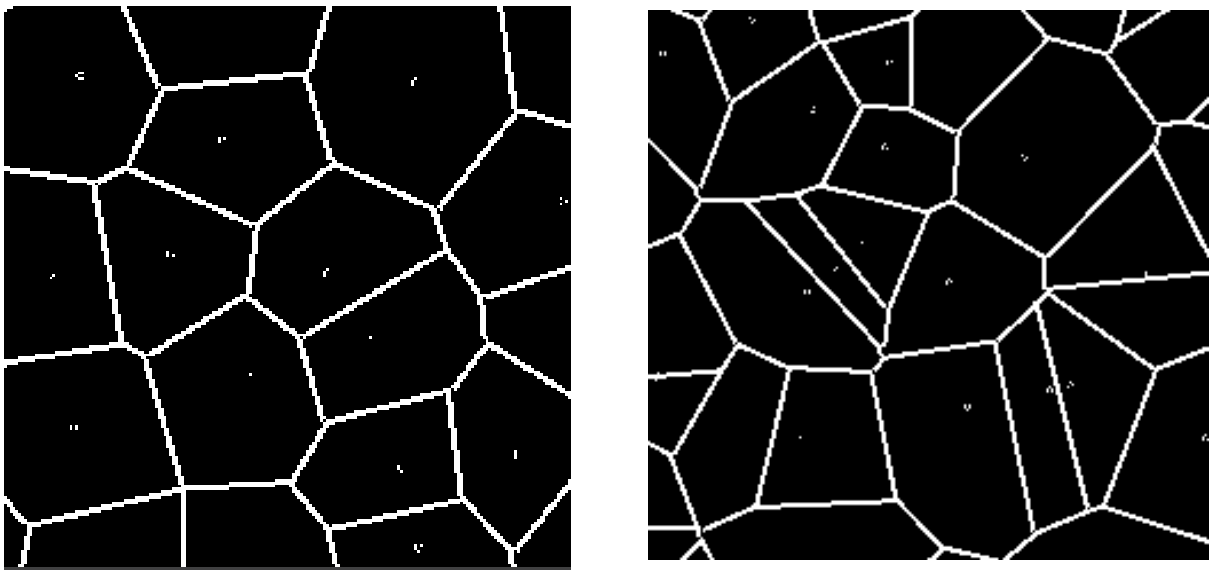

Figure 3 - The comparison of Voronoi Diagrams produced with Lloyds Relaxation (on the left) vs without (on the right). The K value used for Lloyds was 2

Another method to produce more regular polygons are based on creating more regular point distribution to generate the diagram from. Averaging the distance out between points can be done in a host of ways with the most effective technique being Poisson Disk Sampling [9]. It works by generating new points around existing points, which have at least a minimum distance of D, in turn creating a distributed output set. Poisson can be done in O(**k**n) time, where **k** is very small (less than 30), and produces a set of points which are more evenly distributed the larger **k** is (Figure 5).

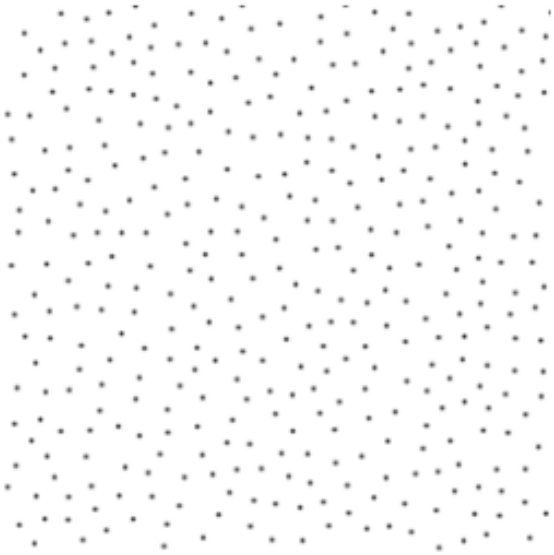

Figure 4 - Points generated from Poisson Disk Sampling with K = 30

This technique has become the concrete improvement over the old technique named “dart throwing” which had a complexity of O(n^2) [9]. This has become a popular method due to its deterministic nature, effective results and its linear running time.

Being able to alter the regularity of the polygons has direct application within this project as it directly effects how the terrain will form. It will drastically change the base structure, and therefore be able to create drastically different terrain just by modifying the regularity. Both Lloyds and point-distribution techniques should be investigated along with their results to compare in practice.

2.2 – Height Map Generation

The generation of height maps refers to the creation of depth (height – the Z axis) of the 2D terrain, thus directly transforming it in to 3D. Some current techniques which have been popularized due to their simplicity in understanding and implementation while retaining effective results are Perlin Noise [10] and Fractals [10].

2.2.1 – Perlin Noise

Perlin noise is a "noise" function which creates an arbitrary sequence of pseudo-random numbers [1]. However, Perlin Noise has a made a fantastic match for game-usage due to its "smooth" transitions between these numbers; the fact that it is deterministic; and fast in computation. Figure 5 shows a practical usage of Perlin Noise that produces a "smooth" transition of pseudo-random numbers amongst regions.

Figure 5 - Practical Perlin Noise Example

As Perlin Noise is both deterministic and fast in computation, it may have direct uses as a component within the projects final algorithm for generating a smooth transition of values over the worlds map. Some practical implementations may include a rain-fall map over the terrain.

2.2.2 – Fractals

Fractals are another technique which involves repeating many nested simple shapes to produce complex terrains with refined detail [10]. The idea is that self-similarity amongst repeated shapes (of varying size) mimics well the natural processes in the real-world such as erosion and plant growth. It works by subdividing a shape of choice (such as a triangle) into many nested triangles inside a larger one. By repeating this process, unlimited refinement of detail in a region is possible.

2.2.3 – Erosion

Erosion is a broad term used to define a technique that helps to mimic the change of a terrain over time due to the influence of an external (generally real-life) factor. The idea is that terrain is

influenced over many years due to external factors, and modelling this will help to produce more realistic terrain. Erosion comes in many forms, but often it will be specific to water-flow, in particular how rain influences a terrain [10].

The core principle of erosion is the displacement of some "mass" in the terrain from one area to another, and therefore inherently, erosion is difficult to implement when considered in terrain which is generated in real-time.

2.2.3.1 – Rain Erosion

Rain-based erosion is popular due to its proven success. However, it is intensive computationally and difficult to implement, which often leads smaller terrain generators to shy away from such an algorithm. Figure 6 shows the transition of the same terrain before and after rain-based erosion, which helps to generate a more realistic mountain range from a rough terrain [11].

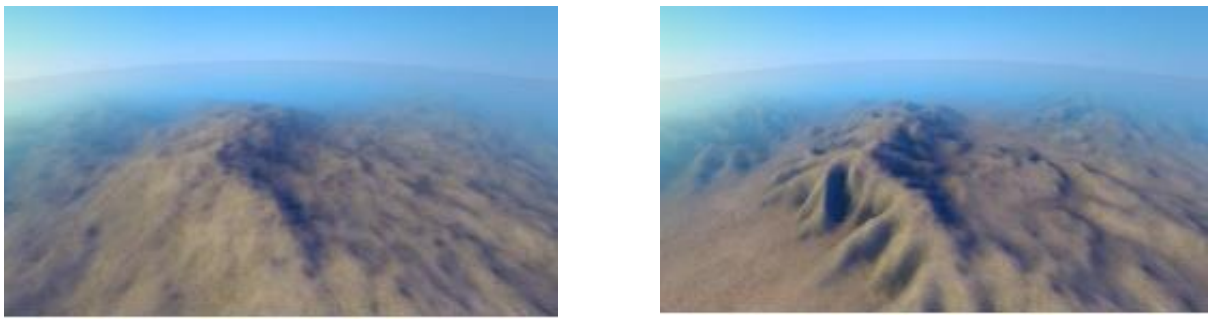

Figure 6 - A terrain before rain-erosion (on the left) and after (on the right) [10]

As can be seen, there is considerable success in the ability to transform a terrain from a rough-outline in to something more natural and realistic by simulating how the flow of water over terrain would mould it.

Erosion has potential in this project to create more realistic looking terrain, however, investigation would have to be put in to ensuring a seamless integration with an infinitely extending terrain.

2.2.4 – Primitive Shapes

Primitive shapes is defined by the use of very simple shapes, such as circles, cones, lines, to be placed over the terrain and influence the heights of the surrounding environment. The idea behind this is that terrain is shaped from very strong geological factors, such as meteors or tectonic plate movements, which leave craters in the earth, large hill gradients or mountain ranges. It is because of this that the use of primitive shapes helps create a "raw" terrain.

However, the use of primitive shapes is difficult as often tricky implementation or the need for post-processing (such as water erosion) to help mimic how these raw influences were shaped over hundreds of years.

Success has been seen in many finite-area terrain generators when coupling both primitive shapes and erosion techniques. Seen in Figure 7, a fantasy map generator has been produced using primtive shapes of cones, circles, and gradients (to mimic tectonic plates) [12]. The use of a water-based erosion mapped water flow downwards to sea. Although successful, this project does not extend easily to make a infinite extension.

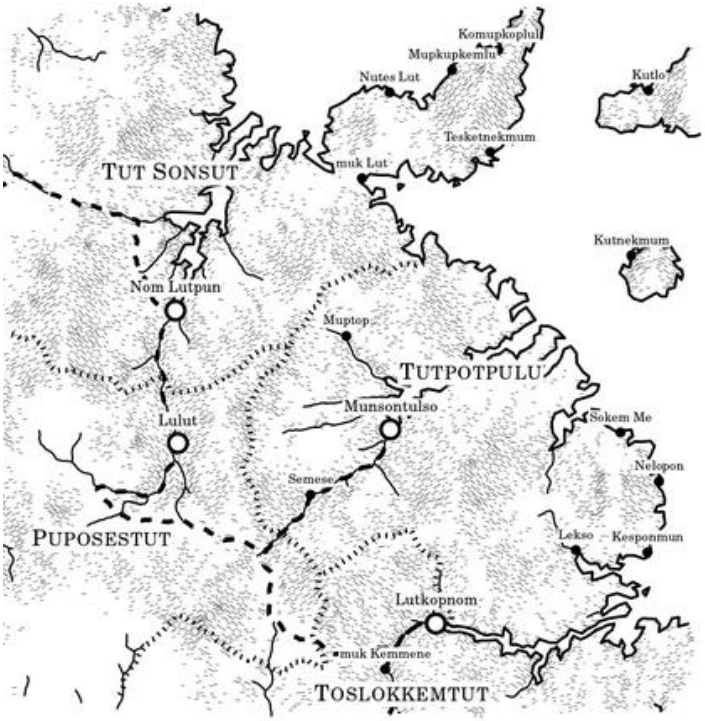

Figure 7 - Primitives with Erosion to produce a fantasy map [12]

Primitives are deterministic in nature to place, and very computational cheap. Depending on the primitives chosen, this has potential due to the fact it scales easily with an infinitely expanding terrain.

2.3 – Biomes

Biomes play a crucial role in the generation of interesting terrain for game-usage, as they add variety and a reason to explore, with different biomes containing different forestry, and other features. Biomes in the real-world exist, and are a product of two main factors: temperature and precipitation [13].

We can see in Figure 8 the emergence of a particular biome type based on an areas' precipitation and temperature.

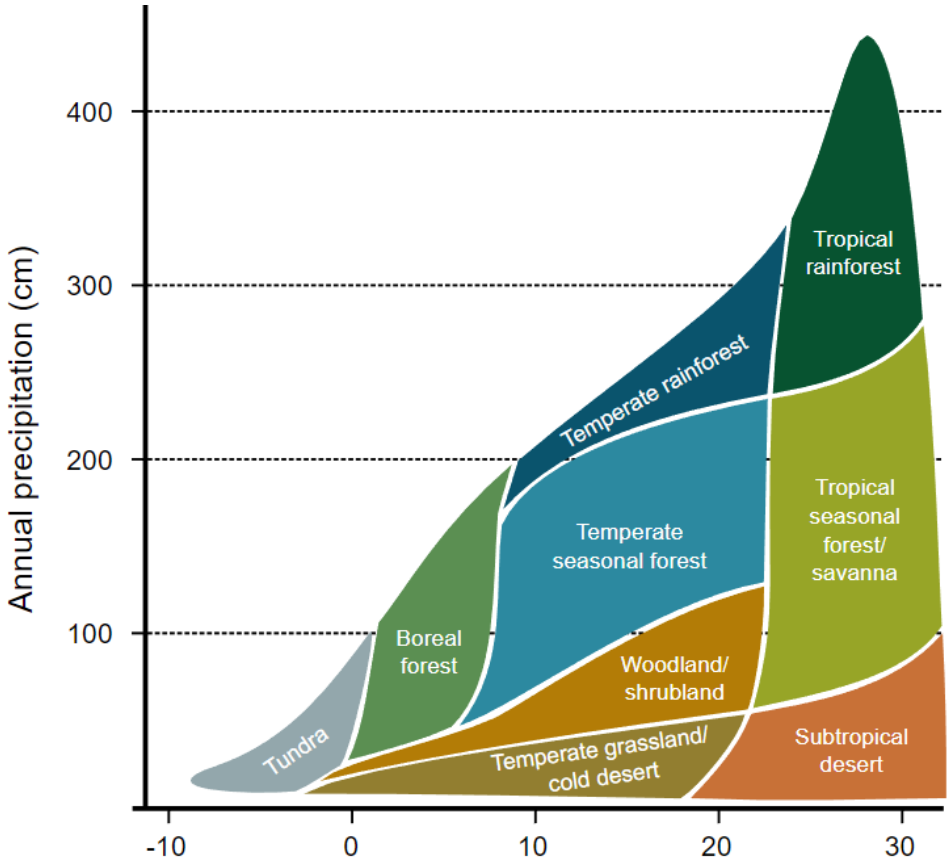

Figure 8 - Classification of a biome based on precipitation vs temperature. [13]

2.3.1 – Temperature and Precipitation

Temperature and precipitation are both hard aspects of a game to accurately generate for an area, between being realistic and interesting to create a proper spread of biomes. Lack of variation will make a dull terrain with no changes in biome, whereas erratic changes will cause too many biomes.

An issue between not generating smooth transitions between temperature and precipitation also needs to be considered as only related biomes should be near each other. Often, this leads to the use of Perlin Noise for the generation of precipitation and temperature mappings due to the smooth transition and deterministic nature of such a function [10].

Temperature, however, is easier to determine for a region based on the different in the regions height to sea-level. Based on real world statistics, on average temperature falls 9.8 degrees C per 1000 meters above sea level [14]. This can be used in influencing temperatures based on elevation.

Precipitation is not as easy to map realistically, due to the nature of the game terrain, there are many factors which contribute to precipitation which may not be implemented.

Biomes are an easy thing to implement, assuming the temperature and precipitation are mapped well, while providing a solid basis for variety within a generated terrain. Temperature and precipitation have potential to be deterministic and computational cheap (through methods such as Perlin Noise) and therefore allow a pathway to implementing biomes in to an infinite terrain.

2.4 – Consideration Techniques

2.4.1 – Performance

The idea that the terrain is intended for game-usage, and can infinitely extend in any direction in real-time, means that the algorithm must be fast enough that it will not cause a noticeable delay in the application and interrupt the gaming experience.

It is shown that 0.1s is the limit before a user will notice any real delay, and that 1.0s is the limit before their flow of thought is interrupted [15]. Therefore in the consideration of an infinite, real-time terrain generator, is it important to account for these limits and ensure that the generation of near-by terrain takes less than 0.1s.

3 – Approach

3.1 – Development Environment

The choice of development environment was biased due to my previous experience with Microsoft .NETs C# programming language; as due to time constraints and difficulty of this project, it was not justified to learn a new language for this cause. MonoGame [16] is a powerful game development engine which will be used alongside C# to produce this project. MonoGame is the open-source replacement of the Microsoft XNA framework [16] containing broad support for 3D rendering projects, perfect to develop an open-world terrain in. The rendering system within MonoGame is based on a X,Y,Z coordinate system, with the X and Y values modifying in the nature seen in Figure 9.

MonoGame has native support for compiling down its projects to be used on the Windows, Linux and Mac operating systems [16].

3.2 – Deterministic Nature

To ensure a deterministic nature in terrain generation while allowing the regeneration of a specific world when desired, all the pseudo-random number generators used must be based off the same numerical seed as this will in turn ensure all the algorithms which are used in the generation to produce the same results each run. This guarantees repeatable and identical results when needed, regardless of the player or instance invoking the call to generate terrain. Because of this, all worlds generated will have a numerical seed associated with them, this also means that the seed will be required to be shared amongst all players of the same world.

3.2.1 – Terrain Grid Overlay

All the algorithms explored within the background were deterministic at their core, yet often limited to a region, such as Fortunes Algorithm. Knowing that the algorithms are deterministic, but for a specific region, it is possible to exploit this nature by creating a simple square grid-structure over the entire terrain. Each grid cell will be a perfect square, with a width and height **P**. The grid cells will be assigned an X and Y value, as if they were in a 2D array, except X and Y may be negative (meaning X and Y are elements of the integer set of values **Z**). Their values increment in the fashion shown in Figure 9.

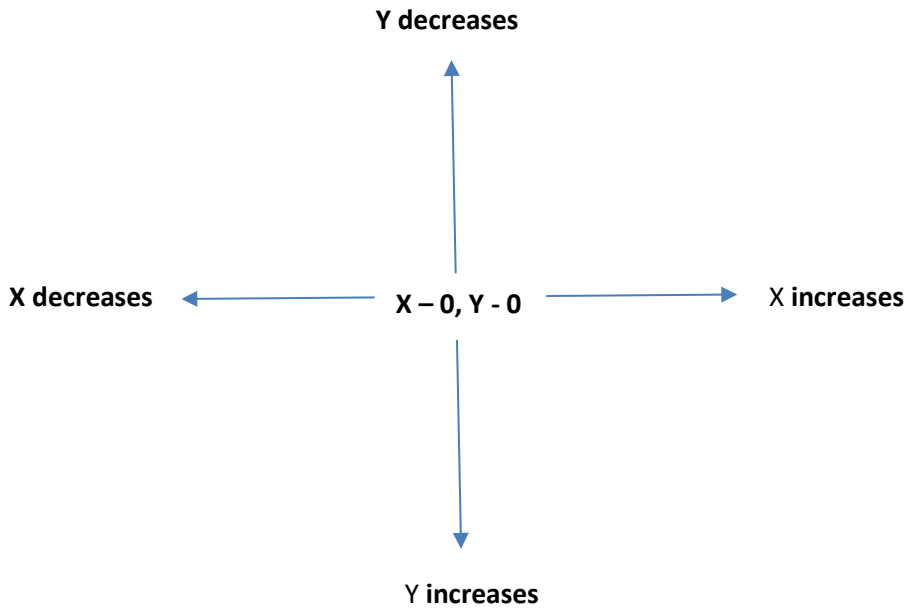

Figure 9 - Grid coordinate system

We can treat each individual grid cell as a region, and perform deterministic procedures on each grid using its coordinate in space (X and Y, as it is a 2D grid) as the basis of its numerical seed to ensure repeatable and identical results. We can map together these numerical values (X and Y) to a single numerical value, using the Cantor function [17] which allows a unique and deterministic mapping function of two numerical values to a single value. In practice, this requires that the two-numerical values map to a single value which has twice the number of software bits as first two. For example, in the example of C#, the *short* variable type is built on 16 bits while the *integer* variable type is built on 32 bits [18]. This means that using the Cantor function, we can uniquely and deterministically map two *short* variable types to an *integer* within C#. However, two *integers* cannot be mapped to a single *integer* correctly as we require the output to have twice the bits and this creates a restriction we must consider when implementing the project.

We can repeat this process again, by mapping the grids' X-Y single value together with the original numerical seed assigned with the world, to get the final seeds value for a given grid cell. This now means that within that cell we can get truly deterministic and identical results every time, and can base the generation algorithms at a cell-by-cell level, yet still allow to have different worlds.

3.3 – Voronoi Diagrams

The choice in the smallest unit of the terrain, although seemingly easy, has the largest effects on the overall structure and how realistic it looks and feels. Having a natural looking terrain starts by ensuring what it is made from mimics the real-world, and therefore the choice of using irregular polygons was made to try hide any regularity to give a less generated feel (An extreme case is "Minecraft" with its obvious square blocks (Figure 10)**)**

Figure 10 - Terrain generated within Minecraft, showing obvious squares as its basis. Source: http://www.minecraftforum.net/forums/mapping-and-modding-java-edition/minecraft-mods/1291067-atg-alternate-terrain-generation

Background research shows how Voronoi Diagrams are a powerful and proven technique used to quickly and deterministically generate a set of irregular polygons within a region. The issue with using Voronoi Diagrams regarding the projects aim, is that Voronoi Diagrams are designed to be constructed in a single finite area without natively allowing for further seamless extension in one of the following directions: up; down; left; and right.

3.3.1 – Voronoi Diagrams

The use of the grid-system explained in the previous section allows us to exploit the nature of Voronoi Diagrams and, without changing the overall time complexity, seamlessly generate additional Voronoi Diagrams in each of the four directions that connect correctly. Seen in Figure 11 we have a basic grid structure overlay consisting of (red visualized) squares. These squares each have generated, independently of other cells, points consisting of pseudo-random X and Y values within their square region, shown as white dots, using a pseudo-random generator with the seed being based off the worlds seed merged with their coordinate seed as described in section 3.2.1.

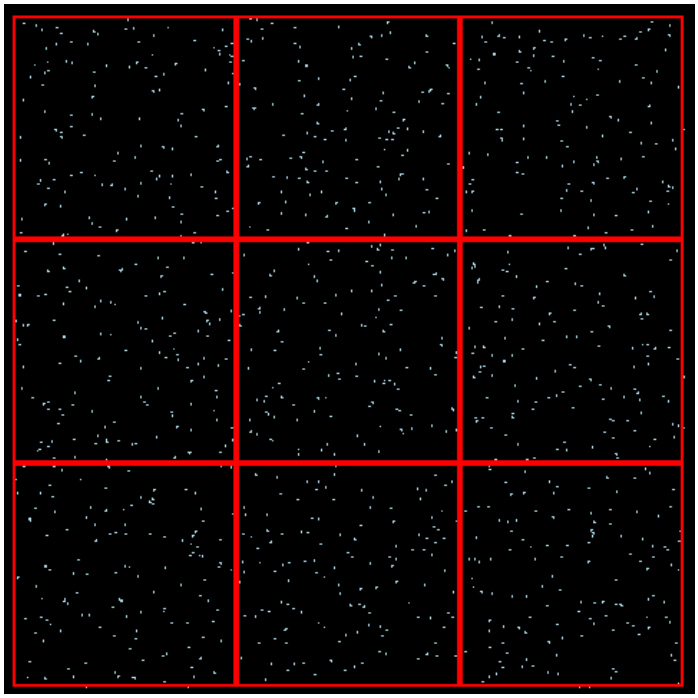

Figure 11 - Grid overlay for generating repeatable results at a large scale, shows generated points within each cell using their coordinates as the numerical seed

As we can generate these points in a deterministic and repeatable fashion, we can now for each grid-cell region produce a Voronoi Diagram also in a deterministic and repeatable way as the points are the input to the diagram. The core issue here, however, is that producing them independently causes a non-seamless connection in neighbouring cells as seen in Figure 12.

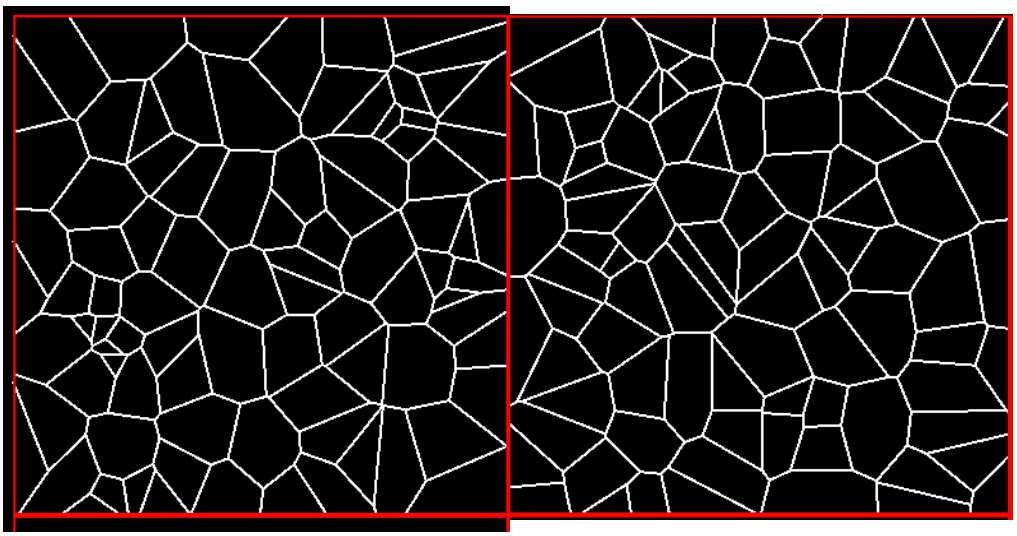

Figure 12 - Two Voronoi Diagrams produced side in neighbouring cells, showing the lack of seamless connection

These Voronoi Diagrams will be produced using Fortunes Algorithm [6] as this currently is the most efficient algorithm to produce Voronoi Diagrams in 2D space. Having neighbouring cells not seamlessly connect (Figure 12), would cause severe and obvious tears in the resulting terrain as the basic blocks would tear at the "borders" of these grids. This would therefore in turn show obvious square-based grids in the resulting terrain, making the push for an irregular polygon based terrain pointless as a way of hiding regular shapes.

It is possible to exploit this grid-structure, while still using Fortunes Algorithm (to maximise speed), to remove this limitation of tearing, and therefore allow the theoretical infinite seamless construction of connected Voronoi Diagrams.

Fortunes Algorithm is a sweep-line algorithm, which defaults to sweeping left to right, iterating over all the points and creating parabolic relations between all the points. This will result, once the sweep is complete, in a Voronoi Diagram. The effect of using the sweep-line within Fortunes Algorithm (seen on the left in Figure 13) has the property that certain polygons (within the yellow rectangle) will not be affected in shape regardless of how the edge points on the right change. This means the very edge points (points outside of the yellow square on the right in Figure 13) in the grid cell have the effect of only changing then outside polygons, while having minimal to no effect on the polygons produced within [6]. Therefore, if we can guarantee that points always exist outside of this yellow rectangle, the inside polygons will always be the same. This is the principle used in the extension for seamless connection.

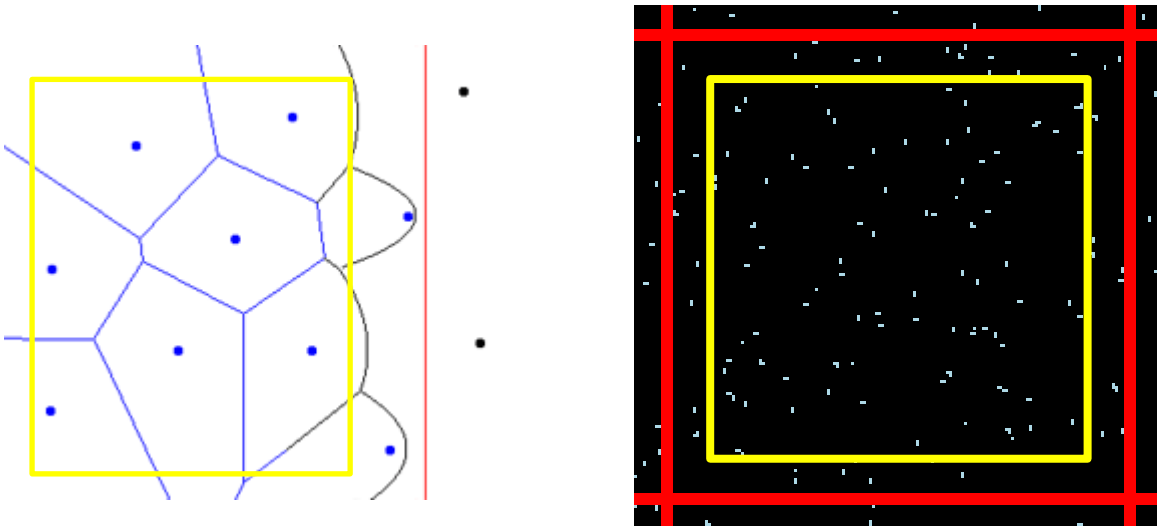

Figure 13 - Voronoi sweep-line effect on the left is exploited to ensure a region of polygon will be unchanged on the right.

This property however only works if there are enough points, and these points are evenly distributed well enough. If the points are not well distributed, or there are not enough (say, 2 per cell) this property will no longer work. This issue will be covered more in detail later under "Point Generation".

3.3.1.1 – Arbitrary Infinite Voronoi Extension

By exploiting this property now known due to the sweep line, seen in Figure 13, we can enlarge this property to multiple cells. If we take a 3 by 3 grid section, from the grid-overlay, and then generate a single large Voronoi Diagram which includes all the points of these cells, we will get a result like what we see in Figure 14.

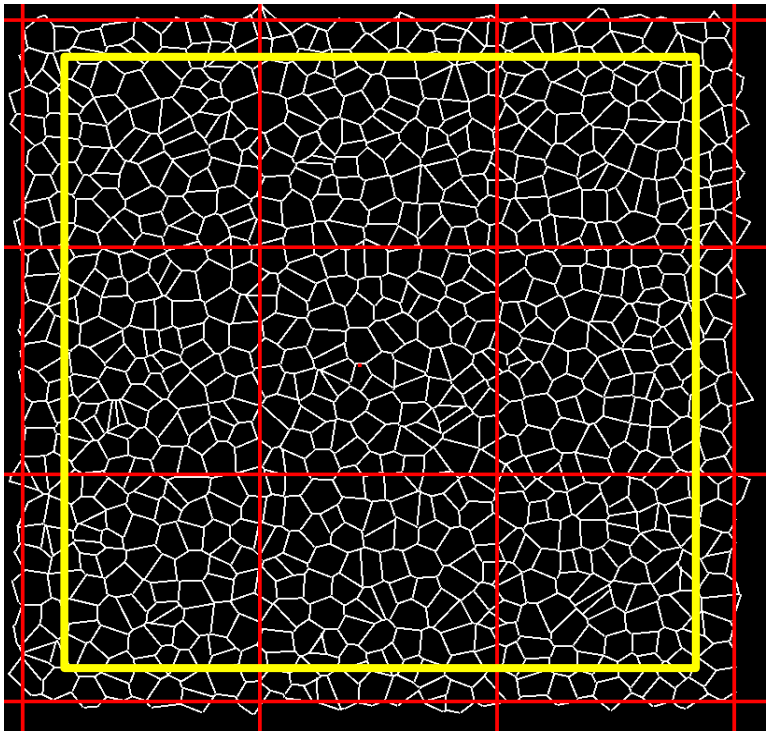

Figure 14 - A single Voronoi Diagram extending a 3x3 grid area

This now offers a potential solution to the neighbouring polygon edge connection issue. The yellow rectangle is shown again as an rough illustration of which polygons will likely be unaffected if only the points creating the edge polygons outside of this yellow rectangle change.

We can exploit this property, if for every single cell, we generate the corresponding 3x3 grid and then only extract the single (centre) cell. This will in turn mean every single cell which is generated and extracted will have edge polygons which directly align with their neighbours. This works, as the yellow square (which indicates polygons of certain shape), will always include the edge connections to its neighbours. This adds a 9 (3x3) times constant factor to Fortunes Algorithm as each cell now must incorporate all the points of the neighbouring cells, but this does not change the overall running time. A possibly optimization is worth mentioning in that we can only process the Voronoi Diagrams up until the yellow lines, which would reduce the overall time complexity.

An illustration of this is seen in Figure 15, where as you move up, you can see there is a change in the edge polygons outside the yellow square, while the centre cell will always have an aligned connection to its neighbours.

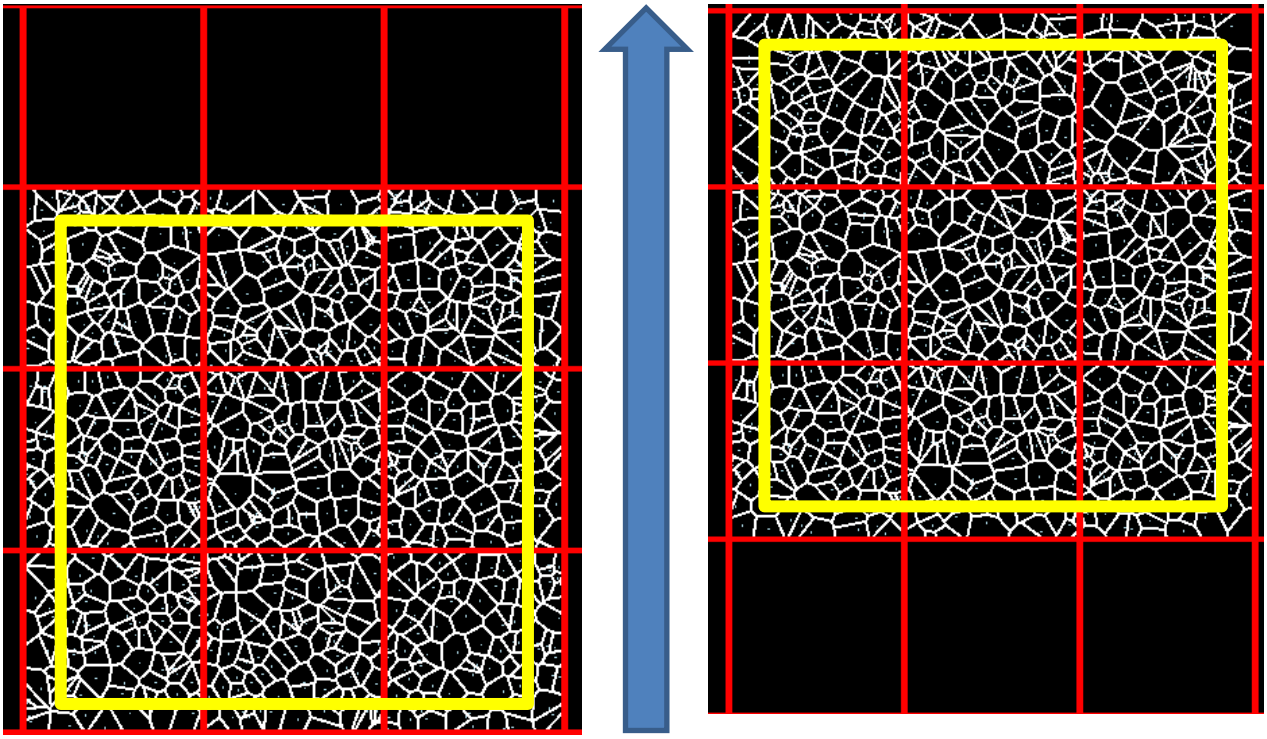

Figure 15 - Showing how the 3x3 grid allows for a seamless connection when moving up amongst the centre neighbouring cells

3.3.2 – Point Generation

The success of this extension to Fortunes Algorithm rides on the assumption that the points within a cell are evenly distributed and that there are many of them, to ensure that only the edge polygons will change in the event of different points being used to generate a Voronoi Diagram. When we randomly generate points within a region, we can break it further down (Figure 16) and then generate points within each of these regions. This ensures that there will always be points within the edges of a square, and will keep the inner-cell polygons unaffected. In practice, however, for larger cells, more refinement of cells is needed and will be discussed further in the implementation. Although Poisson Disc Sampling was investigated in the background section, this method of point generation ensures there are edge points within a given cell.

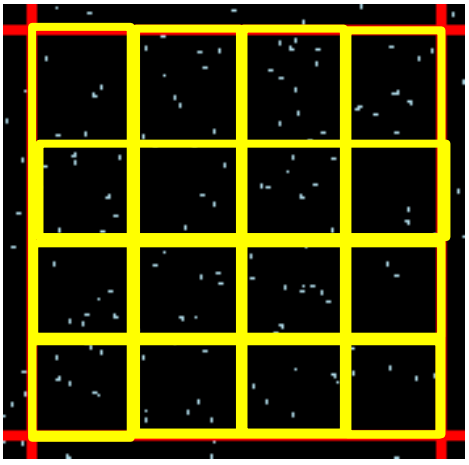

Figure 16 - A cell broken up in to finer cells for adequate point distribution

3.4 – Height Creation

In the background there were many options discussed, such as Perlin Noise, Fractals, Primitive Shapes and Erosion. These can build upon any basic shape, and therefore the choice of using Voronoi Diagrams to use irregular polygons is irrelevant for the choice in height creation.

This height-map will be assigned at a per-polygon level.

However, with the aim of the project being to build large scale worlds in a deterministic and efficient way and to not require the information of any previous generated terrain to generate a new area, this adds serious constraints regarding using the methods discussed. Rain erosion is used specifically for a finite area, and although deterministic, is computational expensive for realistic results, and relies on the assumption that there is some place for material, such as direct, to flow to. As this project is generating areas in real-time, environmental features are not necessarily known for neighbouring cells and therefore the built-up material could become piled up at area borders.

This becomes a challenge as it is very important to hiding any forms of artificial construction, such as erosion build up at cell lines.

Perlin Noise is a proven technique which is deterministic and very fast in computation. Using Perlin Noise alone, however, will generate a lack of large scale and realistic structures due to the degree of variation and influence it gives (Figure 5). This makes it impractical to guarantee the generation of features that will be used for game-usage, such as mountain ranges, peaks, and large flat areas.

Using Primitive Shapes gives the best control over the specific features that should be included in the generated terrain. The choice of shapes can allow for mountain ranges, flat areas, and a diverse changing environment both small and large in scale while involving minimal implementation detail.

3.4.1 – Shape Generation

The difficulty with correctly implementing these primitive shapes to generate terrain, is that with previous implementation attempts in real-world projects they were used with a conjoint algorithm, such as erosion (which is impractical in our infinite setting) to create realistic terrains [5] [12]. However, with some consideration, it is possible to theoretically produce terrain which is realistic enough for game-usage with primitive shape influence alone.

The primitive shapes themselves will be generated at a per-cell basis, meaning, any given cell will have a set of shapes associated exclusively with one cell. This means that the shapes will be generated in a deterministic way, based on the numerical seed associated with that cell.

The shapes that will be used are:

- Circles
- Connected line segments

The idea here was to keep it simple, and focus on creating a “line generator” which would produce connected line segments to mimic real world structures. These include mountain ranges, rivers and more. The circles are to represent natural variations, bumps, and depressions in the terrain.

These shapes will be placed over the terrain, and will influence the polygons in the Voronoi Diagrams which are close to the shape, with either a negative or positive height influence. These shapes are generated by placing them at a random X, Y coordinate within their respective cell.

For circles, there will be a random radius and height (either negative for depressions, or positive for hills) assigned. Connected line segments will be more complicated, and will have random features associated depending on whether it is a mountain range, river or hill.

Mountain ranges will be longer, strong positive influences that will often branch out, requiring that the connected line segments allow for branching. Rivers will be long, and smooth line connections with minimal angle changes. Their height influence will be a gradual depression. Hills will be like mountains, except a weaker influence. The actual line generation will still be randomly created. However, depending on the line-type, their random values will be more likely to assume the characteristics specified.

Each cell will get a randomly allocated a number of shapes, with circles being the most common.

3.4.1.1 – Large Scale Structures

Shapes generated inside a given cell have the potential to overflow the cells borders in to neighbouring cells. It is crucial to detect this and ensure the neighbouring cells are influenced correctly, else there will be obvious influence formations at all the borders which will not look natural. Shapes are required to be very large, to create proper mountain ranges, hills or rivers. It is not practical to generate both very large and very small shapes from a single grid cell.

The way around this is to create a separate grid-overlay structure, specifically for large shapes. This will be a “super grid” in that each cell will be exactly 10 times as large (therefore one super grid cell contains 10x10 normal grid cells). These are specifically to generate large shapes to ensure the generation of large scale structures, while keeping the smaller shapes to be generated in the ordinary cells. This gives both a large and small-scale shape generation to give a realistic feel.

3.5 – Biome Generation

Biomes are a method to add interest and variety to a terrain, which is especially important for very large-scale worlds. The biomes are to be based off Whittakers biome model [13], however a simpler approach will be taken. The existing model features a very realistic and distinct model of the biomes which exist in the real world, but due to both time constraints and in the essence of game-usage, a higher-level biome selection will be chosen.

These high level biomes will incorporate a variety of the biomes listed in Whittakers; and will be:

- Grass – encapsulates Boreal forest, Temperate seasonal forest and Woodland/shrubland.
- Desert – encapsulates Temperature grassland / cold desert and Subtropical desert
- Snow – encapsulates Tundra
- Wet Grass – encapsulates Temperate rainforest and Tropical rainforest.

These simplifications make it easier for the individual to identify and remember unique features about each terrain while still adding a variety.

These biomes will still be accurately based off the precipitation vs temperature chart in Whittakers model, and be applied to each polygon in the Voronoi Diagram.

3.5.1 – Temperature and Precipitation

For biomes to exist accurately, the terrain must have numerical temperature and precipitation value at any given region. This project will incorporate these two features using Perlin Noise, as it is deterministic, efficient and allows the smooth transition of values over large areas which model temperature and rain-fall in the real-world well, as seen in Figure 17. Furthermore, to ensure temperature is more accurate, it will be influenced by the height of the region that it is in, dropping by 7.9 degrees Celsius per 1000 metres as in the real world [14]. The ratio of how temperature is determined is 30% by the Perlin Noise random value, and 70% on the regions elevation. The precipitation will be purely the value given by the Perlin Noise.

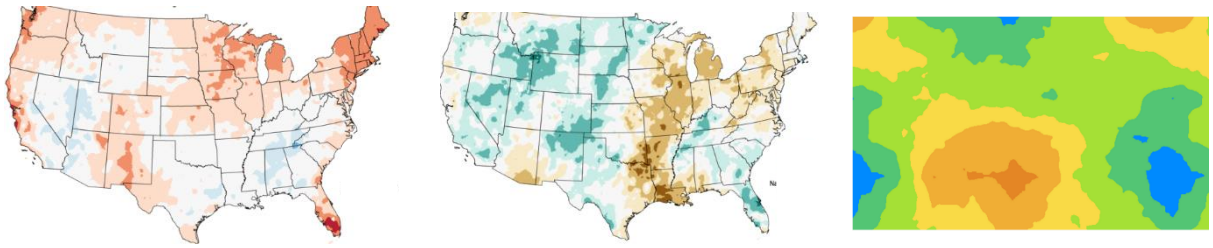

Figure 17 – Comparison of 2017 Mean Temperature of USA (left) [26], 2017 Total Precipitation of USA (middle) [26] and Perlin Noise Example (right) [1]

The temperature and precipitation will use different Perlin Noise functions, and the input to this noise function will be the X and Y coordination's of the region, to ensure a deterministic and repeatable nature.

3.6 – Terrain Representation

Until the height-map was produced, the project needed a way to render to both measure progress and debug. A 2D representation was built to show the Voronoi Diagrams and Shape generator and to ensure that they were working correctly. After this, effort was put in to generating a 3D engine utilizing MonoGames powerful framework.

Having both the 2D and 3D representation will allow the best of both worlds for debugging and evaluation.

3.7 – Evaluation

Evaluation technique is the most important part of this project it needs valid methods to ensure the algorithm was a success. As in the aim, we want a world generation which is fully deterministic (and therefore require no physical storage), efficient, and effective in producing terrain which has realistic features seen in the real world (such as mountains, rivers) and areas of terrain good for game usage (such as flat areas, and mountains also).

The first evaluation technique can simply be tracking the performance of the terrain generation in real time, and ensuring that the in-game computation of generating neighbouring terrain is done in under 0.1ms as this is the time the users will notice delay, and in the event of a teleportation, the overall time taken to generate a completely new area will take less than 1.0 as this will alter the users flow of thought [15].

The next evaluation technique will be ensuring used by comparing the generated terrain to real-world terrain and seeing if real world features exist in the terrain, such as mountain ranges. And in similar fashion, ensuring the terrain holds interesting features that will be used directly for games.

Various other methods such as terrain size in real-world metrics could be used and to ensure that the game meets the aim by requiring no storage on the computer.

4 – Implementation

Sections (3-3.7) laid out the high-level view of the significant components used in conjunction to meet the aims of the project. The implementation varies greatly from the high-level view seen in the approach, as that does not account for practical issues that are encountered in programming. This implementation will specifically be done using C# 6 and MonoGame using Visual Studio 2017 to develop in.

Figure 18 shows a segment of terrain generated by the result of this project, with the initial terrain seed of *123124*. As with the initial aims of this project, all the terrain here can be reproduced exactly just with this initial seed, as every component used was ensured to be fully deterministic. This yields the benefit of only needing to share a single seed amongst all players who wish to engage on a single world, as well as no storage being needed on the computer as every area can be regenerated as needed. For scale, there is a single cell missing in the top right corner which is of size 2000x2000 pixels.

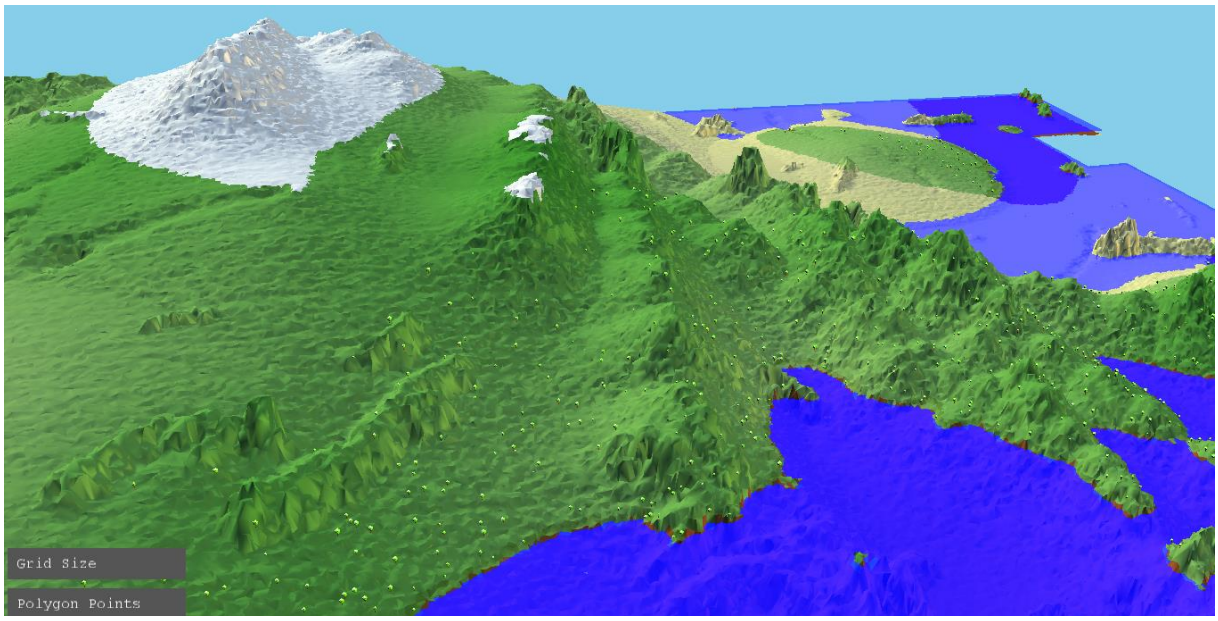

Figure 18 - Segment of the 3D visualization of the terrain generated from this project.

The seed which backs the terrain will be a 32-bit C# signed *integer* value, and will be randomly assigned at the start of each instance unless specified. All of the random values generated within this project utilize C#'s native *random* class, which is a pseudo-random generator [19]. This means that given the same numerical seed it will produce the same sequence of values every time.

Figure 18 is the result of all the algorithms specified in the approach running in conjunction in real time. The implementation is as follows:

4.1 – Grid Overlay

This was the first part to implement as it was the back-bone to ensure a deterministic large-scale terrain generation. Throughout the implementation, extreme care was taken in the coding to ensure as much flexibility as possible in all the algorithmic features. This meant that by the end of the project, it was very easy to tweak major settings. In the grid overlay, there are a set of grid sizes to choose from. These sizes are **1000** (small)**, 2000** (medium)**,** and **4000** (large) pixels, both identical in height and width to create squares.

Each grid cell has a unique X and Y coordinate representing the top left corner, which can be used to generate a unique numerical value (with the Cantor Function [17]), which then is further combined with the terrains unique seed to generate a single numerical value distinct to that cell. This final numerical value works as the seed value for the pseudo-random generators for the algorithms in that cell to ensure a deterministic nature. Due to the constraints of C#, we are limited to using the *long* type (which is 64 bits) [18] as this is the largest numerical type that can be practically be used as the seed backing the pseudo-random generators.

As we are limited by this size of 64 bits for the final seed used in the cell, and the way in which the Cantor function works, this means that the unique value produced for each X and Y must be 32 bits. The terrains seed is also 32 bits (the sum is 64 bits). This limits the X and Y types to be a C# *short* which has a maximum negative and positive value of –32,768 to 32,767 respectively. Using the largest grid cell size (large- 4000 pixels) this roughly allows the total world size to be 7800km wide (Appendix B).

The project itself then will be limited by these X and Y values, and will be unable to generate grids further out than the limits of the *short*, in effect giving a cap to the “infinite” nature. This is a consideration, as it is not practically infinite, rather, just very large.

4.2 – Voronoi Diagrams

Voronoi Diagrams were implemented using Fortunes Algorithm as the means of construction. This algorithm proved to be complex to implement. It involved the use of 2D points, using MonoGames inbuilt *Vector2* [20] class (a struct holding an X and Y *float* variable). The generation within Fortunes Algorithm requires the use of randomly generated values, which again, utilize the *random* class with the cells unique seed.

4.2.1 – Point Placement

Fortunes algorithm takes in a set of points as it’s input, and the region to create the diagram in. The points are created based on the grid-break technique discussed in the section 3.3.2, which ensures that there is always a good distribution of points as well as sufficient edge points for the region. For flexibility, there is a “Polygon Regularity” setting, which allows **Not Ordered (1), Semi-Ordered (2)** and **Ordered (4)** with the default being Semi-Ordered. These set a multiplier for the number of internal grid areas to be created within a cell to produce random points within. A cell will be broken in to 16 multiplied by the setting, sub-cells. The more sub-cells, the more regular the point distribution will be which also effects how regular these polygons will be as they are more sparsely separated.

To ensure enough points too, the number of points is proportional (linearly) to the size (in pixels) of the grid cell. The choice in number of points are **Low (3), Medium (5)** and **High (7)** which represent the multiplier on how many points are generated, with the default being Medium. There will always be enough points to put at least one point in to every sub-cell. The more points, the more polygons there will be, which also means for a fixed cell size, the smaller each polygon will be on average.

All the points were based off the inbuilt *random* class in C#, using the cells numerical seed, allowing for repeatable and identical results. The X and Y values for each point were generated within the specified region.

4.2.2 – Voronoi Generation

As defined in the approach, the "infinite" and seamless generation of neighbouring Voronoi Diagrams in real-time require the generation of a single Voronoi Diagram over the respective 3x3 grid structure surrounding the single cell we wish to produce. This means we must pre-generate the points for these areas. Once the diagram is produced, we cull off all the polygons which are not contained within this single cell and then assign the corresponding diagram to the cell.

The result is that a specific cell will be associated with a Voronoi Diagram, which holds a set of polygons. However, the extension which allows for a seamless connection creates the issue of polygons extending over cell borders, as seen in Figure 19.

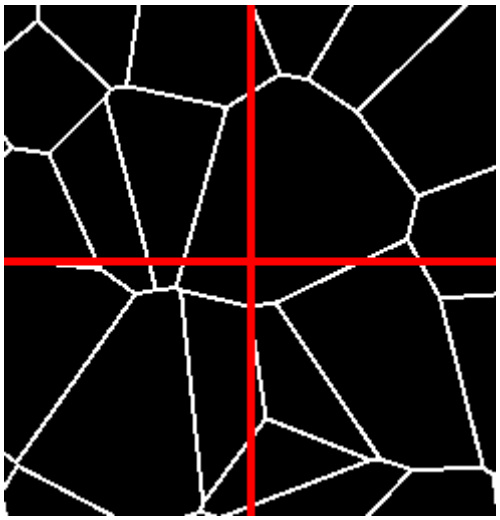

Figure 19 - The corner of four cells sharing a single polygon

In this figure, we can see that at the border of four cells, each of the cells Voronoi Diagrams share a polygon. However, in the construction, four separate polygons will be created. This created a technical complication, as not only do we not want to re-create an already created polygon, but also want to create a link between the separate Voronoi Diagrams so we can map pathing between cells over polygons.

To overcome this, whenever an edge polygon is made (one which overlaps a cell border) it will check all other cells which it overlaps in to check if the polygon has been made already. If it has, it will not re-add it, and furthermore will generate a link between the cells.

This method now allows for connections to be made between polygons which belong to different Voronoi Diagrams, allowing pathing to exist, which will be useful for water flow, and evaluation methods. These connections can be seen by the green lines which show the polygons connections to their neighbouring polygons (Figure 20).

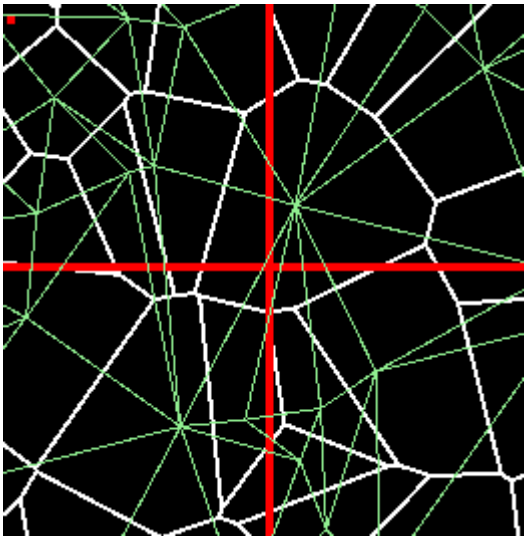

Figure 20 - Connection over polygons over cell borders

Figure 21 is an example to show how only the cells relevant to the user's path are generated. It shows how if a user teleports, as in a game mechanic, it does not require the generation of all the path between, yet will only generate the relevant information to the user. Whenever the user leaves a path, it will destroy the objects to conserve system resources as it can re-generate the identical terrain again later.

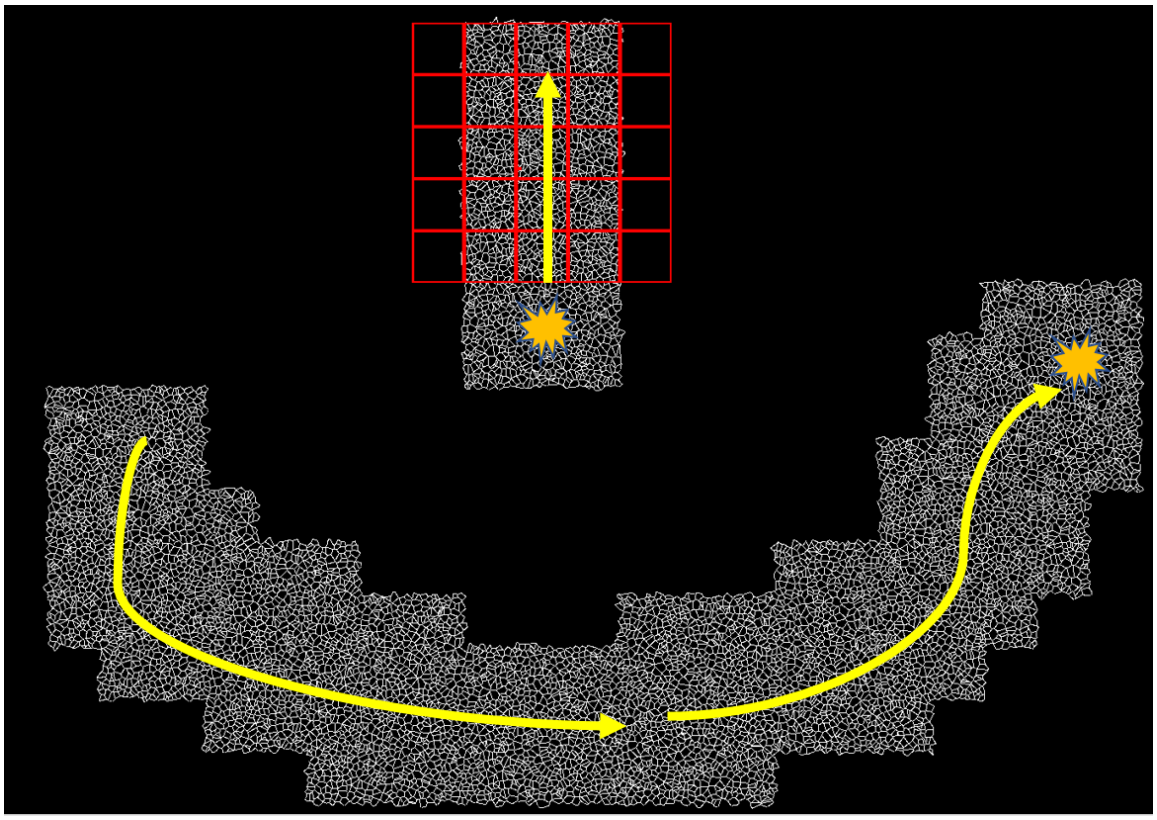

Figure 21 – Shows the seamless generation of connecting Voronoi Diagrams which follow the cells surrounding a player's path (yellow lines). The orange explosions represents teleport endpoints within the game.

The Voronoi Diagrams therefore now are based off three-flexible settings: Grid Size; Point Amount; and Polygon Regularity; which are all changeable at run-time (Figure 22).

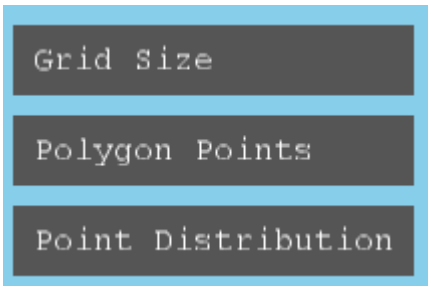

Figure 22 - In game settings which effect Voronoi Diagrams

These settings drastically change how the polygons are shaped; for example, the difference between the Point Distribution setting "Unordered" and "Very Ordered" is considerable – **Low** point amount setting was used for easier comparison (Figure 23).

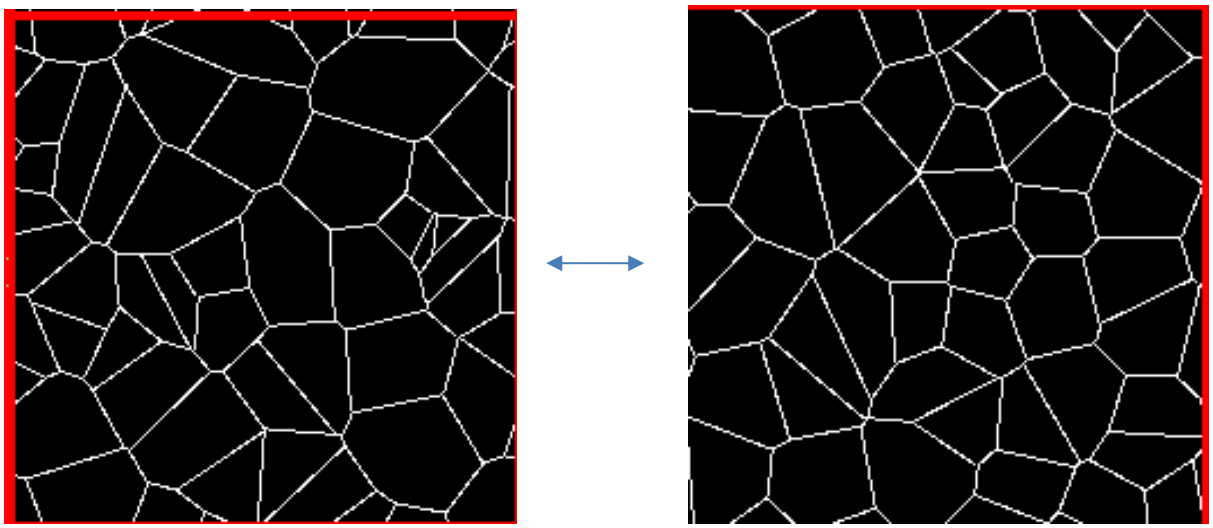

Figure 23 - Comparison of the "Polygon Regularity" setting Unordered (Left) vs Very Ordered (right)

4.3 – Shape Generation

The ability to generate extending Voronoi Diagrams in real-time, creates a structural basis for terrain to be built upon. All the polygons created have their corresponding X and Y regions defined; meaning that if each polygon has a height corresponding to it, it will be able to be rendered in 3-dimensional space.

Each polygon will have a single numerical height, which will be normalized between 0 and 1. The height values used to normalize these values will be zero as minimum height, and maximum being the grid-size (therefore, larger grid sizes allow taller structures). The reason behind this maximum height cap is to avoid have structures taller than they are wide, as a means of controlling very sharp gradients and mitigate areas which aren't explorable in a games context.

As discussed in the approach, each cell which needs to be generated will generate a list of shapes to be associated with it. As mentioned, there is also the concept of "super-shape grids" where each super-grid cell encapsulates 10 standard grid cells. Therefore, when a standard cell needs to generate its terrain, so must the super-shape cell containing it (however only the shapes are generated in this super-shape cell). This grid is implemented in the exact same way containing an X and Y coordinate, and being generated when it is visible to the user.

Shapes sizes are directly proportional (linearly) to the size of the cells dimensions they are generated from (and therefore if it is generated from within a super-grid cell, it will be on average 10 times larger). This helps to scale the shapes with changing grid sizes.

The shapes generated can be large, and go over many cell boundaries. However, consideration was made to ensure that it will never go over more than four boundaries. Because of this, whenever a cells terrain needs to be generated, so must the all the cells in the 4x4 boundary (as well as the super-shape grid) around it to check for any shapes which overlap into the specific cell.

The shapes which currently implemented are circles, and connected line segments.

4.3.1 – Circles

Circles were the first primitive shape to be created, and were simple to imp lement. Each circle holds an X,Y coordinate to represent its location, along with a radius and height influence. Circles are intended to be used for general "noise" in the terrain, and generate hills at a larger scale. Figure 24 shows an example of four standard cells with their generated circles (on the left) while on the right a circle generated from a super cell can be seen, with standard cells for scale.

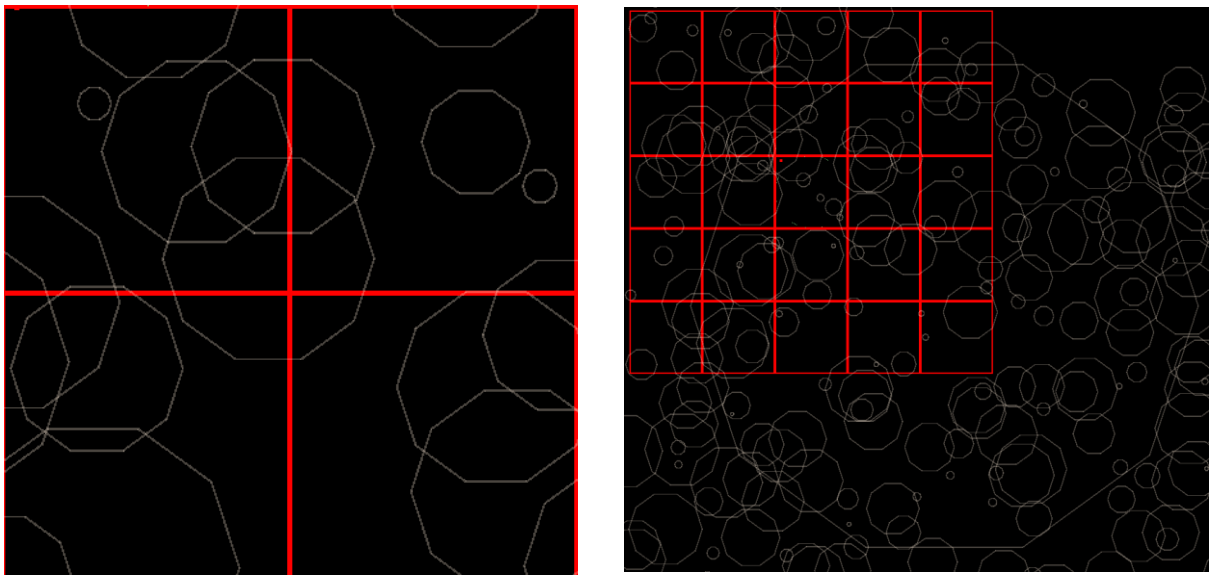

Figure 24 - Visualization of generated circle primitive shapes over standard four grids cells (on the left) – Generation of a circle on the right can be seen from a super shape cell

The circles themselves will only influence the heights of the polygons which are inside their radius. This influence will either be negative, or positive depending on whether the circle is a depression or hill (this is a 50% random chance); as this allows for either lakes or hills features to be formed. The influence is completely linear in its effect, with the peak of influence being at zero distance between the centre of the circle and the polygon.

The height is always directly proportional to the radius, (with a constant factor being random between 0.5 and 1.5). The radius is a random value from one fifth of the grid size, to half the grid size.

4.3.2 – Line Generator

The ability to generate custom connected line segments was difficult task to implement well as this "shape" is intended to generate rivers and mountain ranges (ie, most of major terrain features). This component was implemented by starting with a single line segment, and recursively adding additional line segments to the end of each last added segment until one of the following criteria was met: the overall line segment height influence is too small; there are more than 20 segments; or the total line length is too long.

To easily create very unique and a wide variety of different connected line segments, there were a variety of settings applied to each created line segment.
These are: **Angle Change** – the amount each line segment can change in angle from the last; **Branch Amount** – the probability that a secondary line segment will branch off when creating a new segment; **LineLength** – the overall length of the line segment; and **Height Influence** – how strongly

the influence will be. Each of these settings are flexible, and tweaking them in certain ways allow for a many different types of connected line segments to be created.

To properly judge height influence of a connected-line component, every line-segment within the shape has a radius associated with each end of the line, this is illustrated by the pentagons which have a radius showing their area of influence (Figure 25). A line may have many areas of influence overlapping due to large radiuses and small line-segments length, which can lead to weird height influences on the polygons. To overcome this, the top five influences are taken from each connected line segment.

The starting line segment has a random height and radius, and each connecting line-segment will have a new height and radius based off the last. This allows for a natural growth in influence (or decline) which is a feature seen in hills, rivers and mountains.

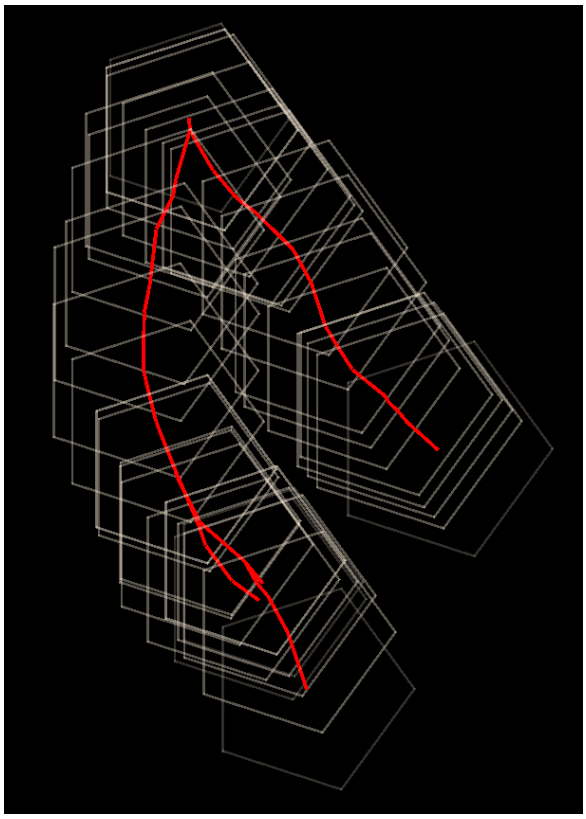

Figure 25 - A connected line segment featuring branching. This is of type hill.

Due to time constraints, the only major line types created were mountains, hills and rivers. Rivers were the only line type which offered a negative height-influence; with the other two being positive.

4.3.2.1 – Line Differences

Rivers, hills and mountains at their core are similar in their generation with a different statistic likely-hood for certain settings to be chosen (defined above). Rivers tended to be smoother, long, with minimal branching therefore the AngleChange setting was statistically more likely to be lower, with a longer LineLength, with a shallow height influence etc.

On the contrary, hills and mountain both provide a positive influence. Mountain ranges have a higher likelihood to branch, and provide a sharper height influence. Hills will be smoother, and branch less. Both of these types have equal chance of all AngleChange settings.

These values are determined based off C#s *random* class again, with the seed being the cells unique value

This causes an issue however, as inherently the super-shape grid cell at 0,0 will have the same unique numerical seed as the standard cell at 0,0 – which will in turn mean they generate the same shapes, except the super-shape cell will just exhibit a larger scale. To overcome this, as the user may be able to recognise repeated patterns, the Y value in the super-shape grid is inverted when used as the seed (as X and Y are *shorts*, this means for the practical use of the seed in the super-shape grid, the Y value will be *short.MAX – Y).*

This completely fixes the issue, as each super-shape cell contains 10 standard cells, the point where these values will be equal again will never exist as the *short* bounds for standard cells will become an issue first.

4.3.2.2 – Height Influences

As we are not using erosion to make these raw shape influences look more realistic, this puts much more meaning in that way that the shapes inherently influence the surrounding area as it must produce decent results from this alone.

In the real-world, mountains generate peaks, while hills are smooth gradual changes and so are rivers. This is not always the case, but the idea of creating realistic features most of the time is fine as this is used in a game context, and the occasional not-so-realistic feature just bypasses the user.

4.3.3 – 3D Rendering

Now with the ability to generate a normalized and real height for each polygon created within visible and neighbouring cells, it is possible to render this terrain in a 3D fashion. This will provide a massive benefit, as 2D rendering is not intuitive to individuals who are not familiar with what they are looking at, thus allowing a larger audience to understand what they are seeing.

Rendering at its core within MonoGame [16] relies on the passing of vertices (points in 3D space) in sets of three (triangles) to the graphics processing unit to render on the screen. To implement this, whenever a polygon is created, it will be triangulated. Then for each vertex within that triangle, the height associated with the polygon will be mapped to that vertex. The idea behind this is that it will create a smooth render surface, as each vertex belongs to many polygons with changing heights, so averaging the heights will create a proper render surface.

This 3D render engine was generated from scratch relying on the framework that MonoGame has provided. Using the influences that were provided from the primitive shapes, and rendering the terrain all one colour, it resulted in Figure 26.

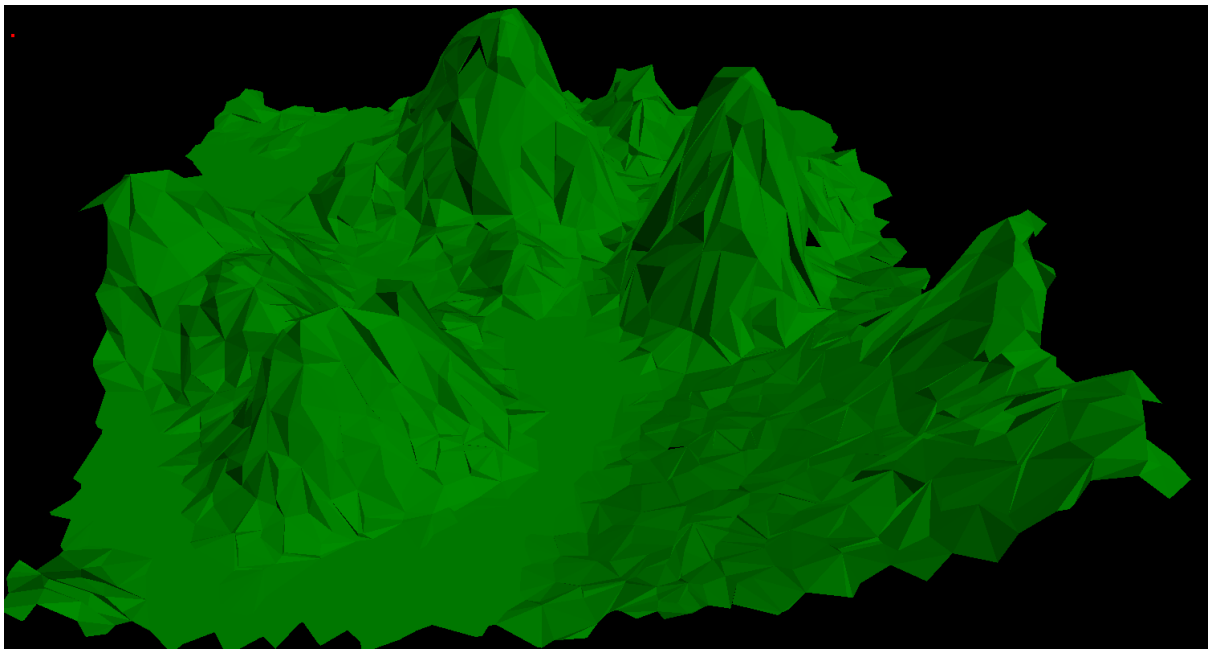

Figure 26 - Initial 3D rendering of a 3x3 grid structure, each cell being 2000x2000 pixels

Figure 26 shows a 3x3 grid structure being rendered in 3D. Each grid cell here uses the Medium grid size (which is 2000x2000 pixels) and therefore the region we are seeing is 6000x6000 pixels total.

4.3.3.1 – Smooth Lighting

Figure 26 has very sharp edges between the triangles which create a very unrealistic terrain surface, as in the real world, terrain is a lot smoother. We wish to remove this sharpness, and can do so with a trick known as Weighted Vertex Normals [21]. The way the lighting engine works is by calculating the normal of the triangles surface, which is a line perpendicular to the triangles tangent plane [21], which then is used by MonoGame to determine how the lighting should effective the curved surface.

A triangle is composed of three vertices, however each vertex is shared by many others each with different normal values. A smooth surface can be made for each vertex, setting its normal value to

the average of all the triangles it is connected to. Furthermore, it is important to not only average them, but weight the normal value based off the area of the triangle. This is important as a tiny triangle should not have the same lighting effect as a larger one [21].

Saving the normal average was done in the same way that the height of any vertex was done. The vertex was hashed, and then the resulting normal average was saved along with each vertex.

The effect of implementing this turned is seen in Figure 27.

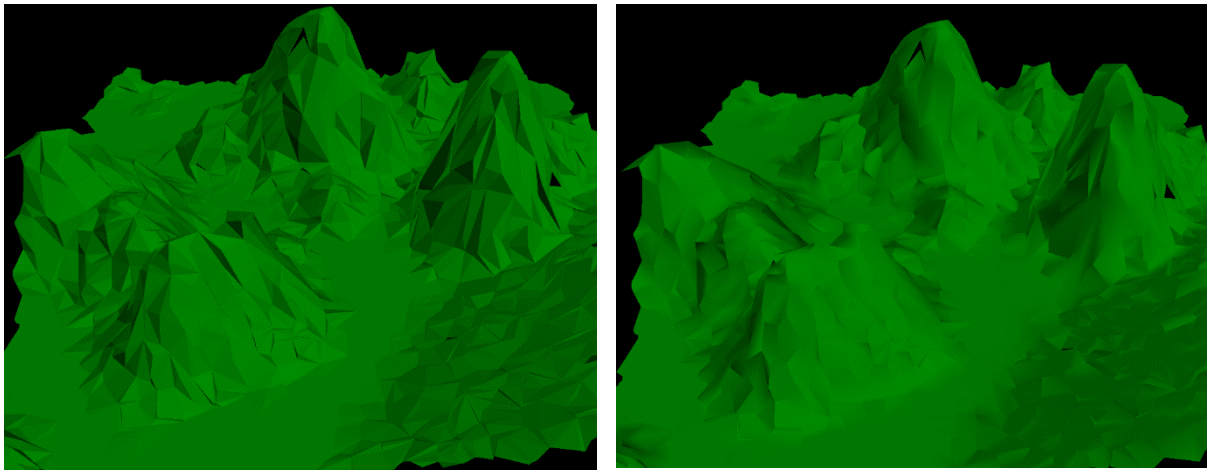

Figure 27 - Comparison of terrain with (on the right) Weighted Vertex Normals against without (on the left)

4.3.3.2 – Float Imprecision

Both the normal averages and the height averages for each vertex were stored with their 2D Vector2 (X and Y) being hashed. The vertices which are a part of edge polygons of each Voronoi Diagram are generated potentially over many different Voronoi Diagrams which connect. This in effect causes very slight differences in their float values and therefore causes a different hash to occur.

In effect, if a float difference is large enough that it hashes to a different value, this will cause the height and lighting normal averages to not be hashed to the right vertex value. This is an inherent problem within MonoGame, as their use of float values within their base variables (such as Vector2 and Vector3) is built in. Although this has its purpose for graphical renders, it is not meant for hashing. Float variables are not precise by nature once they hit significantly large numbers. They are backed by 32 bits, and are only capable of accurately representing seven significant figures [22].

This gives it a problem in accurately representing points, and as the grid cells can be very large (1000-4000 pixels), these vertices can be massive and cause float values to get very imprecise very quickly. Assuming that floats need to be accurate to at least 3 decimal points for accurate hashes, and that the individual has chosen the largest grid size of 4000, the moment 3 grids are created this already has created 8 significant figures (5 figures to store the location of 10,000 + and 3 for the decimal points) which create a high likely hood for inaccurate hashes (Figure 28), which show up as tears in the terrain.

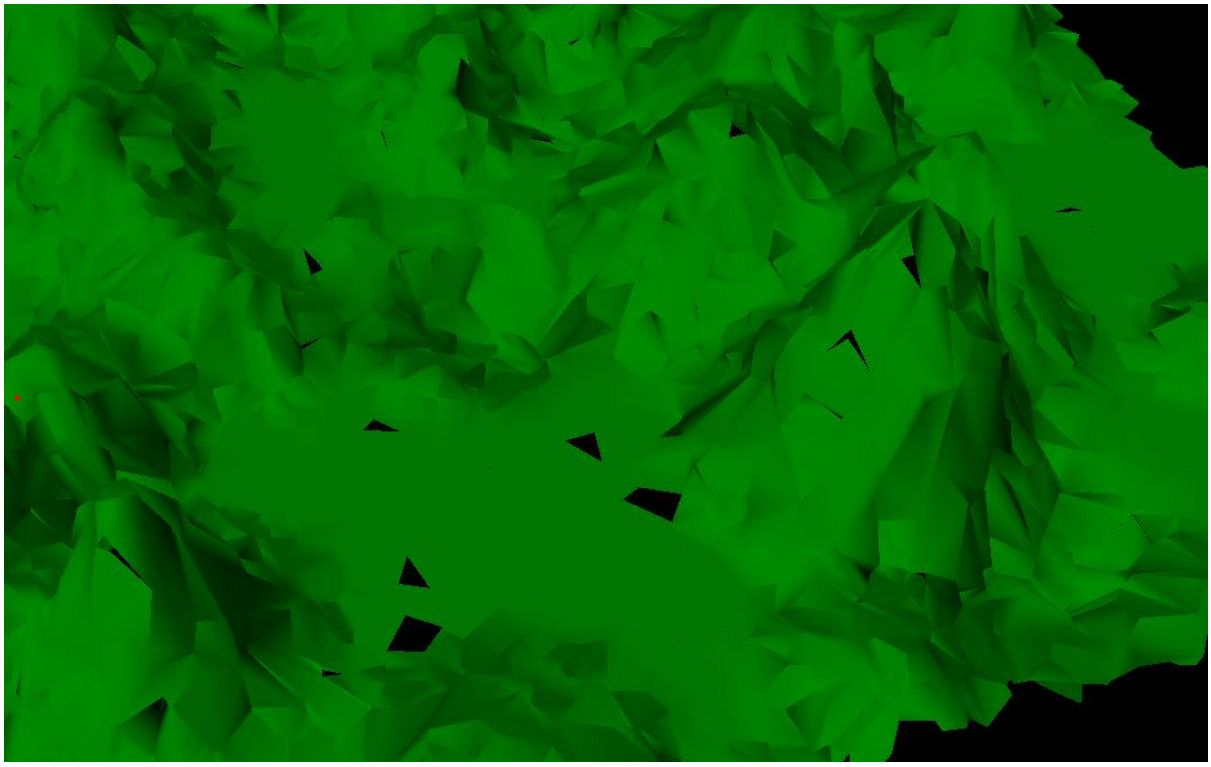

Figure 28 - Incorrect hashing causes tearing in terrain

The limits of using *shorts* for the grid structure, and the maximum size of a grid (4000 pixels) means that at most the terrain will exist at 131068000 (32,767 * 4000). This value is 9 significant figures, including 3 decimal points, meaning that we require 12. This can easily be fixed by switching the floats to doubles, as the *double* type is a 64 bit backed floating number which can accurately store 15-16 significant figures [22]. Because of this, the implementation shifted to using new classes created to mimic the Vector2 and Vector3 behaviour of the native classes, yet having *double* backing instead which completely fixed the tearing as they can store accurately within the scope of the terrain generation.

All modern processors (x86) will handle both *float* and *double* variables in the same way meaning there will be negligible performance differences by now using *double* backed vector classes [23]. The only concern here is that the doubles will use twice the storage. This in the context of our project is not a concern as there will never be many cells stored at any given time.

4.4 – Biomes

The implementation of biomes wasn't a particularly hard task to achieve. As described in the section 3.5, the idea was to utilize Perlin Noise (which is repeatable and identical) to create two unique value maps over the terrain; one used for temperature and one for precipitation. The values were based at a polygon level, and each polygon was assigned a temperature and precipitation value between 0.0 and 1.0 (stored as a *double* type) from a Perlin Noise function which had the X and Y input being the centre coordinate of the polygon.

Precipitation was purely based on the value assigned by the Perlin Noise function, however, to add a more realistic nature to the temperature, this was also influenced by the height of the polygon. As the polygons height has been determined prior, we can now determine its final value.

Temperature is based 30% on the Perlin Noise function and 70% on the height. The height influence on the temperature is determined as follows:

The following values are normalized between 0.0 and 1.0
Average Height: 0.3 (this is an arbitrary value that can be changed)

The difference between the polygons normalized height and the **Average Height** will then directly be the influence for the temperature. For example, if a polygon has a height of *0.5*, its temperature decrease will be by *0.2 (but in reality this is 0.2 * 0.7 as height is a 70% influence)*. This is because the different between the height and the average height (0.5 – 0.3) is 0.2.

The values of the temperature and precipitation then are used to determine the biome based on Whittakers Model.

4.4.1 – Biome Colours and Water

To distinguish between the biomes within the 3D rendering and to add life to the terrain, colours were added. Each biome will be assigned two distinct colours, to be associated with its "above sea level" colour, and its "below sea level". Sea level is a concept which is introduced to create water features which are important for game usage and a realistic feel.

The following values are normalized between 0.0 and 1.0
Sea Level: 0.15 (this is an arbitrary value that can be changed)

Anytime a polygon's normalized height is below sea level, its colour will be set to its "below sea level" colour, else its "above sea level". There is a difference between these as the biome would change its appearance (such as grass to dirt) in the event of heavy water exposure. The colours are intended to mimic the real-world appearance, and are listed in Table 1.

Biome	Above Sea Level Colour (RGB)	Below Sea Level Colour (RGB)
Grass	**Grass -** Green (R:0, G: 128, B: 0)	**Dirt -** Brown (R:160, G: 82, B: 45)
Desert	**Sand -** Yellow (R:238, G: 232, B: 170)	**Saturated Sand -** Gold (R:255, G: 255, B: 0)
Snow	**Snow -** White (R:255, G: 255, B: 255)	**Ice -** Light Blue (R:173, G: 216, B: 230)
Wet Grass	**Saturated Grass -** Dark Green (R:0, G: 100, B: 0)	**Dirt -** Brown (R:0, G: 50, B: 0)

Table 1 - Colours for each biome. Colours are listed with their respective R G B values

Once this was done, it was also important to render water at the constant sea level height. Water within this project was defined as having the colour *blue* (R:0, G:0, B: 200) with an alpha (transparency) of 128 (50%). It is possible to simply render water over each cell of colour *blue* at a 50% transparency to mimic water. We can render this water layer as two triangles which span the entire cell, and will be rendered exactly at the height of *0.15* (normalized sea level), and then repeat

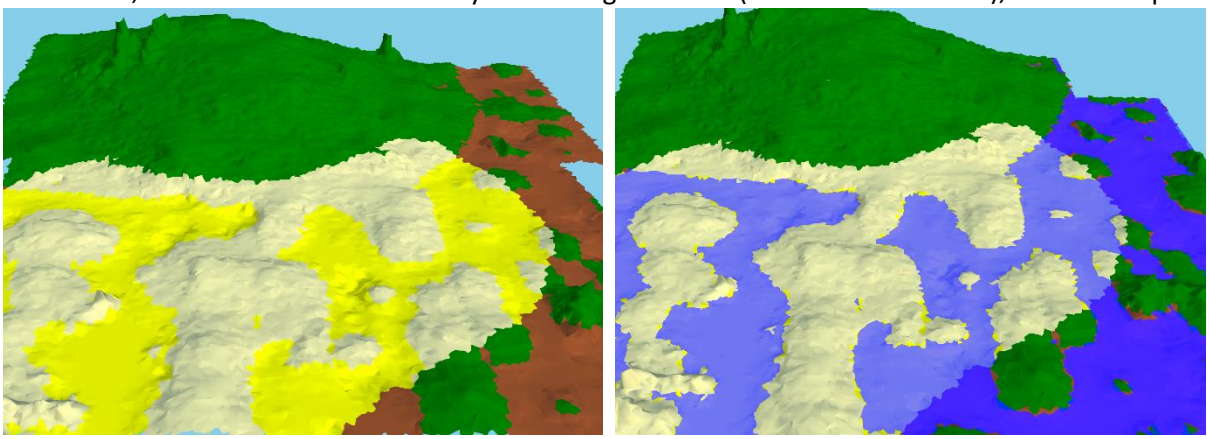

Figure 29 – Comparison of the same terrain being rendering with (on the right) water and without (on the left)

this for all cells that are being rendered. This, along with the biome colours had a result seen in Figure 29.

Figure 29 shows the "Above Sea Level" and "Below Sea Level" colours for both the grass and desert biomes. The usage of a transparent water layer makes it look more realistic, however, the change in colours between biomes are very sharp. As the biomes are determined by a mapping between temperature and precipitation, there therefore exists boundaries where it is a close decision between two biomes. In this event, the project implemented a smoothing colour technique which merges the colour between various biomes depending on how "close" they are to a biomes criteria.

This smoothing is shown in Figure 30.

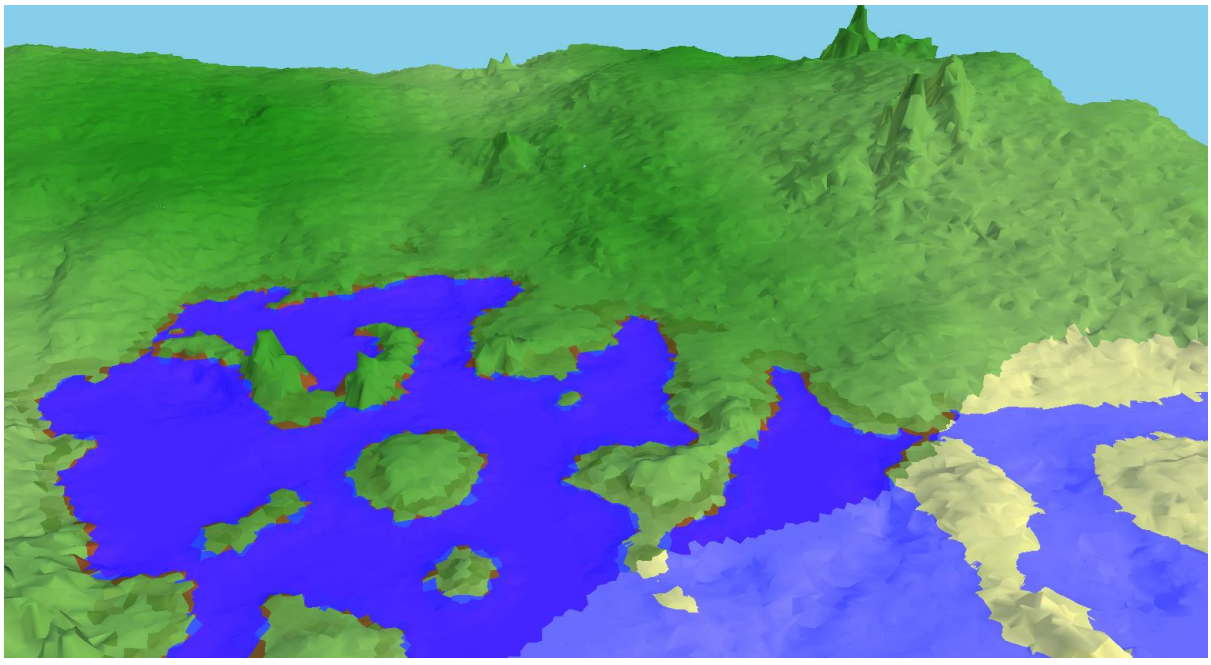

Figure 30 - Smoothing of biome colours to match the criteria (temperature vs precipitation)

Figure 30 shows a clear change in colour over the grass in this example. The smoothing works by making the darker patches of grass correspond to the higher level of precipitation and the lower levels of temperature. This natural variation adds a more realistic feel as opposed to block colours.

4.4.2 – Trees

Trees were added to help add perspective and scale for the individual viewing the terrain. Until trees were added there were no objects to relate to the grand scale of the terrain.

The trees used in the render were oak trees by model. The placement of trees, as for real oak trees, will only be in areas of optimal temperature and precipitation, therefore virtually only appearing on grass biomes. The range itself is between 0.3-0.5 temperature and 0.4-0.7 precipitation. The chance of a tree spawning on a given polygon is random chance, based on how close the polygons temperature and precipitation values are to the optimal value (centre of the bounds). Figure 31 shows the spawning on trees, with a cell removed (of 2000x2000 pixels) for to show scale.

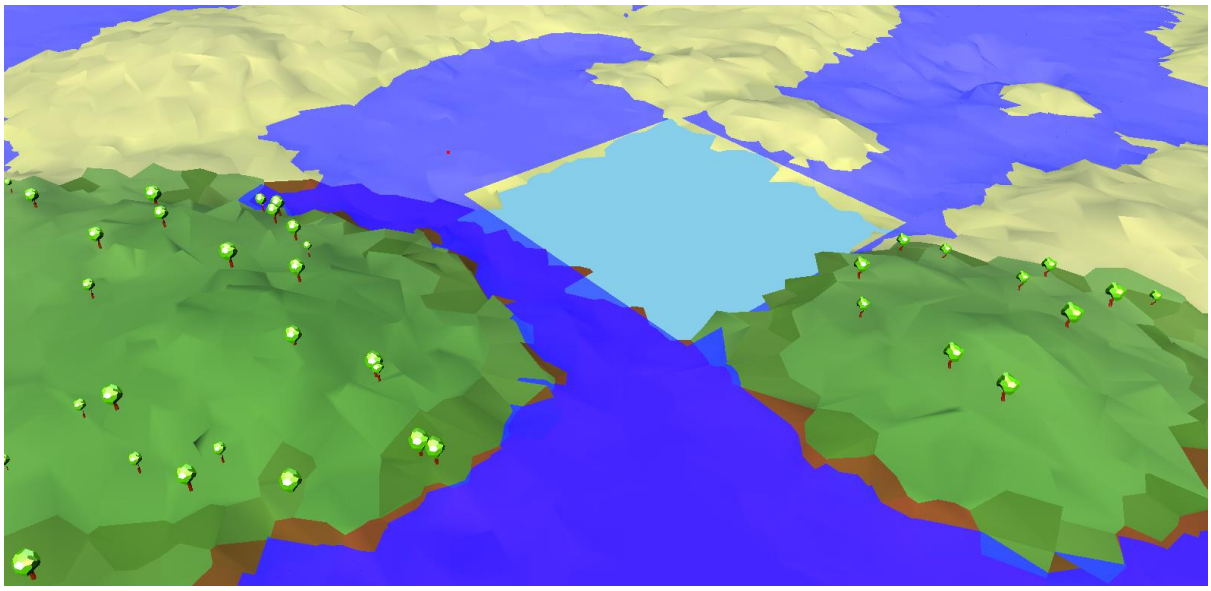

Figure 31 - Showing trees with a chunk removed for scale. The cell is 2000x2000 pixels

4.5 – Algorithm Steps

As discussed, only terrain which is visible to the user will be generated. At present the implementation only ever generates the 3x3 grid around the cell the user is currently in (however this is easy to change, and may be a setting with future work that a user can opt to increase with a more powerful computer).

MonoGame provides a base framework that has an update loop which is called 60 times per second. To reduce putting all the algorithmic stress in to one update tick and potentially causing delay for the user, the algorithm gets split over many ticks. The algorithm in implementation works as follows:

One Tick

- Check to see if the area within the 3x3 of the user has been generated
    - If there are gaps, generate the Voronoi Diagram for it
    - Triangulate all the polygons
    - Keep track of generated areas

Second Tick

- Check to see if the area within the 3x3 if only a Voronoi Diagram has been generated
    - Generate the shapes for that cell
    - Generate the height map
    - Generate the biomes (this completes the terrain generation)

Third Tick

- Check to see generated areas no longer within the visible view
    - Dispose of the area no longer needed to reduce system memory usage

As there are three distinct ticks, that allows at most 20 grid areas to be produced per second which allows the user to move very quickly. This also spreads the computational load over the ticks, to ensure no one tick takes too long and causes delay. As the third tick is purely house-keeping, the first two are of interest to evaluate.

5 – Evaluation

The nature of this project has many aspects which are difficult to truly evaluate due to the aspect of subjective opinions, however, by referring to the aim it is still possible to make as many concrete conclusions as possible.

5.1 – Evaluation Metrics

5.1.1 – Storage Requirements and World Size

The terrains generation is based entirely off a single numerical *seed* value determined at the start of the project (or may be specified). The application itself requires zero storage on the local computer it is being ran on due to the ability that it can regenerate a given area of terrain in real-time for the same result, assuming the same seed. This means that there is no bottleneck for exploring the entire theoretical world. This aim was met successfully.

5.1.2 – Replay Value

Given the same numerical *seed* the terrain generated will always be identical at all areas within the games world, being since all the algorithms selected to perform the generation are entirely deterministic, leading to a repeatable and identical nature. This numerical *seed* is an integer value, being a 32 bit variable within C#. This means that there are at least 4294967295 different seeds available, which correspond to the number of different terrains that can be generated [18].

5.1.3 - Multiplayer

This allows a massive amount of replay value, due to the number of different terrains that may be generated, but the fact it is all based off a single numerical seed means that only this single seed value must be shared amongst different users to effectively use this generator in a multiplayer environment. This removes any bottleneck produced by slow internet connections in terms of no longer having any need to share generated terrain over a network.

5.1.3 – Efficiency

One of the core aims was to produce new terrain at run-time in minimal time so the user will not notice any significant delay. Benchmarking in Appendix A reveals that with max settings chosen (Grid size of *large* and Polygon point amount of *high*) that a modern computer will approximately a generate single cell of terrain within two separate ticks of 11ms and 13ms respectively (Appendix A).

Although 100ms is regarded as the tolerance before users notice any real delay [15], it is worth noting that as MonoGame has 60 update ticks per second. This equates to each update tick lasting approximately 16ms. Therefore, MonoGame itself can handle the worst case of a 13ms computation in a frame, and as such, MonoGame will not have any unexpected delay and nor should the user notice.

Users will experience an interruption of thought process if they experience a delay of more than 1 second. This may be a problem in the event of a teleportation which would mean the whole 3x3 grid would need to be generated (as there are no pre-generated cells). As discussed, the algorithm can produce 20 terrain cells per second, which by far includes the 3x3 (9 cells total). These 9 cells would be generated in at most 27 update ticks (as three ticks per cell) meaning approximately 0.5 seconds within MonoGame.

The efficiency of this terrain generation avoids the 100 ms threshold of being able to create neighbouring terrain cells (for continued exploration) and the 1 second threshold for teleportation, meeting the projects aims in this respect.

5.1.4 – Effectiveness of Voronoi

The aim of using Voronoi was to remove any obvious regularity from the terrains core structure. When looking closely, Figure 32 shows core shapes that the terrain is made from. There are obvious sharp corners which exist, except they do not show any regular pattern. It would be possible to reduce this by increasing the number of polygons per cell at the expense of computation. However, the aim was to eliminate the appearance of regularity which Voronoi Diagrams achieve.

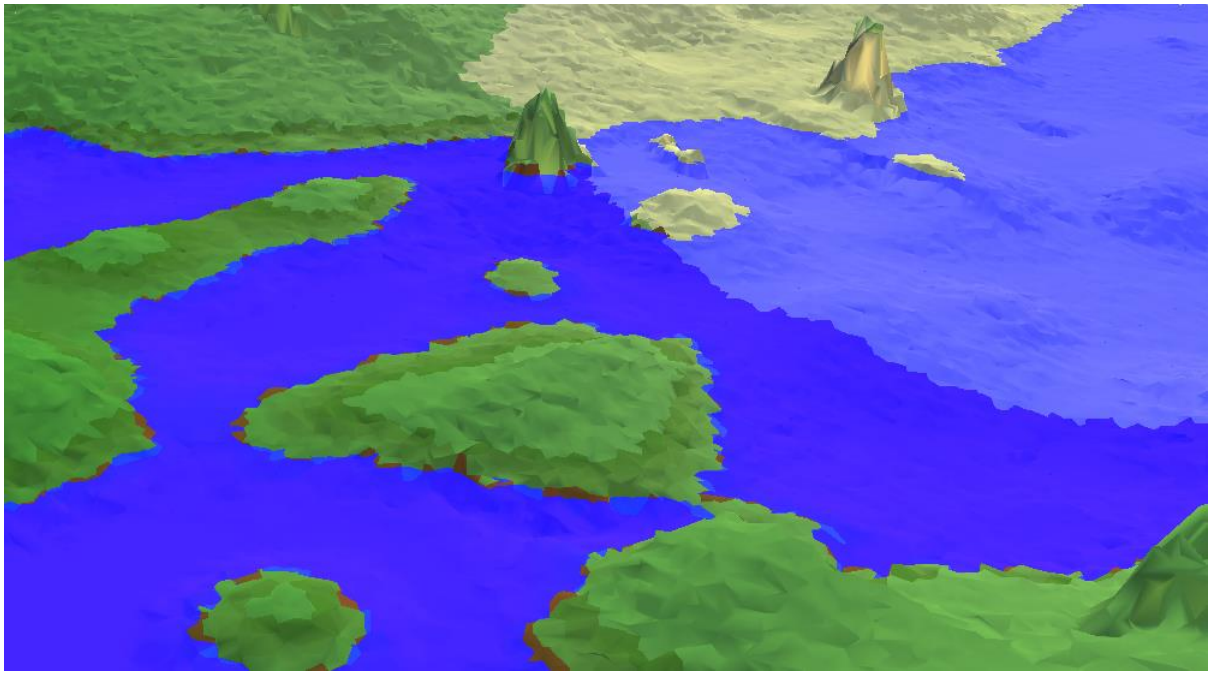

Figure 32 - A segment of terrain to highlight the visible irregular polygon structure

5.1.5 – General Realism

Note, the following comparison photos were received from https://www.pexels.com/ *which are licensed under the* ***Creative Commons Zero (CC0) license*** *and therefore require zero attribution. (*https://www.pexels.com/photo-license/*)*

We can draw comparisons from segments within the generated terrain against real-world features. Figure 33 shows a variety of comparisons held with each respective comparison being left to right on a single row. The pictures of the generated terrain were found within a 50-cell radius from the start, and were chosen over five separate random seed generations. We can therefore draw direct comparisons and see the similarities between the terrain generated and real-world features and say that they do exist within the game environment.

Furthermore, large scale shapes have successfully been implemented which was one of the core aims of the project. The mountain range comparison within Figure 33 (2nd to bottom row) span over 1 kilometre long.

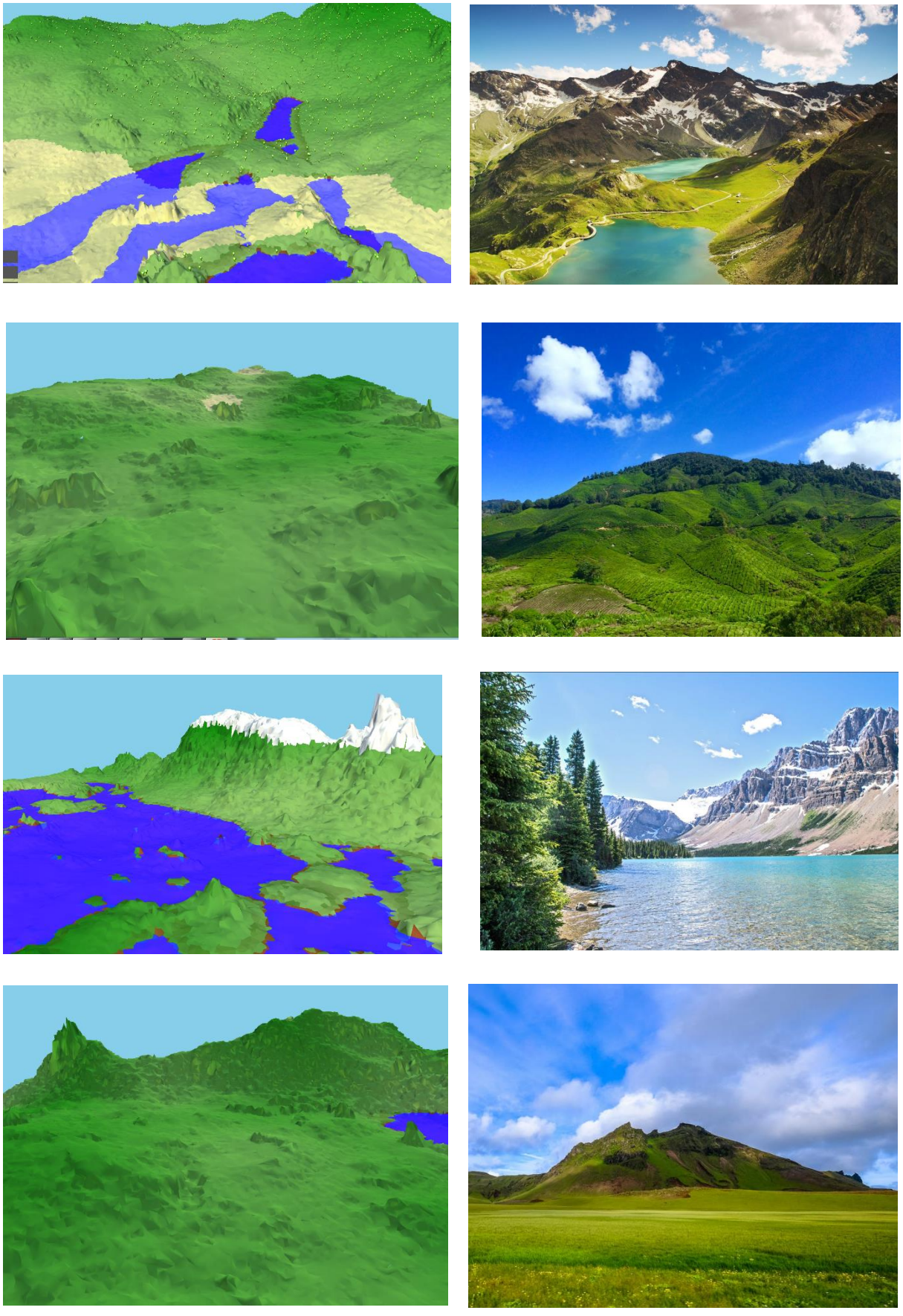

Figure 33 - Comparisons of in-game terrain segments against real-world features (left to right). Photos sourced via www.pixels.com

5.1.6 – Game Usage

Game usage is a subjective topic, and depends almost entirely on the genre of game. This project was general purposed and intended to have flat areas for building upon or having battles on, as well as interesting features to explore. Figure 34 were all terrain segments found within a 100 by 100 cell block area with the seed of *1234.* The fact that these features were easy to find and abundant show good proof of concept of the success of this terrain generation for game interest and usage.

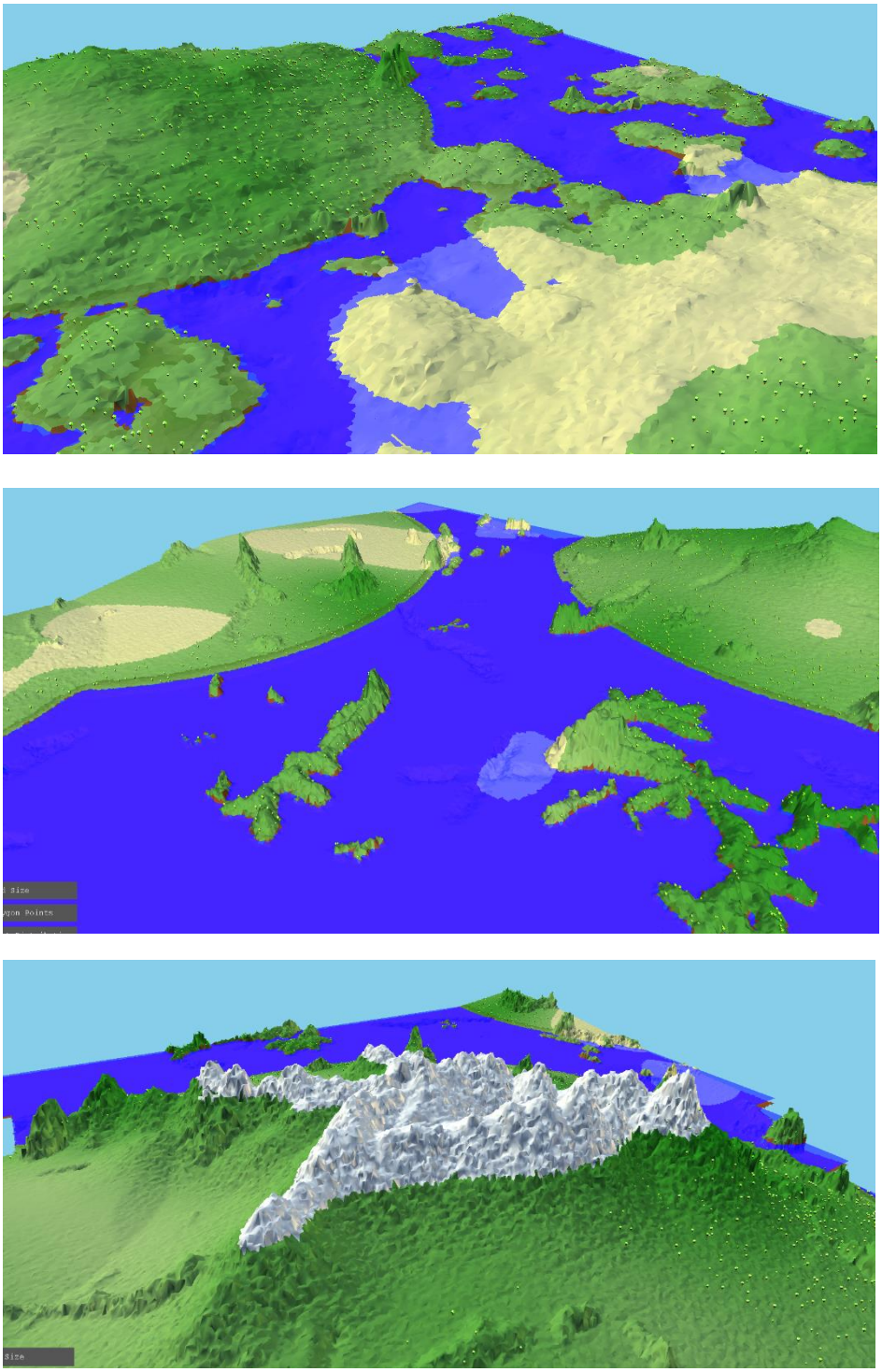

Figure 34 - Three segments of terrain found within the seed of 1234, showing flat areas, large water features and explorable mountain ranges

5.2 – Potential Issues

The existing algorithms in place are prone to unexpected behaviour in the resulting generated terrain, and this is a consequence of decisions made throughout this project.

5.2.1 – Bounded Generation

Although the world size has a maximum size of 7864km by 7864km, it is still bounded by the numeric limits within C#. The decision to use a grid-cell overlay and to set their coordinates as *shorts* limited the cells to go at most –32,768 or 32,767 in the X and Y direction from the centre. Although it is very unlikely the user will ever reach these bounds, it is still an issue if they do as the terrain will not generate any further. (This may happen in the event of teleportation or very fast travel in one direction). This issue can always be mitigated using larger variables (such as replacing *shorts* with *integers* and making the respective changes to integrate this), however this just delays the issue. This will always be an issue within a software environment and therefore the best we can do is ensure the user is unlikely to be effected.

5.2.2 – Double Imprecision

The decision to replace MonoGames Vector2 and Vector3 usage with double-backed identical classes (opposed to float) just delays the issue of imprecision. The use of the *double* type does allow for a much larger amount of precision (15-16 significant figures, vs the *float* 7 [22]). Although the *double* type as of now works due to the size limit of the bounded generation (from the size of the *short* limiting the size), if the *short* ever got replaced to a numerical value of larger bounds, the *double* might again not be appropriate again.

5.2.3 – Shape Super Grid

Although this does solve the issue of having large shapes which extend much further past the users view point, this still delays the issue. Having this structure limits the size of shapes being produced still based on the size of the cell which may cause annoyance in the user considering the world size may be up to 7864km wide, and that the max shape size is a fraction of that. This could be mitigated by adding an even larger scale of super grids.

5.3 – Future Work

5.3.1 – Storage

Although currently the terrain requires no long-term storage, some game scenarios may need terrain related storage. The base terrain requires no storage to generate and use, however, once the user starts to build on the terrain or alter it in any way the changes must be stored.

5.3.2 – More Primitive Shapes

Currently there are only two forms of primitive shapes, circles and connected line segments. While these shapes are intended to mimic real-world features such as mountain ridges, hills and rivers, these two shapes limit the features which can be be created. The addition of more shapes would help add variety as well as enable more features to be seen.

5.3.3 – More Biomes

The biomes which exist within the terrain are based off Whittakers Biome model, however has a very simplified version. The introduction of all the different biomes within this model would increase the variety and interest of the world.

5.3.4 – Immersive Environment

Outside of the terrain, there exists only one type of tree. Addition of more tree types, shrubbery and explorable places such as ruins, or randomly generated villages would increase its game usage potential.

6 – Conclusion

The aims of this project were to create an infinite (only bounded by numeric precision) terrain generator, which had no obvious regular base structure, required no storage on the user's computer, and was applicable for game usage. The approach used sufficiently covered these aims by providing an efficient means to generate terrain around the user at run-time while requiring zero storage on their computer.

This project proved the potential in using Fortunes Algorithm at a context larger than the initial finite area that it was previously bounded to. The seamless addition in any direction by exploiting the line-sweep proved that Voronoi Diagram have applicable uses within a game context to hide any regularity on the surface at a very-large scale. The use of primitive shapes along with the shape super-grid implementation allowed for the generation of realistic height-maps. Furthermore, larger-scale structures were also able to be generated successfully to create features such as mountain ranges, rivers and lakes.

The terrain itself contained flat areas to build and battle on, as well as interesting areas to explore using mountain ranges and ever-changing environments with biomes. The entire terrain world can be created identically if the initial seed remains the same, meaning that in a multiplayer context only this one seed value needs to be shared removing previous network strain.

The end-result showed a realistic and differing terrain when compared to real-world features, while still maintaining direct applicable game usages with flat areas to build-upon and vast mountain ranges to explore.

If this project was to continue being developed, the addition of more primitive shapes and biomes should be implemented to allow for a greater range of features to occur in the terrain as well as allowing for further variety in the terrain.

Overall, this project was a success in meeting the initial aims set out.

Bibliography

[1] H. Tulleken, “How to Use Perlin Noise in Your Games,” 25 5 2009. [Online]. Available: http://devmag.org.za/2009/04/25/perlin-noise/.

[2] Intel iQ, “Fractal Terrain Generator: Simple Method For Stunning Landscapes,” 23 4 2014. [Online]. Available: https://iq.intel.com/simple-code-generates-beautifully-realistic-terrain-with-fractals/. [Accessed 1 10 2017].

[3] J. Fingas, “Here's how 'Minecraft' creates its gigantic worlds,” 03 04 2015. [Online]. Available: https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/. [Accessed 20 4 2017].

[4] NoMansSky, “Procedural Generation,” 24 8 2017. [Online]. Available: https://nomanssky.gamepedia.com/Procedural_generation. [Accessed 6 10 2017].

[5] Red Blob Games, “Polygonal Map Generation for Games,” 04 09 2010. [Online]. Available: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/. [Accessed 2 4 2017].

[6] S. Fortune, “A Sweepline Algorithm for Voronoi Diagrams,” *Algorithmica,* pp. 153-174, 1987.

[7] D. M. Haahr, “Introduction to Randomness,” 2015. [Online]. Available: https://www.random.org/randomness/. [Accessed 10 4 2017].

[8] T. Mounton and E. Bechet, “Lloyd relaxation,” Universiy de Liege, Liege.

[9] R. Bridson, “Fast Poisson Disk Sampling in Arbitrary Dimensions,” University of British Columbia, Vancouver, 2007.

[10] J. Olsen, “Realtime Procedural Terrain Generation,” University of Southern Denmark, 2004.

[11] E-DOG, “Water erosion on heightmap terrain,” 8 October 2011. [Online]. Available: http://ranmantaru.com/blog/2011/10/08/water-erosion-on-heightmap-terrain/.

[12] M. O'Leary, “Generating fantasy maps,” 2015. [Online]. Available: http://mewo2.com/notes/terrain/. [Accessed 10 4 2017].

[13] R. H. Whittaker, “Classification of Natural Communities,” *Botanical Review,* vol. 28, no. 1, pp. 1-239, 1962.

[14] “Does Elevation Affect Temperature?,” 8 8 2016. [Online]. Available: http://www.onthesnow.com/news/a/15157/does-elevation-affect-temperature-. [Accessed 5 8 2017].

[15] J. Nielsen, “Response Times: The 3 Important Limits,” 1 January 1993. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/.

[16] MONOGAME, “MonoGame,” 2017. [Online]. Available: http://www.monogame.net/. [Accessed 2 3 2017].

[17] O. Dovgoshey, O. Martio and M. V. V.Ryazanov, “The Cantor function,” University of Helsinki, 2005.

[18] J. Mayo, “Lesson 2: Operators, Types, and Variables,” 2016. [Online]. Available: http://csharp-station.com/Tutorial/CSharp/Lesson02. [Accessed 5 10 2017].

[19] Microsoft, “Random Class,” 2017. [Online]. Available: https://msdn.microsoft.com/en-us/library/system.random(v=vs.110).aspx. [Accessed 9 10 2017].

[20] Microsoft, “Vector2 Structure,” [Online]. Available: https://msdn.microsoft.com/en-us/library/microsoft.xna.framework.vector2.aspx. [Accessed 9 10 2017].

[21] M. Buijs, “Weighted Vertex Normals,” 23 12 2007. [Online]. Available: http://www.bytehazard.com/articles/vertnorm.html. [Accessed 13 6 2017].

[22] Net-informations.com, “Decimal vs Double vs Float,” [Online]. Available: http://net-informations.com/q/faq/float.html. [Accessed 10 9 2017].

[23] N. Limare, “Integer and Floating-Point Arithmetic Speed vs Precision,” 29 1 2015. [Online]. Available: http://nicolas.limare.net/pro/notes/2014/12/12_arit_speed/. [Accessed 8 10 2017].

[24] “Oak tree facts,” [Online]. Available: http://www.softschools.com/facts/plants/oak_tree_facts/505/. [Accessed 20 8 2017].

[25] L. Viitanen, “Physically Based Terrain Generation,” Metropolia Ammattikorkeakoulu, 2012.

[26] NOAA, “National Temperature and Precipitation Maps,” 2017. [Online]. Available: https://www.ncdc.noaa.gov/temp-and-precip/us-maps/. [Accessed 24 4 2017].

Appendix A

Benchmarks for the computation of terrain; these refer to the distinct two ticks of generation specified in **4.5**.

The specifications of the computer which preformed these benchmarks are:

CPU: Intel Core i7-5700HQ – 8 core 2.70GHz
RAM: 12GB DDR3
GPU: Intel HD Graphics 5600

The benchmarking process will test two core attributes: Grid Size (small, medium and large) and Polygon Points (low, medium, high). Both of these metrics directly create a larger amount of polygons within the Voronoi Diagram which is the basis for computational expense. In the interest of benchmarking, the lowest setting on both Grid Size and Polygon Points will be ignored as we are curious for the higher cases of performance.

The following table outlines the average first and second tick times, taken over the generation of 100 cells.

Grid Size	**Polygon Points**	**Average First Tick (ms)**	**Average Second Tick (ms)**
Medium	Medium	**5ms**	**7ms**
Large	Medium	**7ms**	**8ms**
Medium	High	**9ms**	**11ms**
Large	High	**11ms**	**13ms**
Max Times		11ms	13ms

Appendix B

The scale of the world is based off the trees present in the game for scale.

Oak tree bases approximately grow on average to 4 feet in width (approximately 1.2 metres) [24]. The model base itself spans 20 pixels, and therefore this scale can be used to deduce an approximate size of the world using the metric system.

Therefore, using 1.2 metres per 20 pixels we get the results in

	Metres	**World Dimensions** (Multiplied by the *shorts* limit)
Small Grid Size (1000x1000 pixels)	60m x 60m	1966km x 1966km
Medium Grid Size (2000x2000 pixels)	120m x 120m	3932km x 3932km
Large Grid Size (4000x4000 pixels)	240m x 240m	7864km x 7864km