PRIFYSGOL
BANGOR
UNIVERSITY

School of Computer Science and Electronic Engineering
College of Environmental Sciences and Engineering

Fantasy Map Generation on Voronoi Diagrams

Alex Mainstone

Submitted in partial satisfaction of the requirements for the
Degree of Bachelor of Science
in Computer Science

Supervisor Dr. Llyr Ap Cenydd

September 2019

Acknowledgements

Hello, world! Alex MainstoneAcknowledgements Help and advice from Dr Llyr Ap Cenydd and everyone else kind enough to read this paper has been greatly appreciated and, as a result, I can say certainly that it has vastly improved in quality thanks to your help. And of course, thank you to the authors, directors, and developers for creating the best genre of media.

Statement of Originality

The work presented in this thesis/dissertation is entirely from the studies of the individual student, except where otherwise stated. Where derivations are presented and the origin of the work is either wholly or in part from other sources, then full reference is given to the original author. This work has not been presented previously for any degree, nor is it at present under consideration by any other degree awarding body.

Student:

Alex Mainstone

Statement of Availability

I hereby acknowledge the availability of any part of this thesis/dissertation for viewing, photocopying or incorporation into future studies, providing that full reference is given to the origins of any information contained herein. I further give permission for a copy of this work to be deposited with the Bangor University Institutional Digital Repository, and/or in any other repository authorised for use by Bangor University and where necessary have gained the required permissions for the use of third party material. I acknowledge that Bangor University may make the title and a summary of this thesis/dissertation freely available.

Student:

Alex Mainstone

Abstract

In this thesis, we will explain our implementation of a map generator onto a Voronoi diagram to create natural-looking environments. The implementation of a Voronoi Canvas that can be drawn to will be the foundation of this thesis. It will consist of Voronoi cells that can be specified as land or water, with the outline of neighboring cells being drawn as the coastline for our island Pangaea. We will also be exploring the generation of cities across this map as well as the use of Markov chains to generate unique names for each of these cities. Finally, adding detail to the map, roads from city to city, large forests, and mountains where appropriate.

Contents

List of Figures

Chapter 1

Introduction

Procedural Generation allows game developers to produce an arbitrary amount of content and replayability in their games without needing to manually produce a vast quantity of assets. This thesis will detail the implementation of a fantasy map generator. The goal of this thesis is to replicate the art style found in traditional Tolkien-style maps for their implementation in games and other media.

The video game industry is one of the largest entertainment industries valued annually at $134 billion. The rapid growth and expanse of the industry has led to it becoming a very profitable source of revenue for big businesses. With higher budget game development comes higher consumer expectations, leaving independent developers at an extreme competitive disadvantage in the market.

Complex features such as open worlds have become an expectation from the industry as it brings extremely good consumer value for money for both the business and the player; the player can expect a large amount of content, and the business does not need to invest resources in developing it. Freedom of choice, as well as some random events, can significantly increase the replay value of a game, as the player can expect a different result each time. However, the expectation of procedurally-generated content can result in more disappointing content, as many games seem to add open-world aspects as an afterthought or simply to increase the total length of their game, requiring the player to navigate from objective to objective across significant distances, with many games failing to provide an engaging open-

world experience. Many independent games implement procedural open worlds, allowing them to produce content on a similar scale to a large game studio. This comes with very beneficial side effects, such as incredible replay value as every experience will be significantly different.

As for the theme of fantasy maps, a large resurgence has occurred in fantasy since the relatively recent popularity of epic fantasies such as A Song of Ice and Fire, Brandon Sanderson's literature as well as The Witcher series, has occurred. As a result of this wider appeal of fantasy media, a wider audience is available for similar media. The cartography found in works inspired by J. R. Tolkien's Lord of The Rings has been a staple in modern fantasy, every iconic world having its unique map. My goal is to use procedural generation techniques to encroach on that style of mapping.

Procedural Generation is becoming a widely used tool among independent developers, allowing for a small team to make vast amounts of content that can compete with even the largest game developers. Dwarf Fortress is perhaps the best example of this, using procedural generation to generate a world as well as an entire history for that world, with historical figures, locations, and items. The gameplay value from a simple platformer can be greatly expanded by adding procedurally generating the content a rather simple game can be vastly expanded by implementing procedural content as well as room for variety in gameplay. The Binding of Isaac is a good example of simple procedural generation, there is a rather simple element of procedural generation but the many overlapping gameplay systems provide a very varied experience with every game. Procedural generation has become a crucial element in modern independent development. .kkrieger is a good example of one of the many applications of procedural generation, rather than use procedural generation techniques to generate content, it uses it to generate textures and 3D graphics allowing for the file size of the game to be 97kb while not compromising the visual fidelity of the game. More open games like Minecraft create massive worlds, bigger than any seen with multi-million dollar games backed by massive studios. The limitation of procedurally generated content algorithms usually involves the level of detail, computers

can be very good at generating a large amount of content, but detail and complexity is the limitation.

1.1 Aims and Objectives

1.1.1 Aims

There are a few aims that we wish to meet across the development lifetime of the project. We will state our goals here and then reflect on how closely we met with the aims we assigned ourselves at the beginning of the project. The goals are as follows:

- Generate unique city names
- Generate a Pangaea
- Generate map details such as mountains and trees
- Create a generator that completes in a reasonable time

The last step is important as the map generation will be rather surface level, meaning that it is intended to be used as just a part of a deeper gameplay system, therefore if it cannot generate the visual map quickly, the generation of a more complex system will be far slower.

1.1.2 Objectives

The objectives that we must complete to accomplish these aims are relatively simple:

- Generate an island
- Populates island with procedural cities
- Generate random city names

- Generates vast forest biomes
- Place mountain sprites in altitude appropriate locations
- Create a good foundation for a game to be created

1.2 Hypothesis

The result of these procedural maps will of course, not match the quality of a carefully handcrafted map, this, of course, is out of the scope of the project. Creating a map that is clear and readable is more important than having a detailed map, as the intention of this map is to be used in games, a ridiculous amount of detail could result in overwhelming and confusing the user. We believe that the product of our work will be a fairly rough result, that with small tweaking and tuning by someone with a clear idea for a final game, as well as some artistic talent, can create compelling maps for their game. We believe that the drawing of the Voronoi diagram will be the most difficult part of this implementation if we have particular difficulty implementing this feature we will change our map to being based on a 2-Dimensional grid.

Chapter 2

Literature Review

2.1 Background

2.1.1 Maps

Maps have been used since as early as 14,000BC, few on the planet wouldn't know what a map is if you presented them with one and thus it becomes a very compelling tool to use for world-building, it's easy to hear that events are taking place thousands of miles away and not truly understand the distance being described, visualizing this vast landscape becomes much easier with a visual representation. Therefore, by using procedural generation techniques to generate these maps, we grant a storytelling tool to our procedural content that can be used to make even more compelling worlds.

2.1.2 Game Engines

A game engine can provide a lot of powerful and easily implementable qualities that writing the project in C++ (the language I am most proficient in) would not include inherently. In most situations, writing a game engine would be far more trouble than it's worth when a straight out of the box solution included with Unity or even Godot (a powerful open-source engine). This would be true, even with the experience of implanting engines of similar complexity, it may not be as fast as using a game engine. For most situations, this would be true, but as the aim of this project is to develop a static map rather than a full, complex engine with entity systems (which tends to be a large amount of work and difficulty when writing an engine), having access to

simple, and familiar rendering pipelines will mean that developing a simple foundation for an engine will mean rapid progress not achievable in Unity or Godot due to personal familiarity with the programming language that the engine is constructed with. Essentially, not using an engine means choosing a programming language of preference rather than the language prescribed by the Game Engine. Familiarity with the C++ language means that research into useful objects and language features will be far less frequent.

2.1.3 C++

C++ is a very powerful programming language, and massively popular. The 2019 Stack Overflow developer survey states C++ as the 9th most popular programming language [1], with this noted specifically for general development. In-Game Development C++ is more prevalent as its low-level language features make it very attractive for developers wanting to create fast and efficient games.

2.1.4 SFML

SFML is a multi-platform multimedia library for C++ (and other languages) [2]. It is an all in one package for handling user input, game graphics, networking, and audio, which means that this library doesn't require others to handle different aspects of our program. It also works for Linux, Mac, and windows which means that our program can be compiled to all these platforms. It is massively popular in C++ game development and therefore developing with this library in mind makes our work accessible to all who are currently using extbfSFML. With games like Atom Zombie Smasher [3] and KeeperRL [4] being created using this library, it certainly isn't hard to find many games written in this library.

2.1.5 Voronoi Diagrams

A Voronoi diagram is simply a partitioned plane. In this case, the plane is the window that we wish to draw to. My goal is to partition the window into many random shapes so that we can draw onto it like its a grid. Essentially, we want to use a Voronoi diagram like a canvas to draw irregular and asymmetric

maps onto. This should result in natural-looking coasts, being rough and jagged like a real one would be.

Figure 2.1: Example of a Voronoi Diagram

2.1.6 Perlin Noise

Perlin noise is a type of gradient noise specifically created to look like it wasn't generated by a machine [5]. Its essentially a way of giving a pseudo aspect to computer graphics, and it's already widely used in map generation. Games like Minecraft [6] use it to great effect to create vast ranges of hills, forests, and mountains, so it's already a proven method.

Figure 2.2: Example of Perlin Noise

2.1.7 Markov Chains

A Markov chain is essentially a map of probabilities, it is given the current state of a process, and can use that state to predict the following state. How is this useful? We can apply this state machine to text, training it with a set of words, it can generate new ones that follow the same structures and more importantly, can be pronounced. This will be useful in generating truly random and new city names.

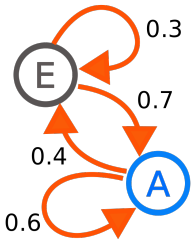


Figure 2.3: Shows structure of a Markov Chain

2.2 Markov Chain Name Generation

Finding research papers into this specific use of Markov chains was difficult, yet one of the best and most detailed explanations is done by Gustavo Zomer and his implementation of Markov chains to generate start-up names is provided in his article [7]. In this implementation, a set of words is given to the algorithm as an input, from that a list of *n-letters* is created (*n* being the number of letters stored e.g a *n* of 2 would store a list of every 2 characters in every provided word with a sub-list of every 2 letters that follow the initial 2 and a number representing the frequency at which those 2 letters occur after the initial 2).

For scientific findings for Markov's name generation, there were a lot of difficulties isolating word generators rather than sentence generators with Markov chains. Surprisingly, one of the few papers that were demonstrating unique word generation was sourced from password generation and cyber-security [8]. Using Markov chains rather than random letters brings with it a very desirable feature for some; having random, yet phonetically generated passwords, allowing it to be memorized far easier (although in research it should be noted that Markov chains were also being applied to brute-forcing these very same passwords). Examples of Markov chains each with a different number of characters being added at each step is shown in the paper allowing for the comparison of different character steps.

As far as an implantation guide for Markov chains goes, the book "Operator Theoretic Aspects of Ergodic Theory" goes into an incredible amount of detail into the functionality of Markov chains [9]. Taking advantage of the few

first simple chapters (relatively simple, still very complicated) helped gain a deep understanding of the basic principles of Markov chains, allowing us to conceptualize a class structure for Markov's name generation.

2.3 Procedural Maps

An overview of both Voronoi Diagrams and Perlin Noise is explored by Jacob Olsen [10], specifically their implementation in generating 3-Dimensional game worlds. Proposed is the use of Voronoi Diagrams to create irregular terrains that break the monotony that will usually come with computer-generated content, Perlin Noise can be applied to this Voronoi diagram to give each cell a height that can be used to render depth in the map. As the implementations proposed in this paper are specifically for 4-Dimensional worlds, simpler methods for placing details such as trees can be used.

As for 2-Dimensional implementations, after further researching Voronoi Diagrams we discovered a paper on generating Islands using Voronoi diagrams [11]. This blog post covered a similar approach for what we were aiming for so offered great insight into what I'd be doing. The idea to use Lloyd's relaxation to relax Voronoi diagrams was an idea that particularly stuck out to me. This map was really what we were trying to implement, so to differentiate ourselves from this map we decided that aiming for a more hand-drawn map with a fantasy setting inspired by Tolkien would help differentiate our work from theirs. Implementation of Markov chains for city name generation would also help our map have some more depth (not to discredit the original author, Markov chains were well out of the scope of their project).

Interestingly, a paper is written comparing procedural city generation techniques [12] explores a cellular method that applies a Voronoi diagram to Perlin noise, known as Worley's algorithm it is a method that the authors apply to generate photo-realistic textures that mimic the naturally occurring cellular structures that appear in nature. This paper notes that a large variety of natural textures can be generated by applying a Voronoi diagram to some kind

of noise. Although the purpose aimed for here is a map and not procedural texture generation, the implementation of Voronoi diagrams and noise as a way to replicate nature is clearly shown to be effective in displaying a variety of natural textures.

Possibly the most insightful of papers reviewed is an implementation of a map skeleton extraction tool, that takes images or scans of maps as an input and calculates a fitting Voronoi diagram that can then be edited [13]. This is perhaps the definitive proof that Voronoi diagrams are incredibly effective for representing maps. Figure 13 of this paper, displays the extractors' attempt at digitizing a satellite image of Guinea Bissau, where the extracted edges could very well be the foundation of its fantasy map. Capturing a fraction of the success of map portrayal that this algorithm demonstrates would be an achievement. Immediately the idea of simply reverse engineering what the authors of this paper did was considered, but careful thought about the problem made it clear that the task of generating completely new maps would not be so simple. This paper brought attention to how difficult generating complex land masses with a Voronoi would be, the concept of a rough polygonal shape representing an entire landmass conceptualized instead as essentially a tilemap of small polygonal cells having Perlin Noise applied to them to distinguish water from land.

Chapter 3

Design and Implementation

3.1 Design

After looking through similar projects in our literary review, we have determined the technologies that will be used. The concept of using a Voronoi diagram for the maps generation is both intimidating and exciting. Getting a Pangaea rendering on a Voronoi diagram brings a whole new set of challenges that will have to be both investigated and solved. Lack of experience working with Voronoi diagrams means that a lot of experimentation can be done with this. Fitting the Voronoi object to the rendering pipeline provided by extbfSFML will be an issue, as well as having map properties and graphics applied over this diagram.

Creating the foundation of a game engine was simple, although some unnecessary features were included purely to make it easy to extend the foundation being constructed. For example, an update function with a delta time (no persistent game logic), an event handling function with no planned use, although it will be very useful for debugging, allowing a keypress to run code if needed, "R" regenerating the map seems like a very important element in debugging the generation. One downside to procedurally generated content is bug fixing, as there will be bugs that will occur due to unforeseen world generation circumstances or rare conditions not met while actively hunting for bugs. It would be difficult to find minor generation bugs without an active player base to report bugs, this is a reason why many independent procedurally powered games go into early access, crowdsourcing debugging

is very efficient. Therefore a lot of development time needs to be spent simply regenerating the map and observing for bugs.

I created a scenes folder for this project, and although it isn't implemented, eventually the purpose would be to have multiple game scenes such as a menu that the player could navigate through. As that was not an objective listed, an abstract scene class that replaces the game object in this loop would allow for anything inheriting the scene object (such as the game) to be instantiated as the current scene. This is very simple to implement and would just require an abstract outline of a scene to be created and have the game scene inherit that.

Deciding on a character step for the Markov chains of this project was done by examining the output examples provided by the paper [8]. A notable difference between the cities generated and the paper showing examples is that the cities' names will be one word, as this is a true statement for most cities, thus we will not be dealing with spaces. Out of the examples provided in the paper on password generation, the two-character step seems to be the most appropriate. It has a balance between being pronounceable (the single-step examples are not easy to pronounce) and being truly unique, as higher character steps tend to have many distinguishable words.

One problem that required deep consideration was that of coastal rendering, after careful consideration the challenge it presents is more complex than originally considered. A boundary around all connecting land tiles needs to be calculated on generation, meaning that there will need to be a way to store the properties of each cell, most importantly if the cell is water or land. After consideration, a map would be most appropriate here, as we do not want to intrude on the Voronoi library used, as it would make it difficult to distinguish written code from the library's original, as well as present issues with updating the library in the event of a necessary update (potential new useful features that haven't been considered). Therefore, a way to store the properties for each cell needs to be established completely independently from the libraries' cell class.

After careful consideration, we decided that a cell property class would be appropriate to store Voronoi cells map properties. This map property class will have two variables, altitude, and tile.

- Altitude - the altitude determined by Perlin noise
- Tile - 0 represents this tile as water, 1 as land

Deciding to include a tile integer to represent the type of tile was considered heavily. Ultimately, it was decided that including a tile variable best aligned with objective 6, allowing for the inclusion of more tiles to be added to the engine at a later date would best align to make the foundation of a deeper and more impressive game. The altitude determines the tile type, it is either 1 if the altitude is above an arbitrary "sea level" and 0 if below, allowing this to be extended to include more varied types of land fits well with the objective but also allows for quick checks to see if the tile is land or water, rather than comparing the altitude to the "sea level" set in code.

Originally the plan was to connect cities that are close together, an idea presented itself that somehow had not been previously considered. A set of few simple rules would dictate which cities have roads leading to each other, primarily the closest cities would always be connected. This would result in inefficient road design that created a mess nonsensical roads that would never be constructed as it was in a realistic world. Paths crossing, and generally, there being a general pattern of hideous road design. Then the idea of reapplying Voronoi onto the cities as a new set of source points to generate a city-based Voronoi diagram, the edges of which would be the paths.

3.2 Creating an Engine

We will be using **CMake** to build our project as we can develop on both Linux and Windows and using **CMake** will allow me to jump between the two

easily. It also allows me to easily work on **vscode** rather than **Visual Studio**, which is our preference.

As I'm not making a game, a simple class structure will work for this project. We have a game loop that will handle user input events, update game logic, and then render the scene. We have created room to extend this, passing delta time (the time between each game's update function call) to the update function to allow frame independent calculations. For this project, however, an update function will not be needed as there is no frame by frame game logic. Now we have a structure to render our graphics on.

The first step to any game engine is the game loop. Within this "while loop" is where all game logic will occur until the game is closed, this file will be called *Main.cpp*:

```
#include <SFML/Graphics.hpp>
#include "Scenes/Game.hpp"

int main()
{
    // Create window
    sf::RenderWindow window(sf::VideoMode(1280, 720), "dissertation-project"

    // TODO create abstract scene class for switching
    Game *game = new Game(window);

    // Delta time clock
    sf::Clock dt;

    // Game loop
    while (window.isOpen())
    {
        // Handle Events
        sf::Event e;
        while (window.pollEvent(e))
```

```
        {
            if (e.type == sf::Event::Closed)
            {
                window.close();
            }

            // Pass event to scene
            game->handleEvent(e);
        }

        // Update
        game->update(dt.restart().asSeconds());

        // Render
        window.clear(sf::Color::White);

        // Render scene
        game->render();

        window.display();
    }
}
```

This is essentially set up for the project, this allows for development to occur independently of this new main method, meaning no further changes need to be done. This main loop consists simply of a clock to count the time between game updates to allow for frame independent gameplay and timers, the window is initialized in *1280x720p* as this is a *16:9* aspect ratio that is small enough to easily view the console (for debugging) and also be easily scaleable to a higher *16:9* resolution as needed, which is the most universally used aspect ratio. The development of the map will occur in the game class. In saying this, the game class was primarily created to allow for drawing of UI elements, entities, and to handle game logic, all of which aren't needed

thus making the game class' function solely that of providing a good point to extend the game, aligning with objective 6.

With the creation of the game loop, we finally have something to compile! This means writing a "*CMakeLists.txt*". As we have worked with *CMake* and *SFML* before, we are familiar with how to set up the *CMakeLists.txt* file, and build the project file to compile the program. Opening the newly compiled game engine will result in an empty window. Library implementation is done as needed into the *CMakeLists.txt*.

3.3 Implementing a Voronoi Canvas

The Voronoi diagram will be generated using **Fortune's Algorithm** which is a line sweeping algorithm that calculates the intersections of arcs that are based around a set of points given to the algorithm [14]. An implementation of this algorithm is complex and would take a lot of time to do and as such we will be using a Fortune's Algorithm and Voronoi library for C++ called *mdally/Voronoi*. This library comes with another algorithm known as **Lloyd's Relaxation** which will also be very useful. **Lloyd's Algorithm** is an algorithm for generating centroidal Voronoi tessellation which is a Voronoi diagram where the seed (the source points we provide **Fortune's Algorithm**) is also the centre of the cell. These are generated through multiple iterations of this algorithm, it takes a Voronoi diagram and runs Fortune's Algorithm using the centres of each cell as the seeds [15]. Now, a perfectly symmetrical Voronoi diagram would defeat the point of using them, which is asymmetry, but truly random Voronoi diagrams can result in the creation of very small Voronoi cells which we don't want. Running **Lloyd's algorithm** a few times will create more reasonable cells. The idea to use **Lloyd's Relaxation** to relax the Voronoi diagram came from a blog post we read in our literary review [11].

Now, to generate some random points to apply to **Fortune's Algorithm** and **Lloyd's Relaxation** we will be working in a *1280x720* area, as this is the screen size and with some testing in this canvas area, determined that 10000

seeds create a Voronoi diagram with a reasonable amount of detail. One relaxation using Lloyd's algorithm seems to be enough, these two numbers can easily be changed later.

3.4 Applying Perlin Noise to the Voronoi Canvas

Again for Perlin Noise, I'm going to use a library. **Reputeless/PerlinNoise** is the library we will be using for this. For this library, we simply provide an X and Y coordinate to the library and it will return a height value, we will divide our X and Y by 120 as we found it to be the right level of complexity. Now, we can generate the height for the map we need to store each height for each map, we do this with a variable type coincidentally called a map, which is a variable that can store a variable by a key, the key can be anything and in this case, it will be the Cell variable type provided by **mdally/Voronoi**. In this map we are storing our variable that we called a MapCell, it stores altitude and a tile integer which represents what the tile is (water/land/mountain and can be extended because of this).

The true difficulty of this is drawing to the Voronoi Canvas. We have our Voronoi diagram and our cells, now if we want to draw a tile we can just render the boundaries of the Cell that the library gives us, but we only want to outline a landmass that makes things significantly more difficult. We will store a coastal vector that stores each Voronoi edge that appears on the coast of landmasses. To get this edge we iterate over the edge of each cell, each edge has 2 cells that neighbour it, if one of these cells island and the other is water then we can say that this is a coastal edge and add it to the vector to be drawn.

We decided to draw parallel lines across from these outline drawings, this will mimic the rough, hand-drawn style that comes with a hand-drawn map. It also mimics a coastal mapping technique in which the lines are drawn parallel.

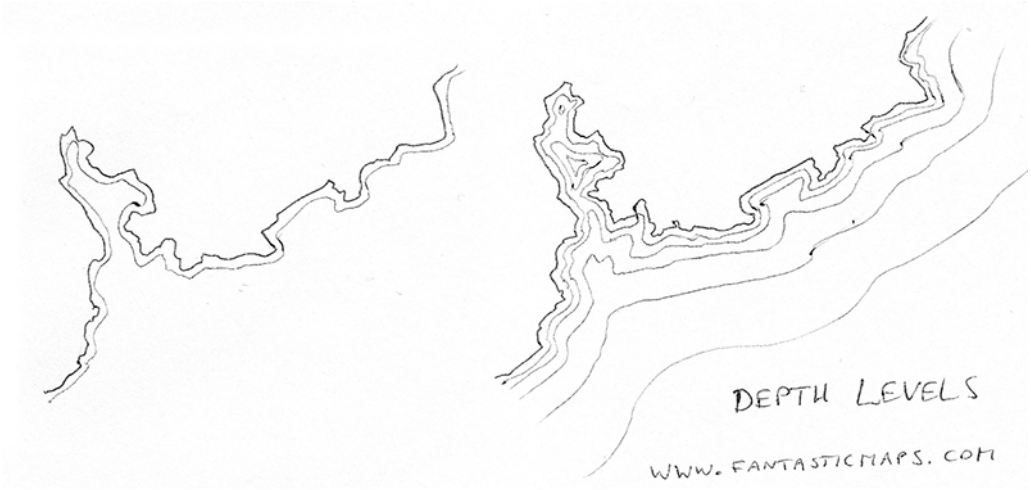


Figure 3.1: Shows an example of a map outline

With this result, the map looks like it is larger than what can be seen, with massive enclosed lakes that might not make much geographical sense. It needs more refining:

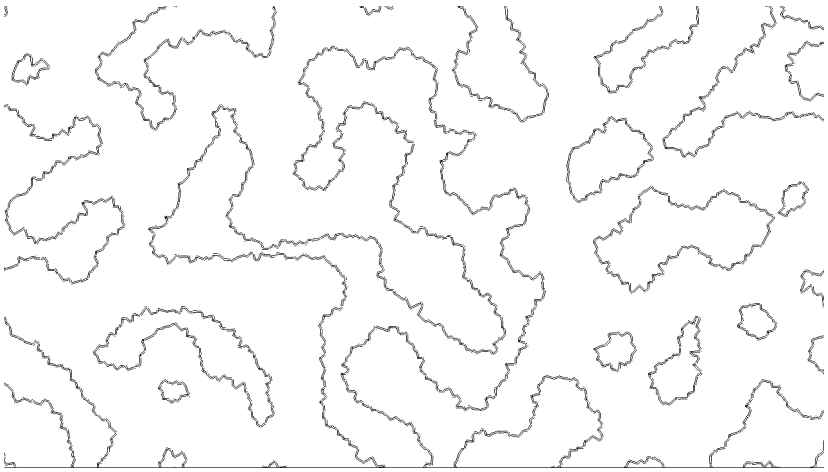


Figure 3.2: First example of map output

So, how can this initial output be refined? We want to have the landmasses to be constricted towards the center of the screen, this way there won't be any generated terrain that isn't completely shown to the user. The solution: modifying the altitude by the distance from the centre of the screen:

$$a = noise(x, y) - \frac{\sqrt{(x - (width/2)) + (y - (width/2))}}{300}$$

300 is a number that we found through trial and error, it seemed to create a consistently sized island that filled the height of the screen without leaving it. The results are very promising and we are satisfied with this number :

Figure 3.3: Finished Result of the terrain generation

The result contains lakes and islands and looks like it would work well for a game world. Islands are close to the mainland and there are no stray pieces of land. We can add some colour to this map to ensure that the altitude is still distinguishable:

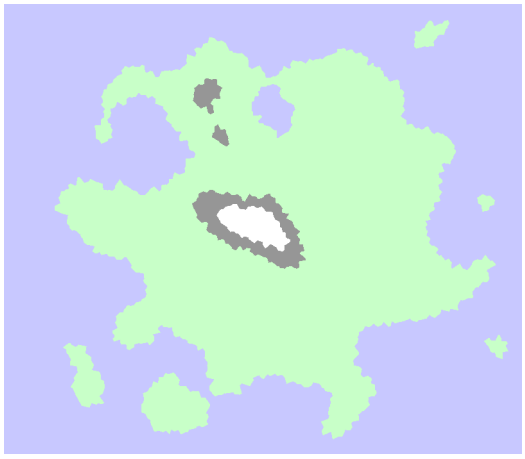


Figure 3.4: Coloured Map

We were surprised and shocked at how well this coloured map turned out, it creates a stylised map that could be used in a game with a bit more tweaking. My aim has been a more ink-drawn black and white map but this is already a good result.

Finally, we bound our "R" key to regenerate the map. This is for debugging purposes. This led me to discover that there was a memory leak in our

program. We were not de-allocating the pointers in our map variable, this meant that every time that the map was regenerated, the amount of memory that our program was taking up was increasing indefinitely. This would eventually lead to a crash which would not be ideal. We created a destructor for our map class that would de-allocate the pointers that were opened during our map's creation.

3.5 City Generation

3.5.1 Markov Chain Name Generation

The generation of truly unique names would be difficult. We have decided to write a header library to do this, which is a library consisting just of a C++ header file. It's the simplest type of library as it can be added to a C++ project very simply, we decided to do this so that we can reuse this code in later projects.

Constructing this header library we referred specifically to the blog post we found in our literary review [7]. In the example given, only a few source words are used, we wanted to find a source of a great many numbers of names so that a great deal of variety could be seen between the city names. For testing, we decided to look online to see if any pre-compiled name lists had been created and stumbled upon *dominictarr/random-name* [16]. This contained a file *fist-names.txt* which was a list of 4945 names, we had decided to use this vast list as a benchmark for our Markov chain generator. We had planned to change this file at a later date.

Now for the development of our algorithm, We decided that we would not be using *json* files to store our Markov chain, unlike the blog post [7]. This is for a few reasons:

- Reading *json* files would require me to write an interpreter or install a library

- The Markov chain generator would only load the words into memory during the initial generation of the map, once the map is generated it would be unnecessary to store the words in memory

- Reading text files it typically very fast, even at this size

- It's much simpler for me to write the Markov library

To store our Markov chain letter frequencies, we needed a data structure. Luckily we already have been using a data structure perfect for this in our Voronoi generation, that is a C++ *map*. The variable contains rather a lot of templates, which is where you dictate the variable type that a C++ structure is stored in. I'll show the final variable and break it down:

```
std::map<std::string, std::vector<std::pair<char, int>>> markov_map;
```

We fetch data from this map using a string (the key), this string will be the initial letters that we want to find the probability of its proceeding letter. When we use our string as a key, a vector (resizable array) of the following letters is returned as well as their frequencies. This is returned as a pair, which is a variable that stores two other variables (determined again by templating) one of which is a character (the next character) and the other an integer (the frequency at which that character appears. Data from this variable is accessed as follows:

```
auto v = markov_map["al"];
v[0].first // get the character
v[0].second // get the frequency
```

With the variable to store our data it was time to write some functions to use outside this class:

- void add(std::string input): takes a string and adds every word in it to the map

- std::string generate(std::string beginning = "", int min = 2, int max = 6): returns a generated word, can take a word size range as well as the start of the word if the user specifies them

The implementation of these functions was surprisingly simple. We just iterated through the letters, storing them in the map and increment the frequency as it appeared, there was some difficulty in ensuring that duplication doesn't occur in the vector, but other than that it was simple to implement. The first letters of the word are selected by randomly selecting a map key.

Using these two functions we were able to generate some names, unfortunately, we ran into an issue that ruined the results; there was a chance of two consonants appearing at the start of a word, which is not ideal. This led me to create a function to test if a word was a vowel so that we ensure that the start of each word has to have a vowel in either the first or second letter. This led me to create a function that checks if a character is a vowel:

```
bool isvowel(char c) {
    return (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
}
```

It's a rather simple function, but it's all that's needed. Using this we can loop our random key picking until one of the first two letters of any word is a vowel. It can be either letter if you think about names they can start with consonants or vowels, but two consonants result in a name that is very difficult to pronounce.

Here are the results of our generation:

```
Zila
Uzalis
Dulene
Hariga
Jellorg
```

And we were pleasantly surprised when the results of this generation were pronounceable and also foreign-looking names. We were satisfied with generating from the name lists, after considering using city names as a source for the Markov chain generator, we realise that many cities come with suffixes and prefixes like "New" which would be difficult to filter out all of them. Similarly, when you consider that many cities are named after people, it makes sense for this fantasy world to have cities named after these fake names that we have generated.

3.5.2 Adding Cities

For rendering cities, we had decided on a circle with the city's name next to it, nothing complicated. As for placing the cities, we ensured that cities would not be generated on ocean tiles as well as ensuring that they were a minimum distance from each other. Finally, we render the city names with a nice fantasy font. The result of rendering our cities into the game world is this:

Figure 3.5: Cities implemented!

Implementing the city names into our generator and viewing a few different generations of these generated cities, there seemed to me to be a diverging style of a city name, the phonetic ones that look as though they could belong to a phonetic language (Welsh is the immediate counterpart) with cities such as "UNFIN" (which can perfectly translate to "an angry one", following naming conventions found in welsh) or "AWNYB", whereas others seem more anglicised, like "GABE" or "ZONIEDY", not all of them are so easily categorised, but the division between words that can be pronounced as if they were English words versus the words that clearly can only be pronounced phonetically as if they were Welsh, these two are the two that stand out and it gives the illusion (as nothing intentional is happening, it's just a coincidence that the Markov generation developed appears to have patterns like this) of a cultural difference. This is definitely to do with the variety found in our Markov chain source names, those names being derived from different languages and cultures and therefore developing light patterns. This was a pleasant surprise as a lot of modern Fantasy worlds have languages heavily inspired by Welsh, such as Elvish in Lord of The Rings or the magic in The Witcher 3: Wildhunt [17].

3.5.3 Road Generation

The design idea to apply the same *mdally/Voronoi* library to connect the cities into roads was an idea that presented itself as a simple one but brought with it a few unforeseen issues. The library was a big problem, it had very little documentation and the documentation that it did have was all written with the code, making it difficult to navigate, which meant diving into the libraries codebase. A misleading and odd C++ feature that caused some problems was **friendship**, which is quite an odd feature that allows a programmer to define specific classes to access other classes private methods. The first attempt at creating a Voronoi diagram not affected by the bounding box required to be passed as a variable to **void compute(std::vector<Point2> sites, BoundingBox bbox)**. This is because the BoundingBox will connect the Voronoi cells to a bounding box which you are required to set. Thus an attempt was made to essentially mimic the function requiring a Bounding-

Box, but having it simply add the points rather than calculate out of bound nodes. It was decided to do the implementation of this boundless **Fortune's Algorithm** inline as the cities are generated, rather than directly edit the library, as previously stated this would be a bad practice. Copying the *VoronoiDiagramGenerator.cpp* class, an error occurred accessing a method for the *Diagram* class, this error simply stated that this function was inaccessible, which was very confusing because *VoronoiDiagramGenerator.cpp* used the same method with no issue. A difficult amount of time passed, and while venturing through the header file of *Diagram.h* (a lot of time was spent looking through *Diagram.cpp*), where the concept of friendship was brought to attention, it essentially means that specified classes can access private and protected scopes of another object by being assigned as the objects friend which meant that this attempt at getting around the BoundingBox was an utter failure and an educational waste of time.

It turns out that passing an empty BoundingBox would result in the exact result sought after, which was very frustrating. The library could have handled a few things differently. Firstly, after some research into use cases for friendship in C++, reading a paper on the matter [18], which laid out a set of specific use cases for friendship, the first case of which this instance met as *VoronoiDiagramGenerator.cpp* can be changed by reckless calling of private diagram functions and the reckless modification of the private variables in the *Diagram* class. This issue could have been resolved by simply designing a better system of architecture. Notably mentioned in the paper is the fail rate increasing as a result of coupling (friendship) as well as more critically for a library, the fact that it makes the code simply harder to understand, which along with the lack of real documentation is what led to a large amount of time being wasted when all that was required was to call the function with an empty constructor. A library should at least try to make an intuitive code base, and cutting library users off from many private library functions just seems like an odd decision when developing a library, why cut a user off from functionality? Another solution that wouldn't require any re-engineering of the library architecture is to have the BoundingBox variable passed to the compute function have a default argument, making it clear that you can leave

this argument blank to ignore the bounding box (this was only discovered through trial and error with the library).

This section of the project is probably the one that has encountered the most problems, but this has resulted in a successful Voronoi road system being generated, it is highlighted in red:

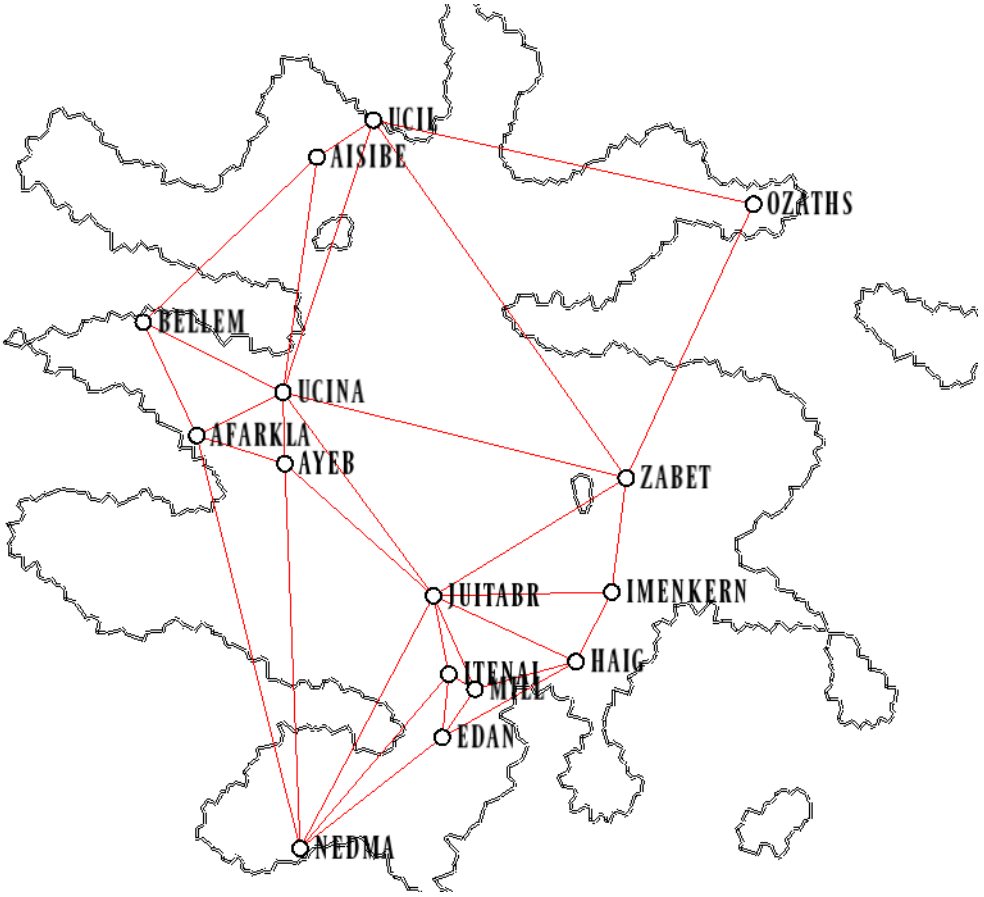


Figure 3.6: Demonstration of the Voronoi road system

3.6 Adding Detail

3.6.1 Adding Mountains

One detail that we wanted to add to be truly faithful to traditional fantasy maps was to generate little pictures of mountains in the world. These pictures would create mountain ranges and give an idea of altitude without literally rendering the height on the terrain. To do this we created a vector of mountain positions, everywhere that there was an altitude above an arbitrary limit (set in code) we would add a mountain to our vector. This, of course, created an unrecognisable mess wherever the altitude was above a certain limit, so

we add a check to see if a mountain was close to another mountain before adding it to the vector, meaning that mountains appear well-spaced in the final result.

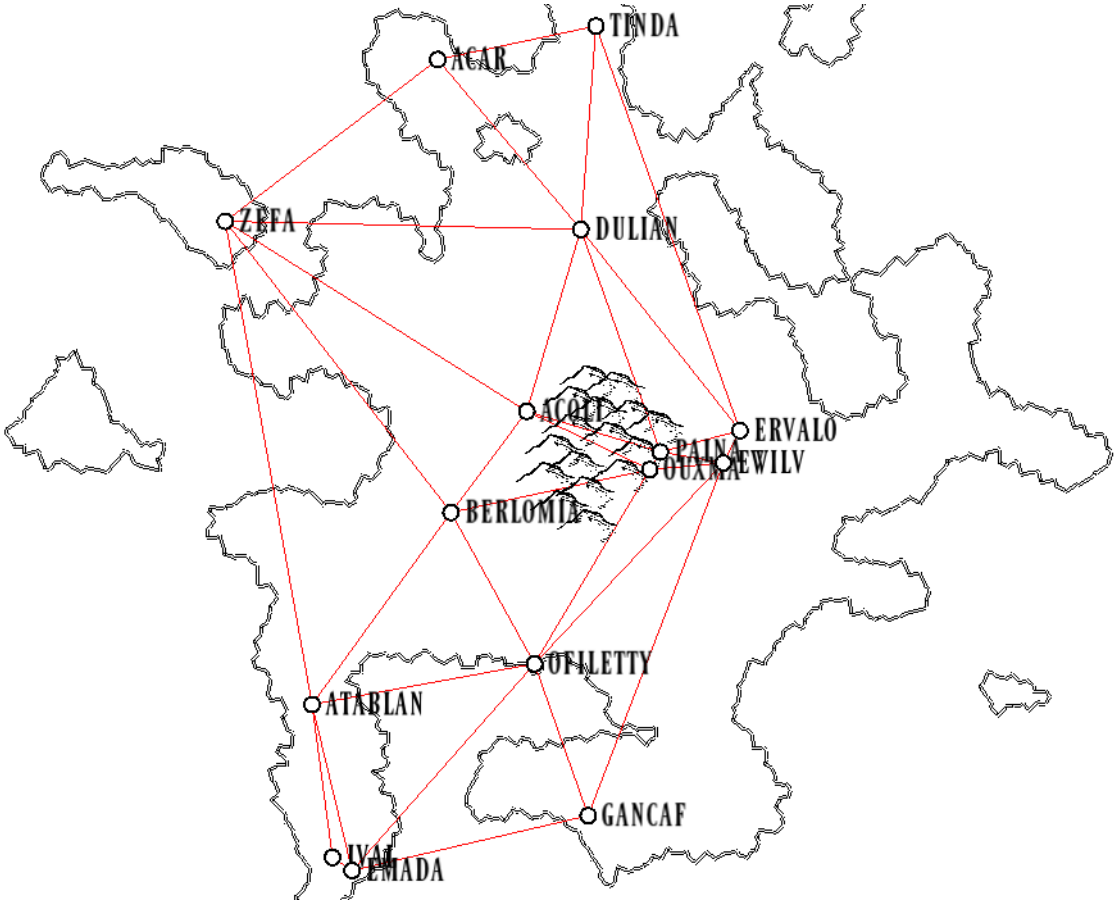


Figure 3.7: The first mountains on the Pangaea

3.6.2 Adding Trees

For adding trees to our landmasses we follow the same approach for our mountains, adding little circles that were well spaced. But where would we add these little circles? Running another Perlin Noise generation over the island generation and did the same thing as the mountains! If the "altitude" was above a certain height this was considered a forest on our map, and if this forest was also a land tile it would render trees!

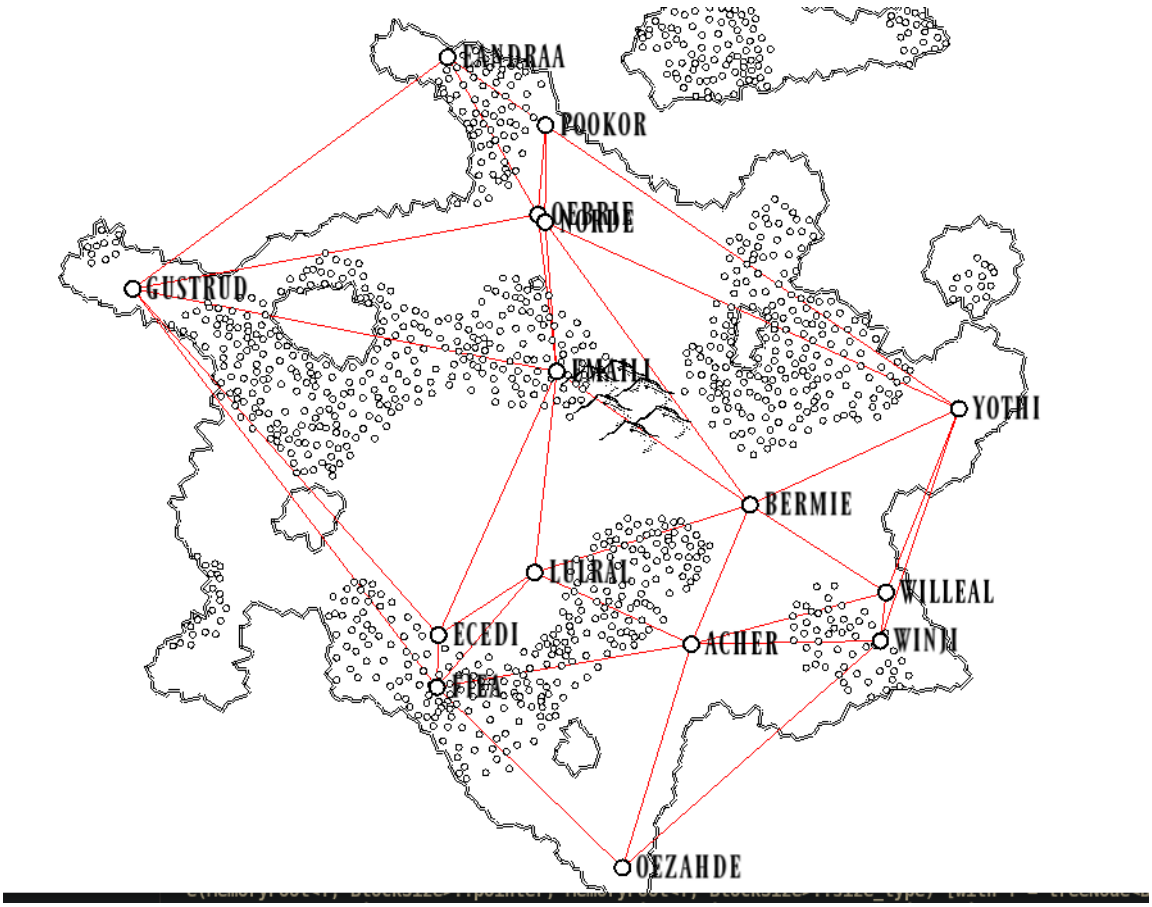


Figure 3.8: Forests are added!

Chapter 4

Results

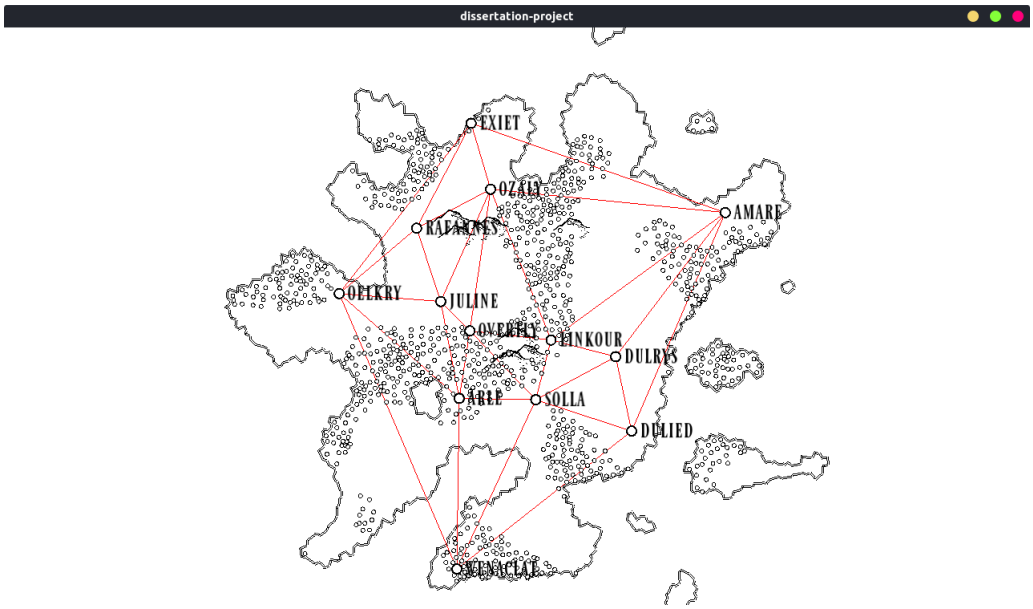


Figure 4.1: The final result of the map generation

This is the final result of our map, it's recognisable. We believe that procedural generation techniques have been applied to a high standard. As far as an aesthetic choice might go, I am very fond of the black and white ink map, it is the style of the original Tolkien maps, some may not find this desirable if they were using this map as the foundation for a game, to demonstrate the depth of the customisation possible, as some smaller features that have been added.

Figure 4.2: A display of mouse wheel scrolling, now implemented.

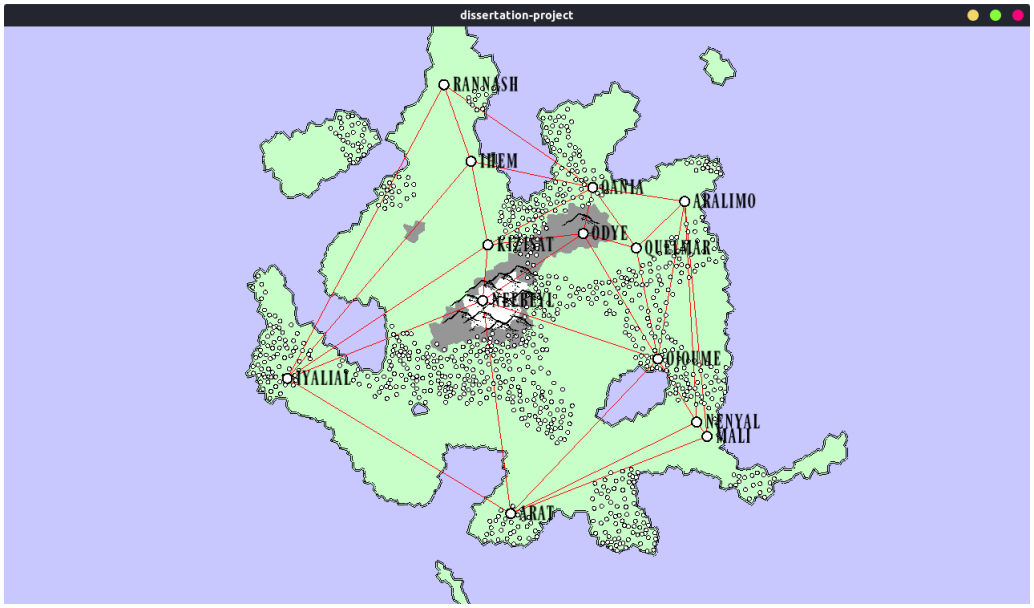


Figure 4.3: Coloured rendering on the updated, more detailed map. It may look even more appealing with some tweaking to the trees and perhapse disabling mountains

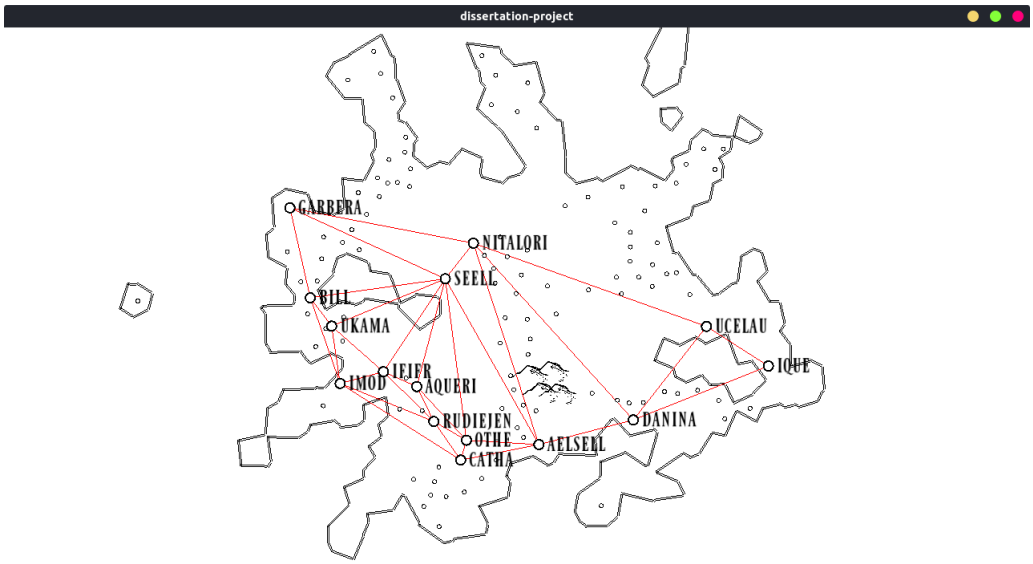


Figure 4.4: Reducing the number of randomly placed points for our Voronoi diagram's generation will result in the generation of a low-poly map

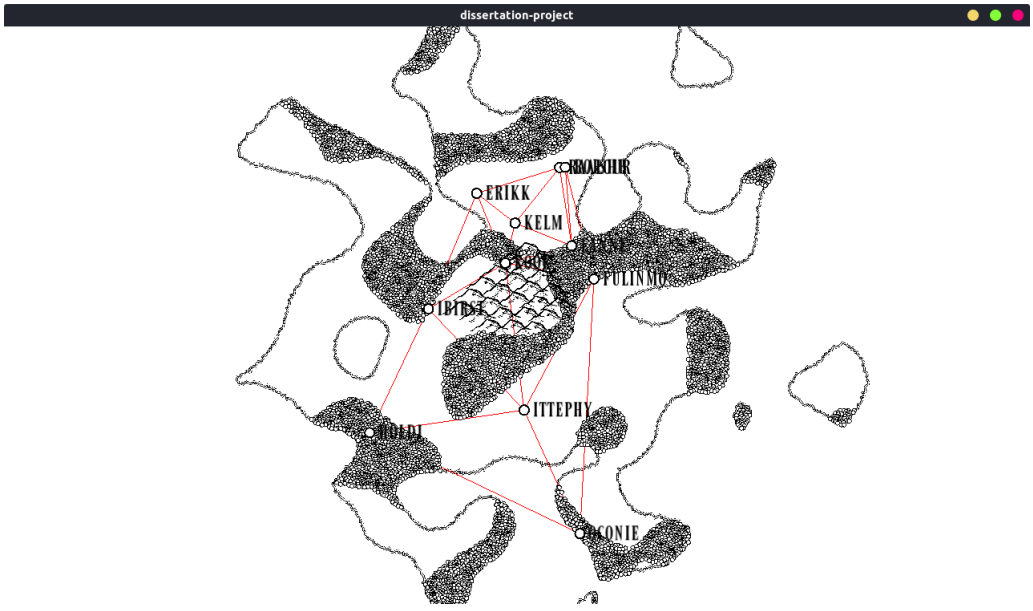


Figure 4.5: Multiplying the source points for the Voronoi diagram by 10 seems to be TOO high quality. Tree placement relies on the cell centres being well spaced apart, when the cells are small like this it results in forests with no visible land, it may not look like an island anymore, but it is very striking

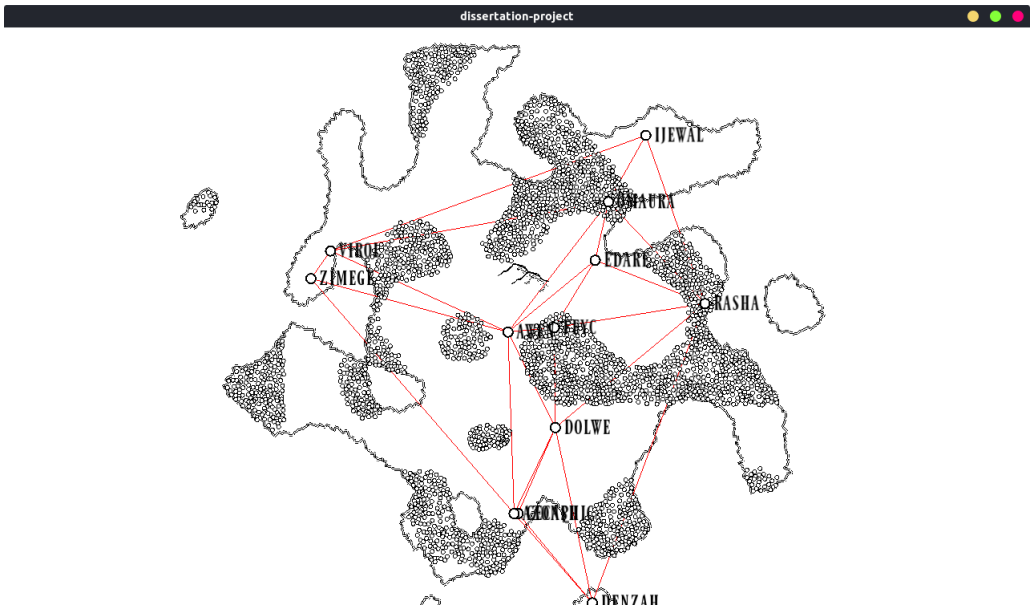


Figure 4.6: Multiplying the resolution by 3 seems to work better, although the significantly slower run and start time don't seem to be worth the very insignificant level of coastal detail

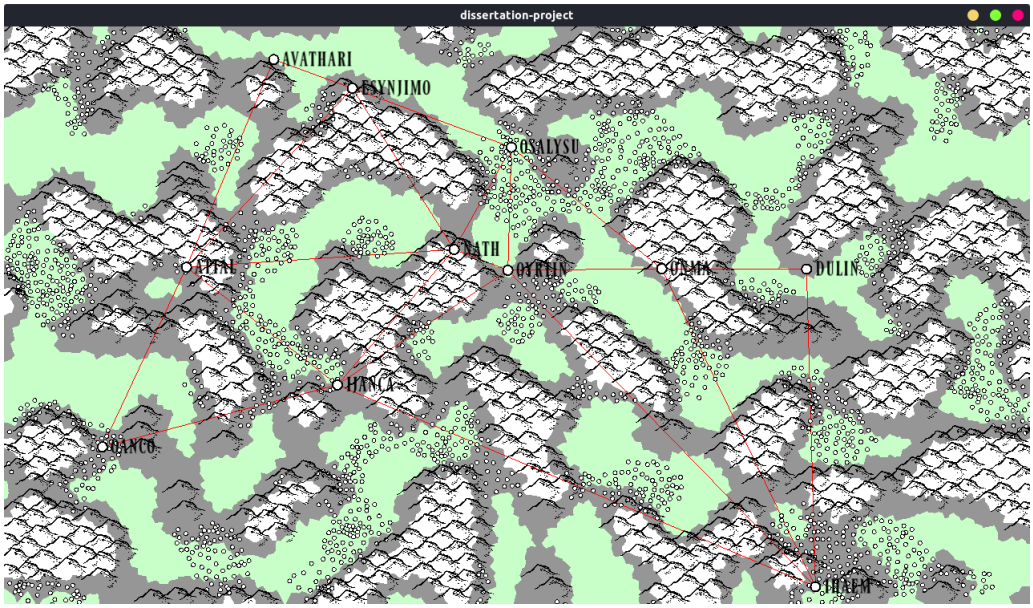


Figure 4.7: Increasing the island size is relatively simple, here the distance check was removed all-together meaning that the whole screen will end up being land, the Voronoi canvas is the same size as the screen by default, if the need arises to have maps larger than this it is just a matter of increasing the canvas size.

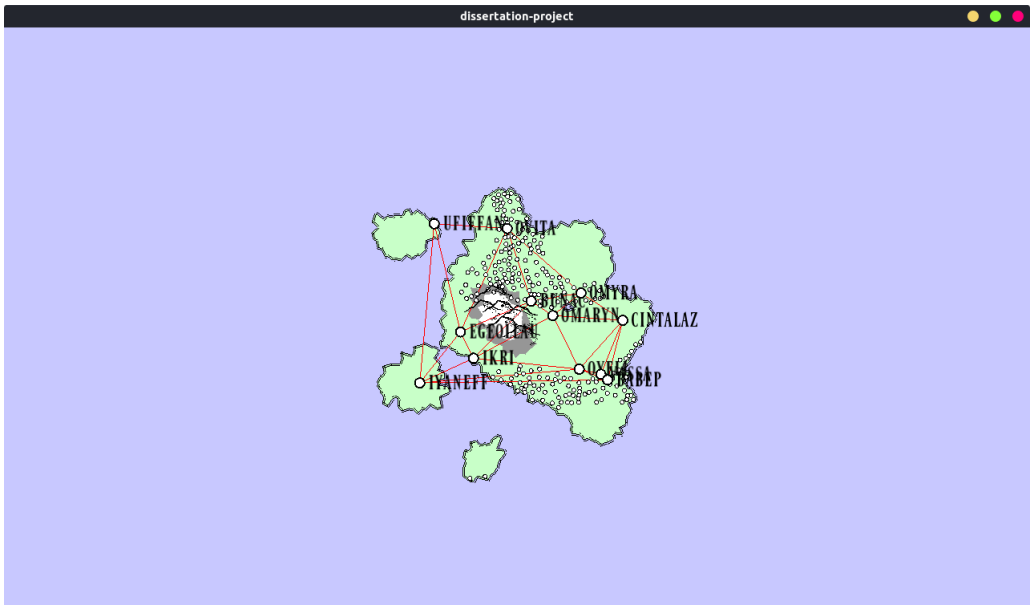


Figure 4.8: Generating tiny islands is possible too!

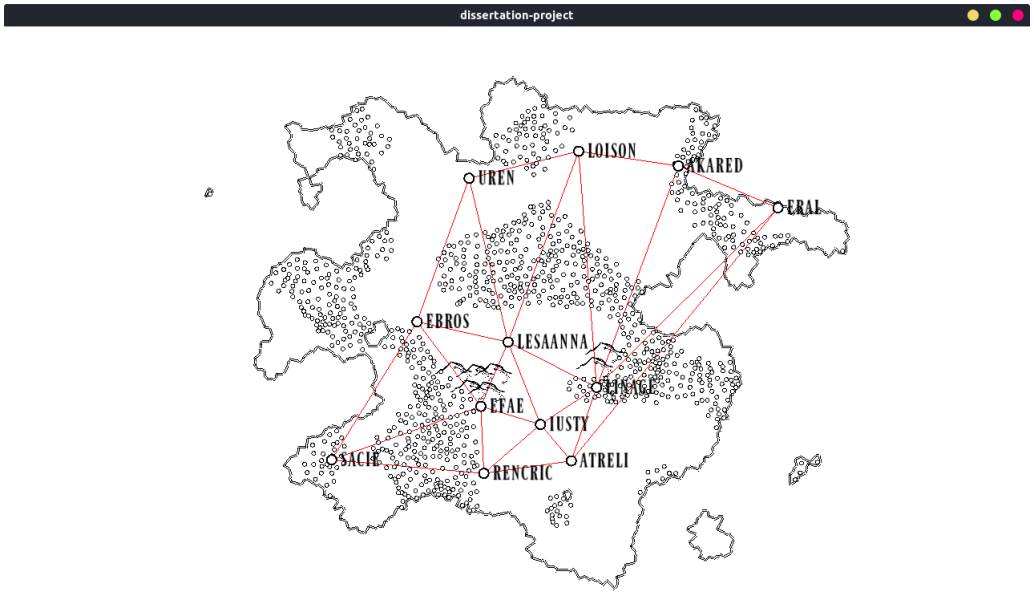


Figure 4.9: I do believe that my vanilla map is the best looking however! This Pangaea seems to be very

Chapter 5

Conclusion

In conclusion, we believe that we have successfully achieved every aim and objective set out for ourselves at the beginning of this project. We have successfully generated a Pangaea of far greater complexity than initially intended. We had thought that my map would be constrained by a tilemap but the literary review led me to research Voronoi diagrams for my map, resulting in a realistic coastline that mimics the irregular pattern that would appear in real-world coastlines. Using the Voronoi map as a canvas, we applied Perlin Noise that shaped the island (with a modifier to give the altitude a bias to the centre of the Voronoi diagram, generating a tidy Pangaea) without being bound to a tilemap. Using Voronoi diagrams has brought the unforeseen advantage of having infidel detailed maps, as the coastline is drawn as a vector shape, the user can zoom infinitely without the loss of resolution. A vector shape isn't based on a pixel grid-like traditional graphics, it is based on floating-point coordinates, and as we redraw the map every frame it will always draw the same resolution lines between these fixed points defined in the Voronoi diagram. The placement of cities only required us to ensure that they are properly spaced, meaning that we just had to check that this city location was distant from nearby ones.

For name generation we wrote a simple Markov chain header library, this was surprisingly simple and required a lot less code than we had expected, for great results from my source set of 5,000 names, generating interesting and genuinely unique names that can always be pronounced. One issue that we had not expected was that sometimes the randomly selected starts of these city names could generate with two consonants, meaning that it

would be difficult to pronounce, this had me create a function to check if a character variable is a vowel or a consonant, and ensure that at least one of the starting two letters was a vowel. We implemented a random length for these generated city names and the variety was surprisingly interesting.

As for the coincidental ease at which some of the generated words were pronounced in welsh, a hypothesis of why this is occurring took a great deal of time and thought. The best conclusion that we have devised would be that the generated city names tend to feature a large number of consonants that appear in a non-phonetic language, to make it difficult to pronounce in English, whereas Welsh's frequent use of oddly placed consonants (relative to other languages) and a wide variety of unique "letters" that occur in cases where consonants are placed next to each other lead to a word that simply works better in Welsh. Part of this perception of welsh words is definitely in part a form of Pareidolia [19] (likely as a result of speaking welsh) to project Welsh pronunciation onto these new words. We are unsure if others who don't speak welsh would assume that it is simply a badly generated word rather than a city name derived from another language (even if accidentally), but the fact that we have not found a word that is impossible to pronounce in both languages, the "perceived" pronunciation (as these are usually truly new words, therefore, lack a correct pronunciation) has almost always sounded appropriate to one, and in the rare case that the generated word does not fit either Welsh or English, it can always be claimed to be some guttural orcish or another fictional being. A simpler, and the less exciting answer would simply be that as a phonetic language, Welsh can pronounce a wider variety of words, and although the flexibility of Welsh's phonetic ability and capacity to deal with odd variations of double consonants expands the number of words that I as a welsh speaking person would find as phonetic, although it is an interesting thought that a good generator for me would be an awful generator for someone who simply cant speak welsh. Although Pareidolia is probably the most reasonable explanation, the fact that a fantasy map City name generator, trained from a completely random source of names that were found on a Github repository [16] lead to names that we perceived as

welsh, for a fantasy map generator (a genre whose most famous language is based on welsh) is just really interesting.

The Voronoi road system was a mess, the Fortune's Algorithm library being used had awful documentation (definitely something that should have been considered), which brought to my attention some very odd design principles with the construction of this library, it using a rarely used and generally discouraged (it, of course, has use cases) programming concept of led us to the conclusion that the library had some serious flaws [18]. Developing with *mdally/Voronoi* was an unforeseen hassle, and is something that would change on a second attempt (and would try to replace with a custom Fortune Algorithm implementation if developing this project further) even with the notorious difficulty of implementing Fortune's algorithm, after religiously combing through *mdally/Voronoi* a general understanding of the algorithm has developed, the library has overlap with SMFL for some objects, the most glaring fault being that of the constant conversion of the Voronoi library's Pointer2 and **SFML**'s sf::Pointer2f, using the **SFML** variable in the development of a library or having support for casting between the **SFML** 2D point and a custom one by implementing some operators between the two. Another noteworthy thing about a custom library for Voronoi generation is that there are algorithms that simply aren't as difficult to imp lent as Fortune's as well as other sources of Voronoi libraries to be gotten. Having no experience with Voronoi generation meant that the use of this library was slow going, and by the time of encountering the unappealing issues with the road generation development was well on its way, having implemented the Voronoi Canvas already.

Forests meant spreading dots along with a second set of Perlin noise, meaning that we can just run Perlin noise over my map and if the altitude was greater than a specified amount, it's considered a tree cell, this also ensures that the cell is, in fact, a land cell, meaning that trees will never appear in the water. Repeat this process and ensure good spacing among the trees and it generates an identifiable forest.

Sprite rendering for mountains we create a vector of mountain locations, of course, we ensure proper spacing between these locations as the mountain sprite that we are using for generation is large, as to be expected from this style of map. The resulting mountain ranges that captured the targeted style that was aimed for by this project.

As for being a good foundation for a future game, we left ample room for adding complexity to the engine. Entity systems would need to be made, other features that may be included in an engine like Unity will need to be developed from scratch or implemented from a library. Previous experience with game engine development means that this hurdle will be surpassed without issues that have arisen in previous projects. Perhaps using a more widely used engine such as Unity would mean that a developer wishing to use this generator would have far less difficulty, but we believe that the increased productivity and quality of the written code is worth this sacrifice. We also believe that presenting my project in this manner makes it easier to gain an understanding on its functionality, as there are no hidden attributes hidden in a game engines file system, which has previously been a barrier to understanding similar projects. Essentially, code is easier to understand than an unfamiliar game engine and therefore will make the code more accessible to all.

Therefore, we have completed each of my objectives to a reasonable degree, far surpassing my initial goals for generating an island and generating maps of a quality much higher than anticipated. There is room for more fine tweaking, a more artistically minded person would be able to tweak this generation system to get more desirable results.

5.0.1 PESTLE

- Professional - Can be applied to game development. Will definitely be used in my portfolio.

- Ethical - No current ethical impact, as a foundation towards works that do explore ethical questions and dilemmas and that perhaps that this could someday encroach on that.

- Social - Fantasy also often explores Social themes, even if currently this map generator does not.

- Technological - Interesting implementation of varied technologies.

- Legal - No relevant legal aspect, just a dark lord.

- Environmental - Could be developed into some form of an environmental simulator, simulating rising sea levels and other concepts.

Chapter 6

Future Work

Given more time here are features that would be focused on:

- Implement rivers and erosion, this should include bridges for advantageous crossings near cities.
- Generate named regions and territories. Name landmarks like notable mountains and rivers should have names too.
- Implement roads conforming to the Voronoi diagram
- Intelligent agents to bring the world to life. This is out of the scope of the project, but it would be really interesting implementing agents moving about to fulfil their factions needs. Trading, farming even war and raids would be a really interesting project.
- A gameplay loop. Again out of the scope of the project but the eventual aim was to get a game based in this map. The initial plan for this map was a real-time strategy implementing this map generator, it would be expanded on, implementing gameplay elements such as resources, units, and upgrades. This would need a lot of thought and design.
- History Generation. Generating more depth and complexity in my world would distinguish it from contemporary procedural content.
- developing a custom implementation of Fortunes algorithm with mapping classes in mind to streamline the process and overcome the issues

had with the Voronoi library. All Voronoi cells should store altitude and tile type default.

- Biomes adding variety to a map.

- A sense of time in this world, using shaders perhaps to display day and night in the world.

- Continental generation rather than generating a Pangaea.

- Factions, cities should have a hierarchical structure, with cities reporting a capital city.

- City name generation should have different sources for its Markov chains, meaning that factions have a similar naming process across their cities.

- An alternative to some of these gameplay focused suggestions would be to have competitive neural networks develop on this procedural terrain.

- Experiment with more varied altitude generation for more interesting maps.

- City prefix such as "New" and suffix such as "City" to mimic real cities. Low chance of this occurring

- Multiple words being generated for the city names with Markov chains, this would bring many difficulties as ensuring that words are phonetic would be difficult as the name of a place increases the number of characters. e.g there are two words to ensure both have at least one vowel as their first two letters. Other difficulties would arise like the length of each of the words needing to surpass one character, ignoring the maximum character limit if need be.

I think this project is a good foundation to launch other similarly themed projects. The code can at the very least be reapplied for other uses. We are passionate to pursue the possibilities that this foundations project can provide.

Bibliography

[1] *Stack Overflow Developer Survey*. URL: https://insights.stackoverflow.com/survey/2019#technology.

[2] *Simple Direct Media Layer*. URL: https://www.sfml-dev.org/.

[3] Blendo Games. *Atom Zombie Smasher*. URL: http://blendogames.com/atomzombiesmasher/.

[4] *KeeperRL*. URL: https://keeperrl.com/.

[5] Ken Perlin. *An Image Synthesizer*. 1985. URL: https://dl.acm.org/doi/10.1145/325165.325247.

[6] Markus Persson. *Terrain generation, Part 1*. 2011. URL: https://notch.tumblr.com/post/3746989361/terrain-generation-part-1.

[7] Gustavo Zomer. *Generating Startup names with Markov Chains*. 2019. URL: https://towardsdatascience.com/generating-startup-names-with-markov-chains-2a33030a4ac0.

[8] John Clements. *Generating 56-bit passwords using Markov Models (and Charles Dickens)*. 2015. arXiv: 1502.07786 [cs.CR].

[9] Tanja Eisner et al. 'Operator theoretic aspects of ergodic theory'. In: *Graduate Texts in Mathematics, Springer, to appear* (2014).

[10] Jacob Olsen. 'Realtime procedural terrain generation'. In: (2004).

[11] Amit Patel. *Polygon Map Generation*. URL: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/.

[12] George Kelly and Hugh McCabe. 'A survey of procedural techniques for city generation'. In: *The ITB Journal* 7.2 (2006), p. 5.

[13] O. Sharma, D. Mioc and F. Anton. 'Voronoi Diagram Based Automated Skeleton Extraction from Colour Scanned Maps'. In: *2006 3rd International Symposium on Voronoi Diagrams in Science and Engineering*. 2006, pp. 186–195.

[14] Ivan Kutskir. *Fortune's algorithm and implementation*. 2012. URL: http://blog.ivank.net/fortunes-algorithm-and-implementation.html.

[15] Daniel Jacobson. *Lloyd Relaxation of Voronoi Diagrams*. 2012. URL: https://demonstrations.wolfram.com/LloydRelaxationOfVoronoiDiagrams/.

[16] dominictarr. *random-name*. URL: https://github.com/dominictarr/random-name/blob/master/first-names.txt.

[17] *Elder Speech*. URL: https://witcher.fandom.com/wiki/Elder_Speech.

[18] Lionel Briand, Prem Devanbu and Walcelio Melo. 'An investigation into coupling measures for C++'. In: *Proceedings of the 19th international conference on Software engineering*. 1997, pp. 412–421.

[19] Wikipedia. *Paredolia*. URL: https://en.wikipedia.org/wiki/Pareidolia.

.1 Poster

.2 Code

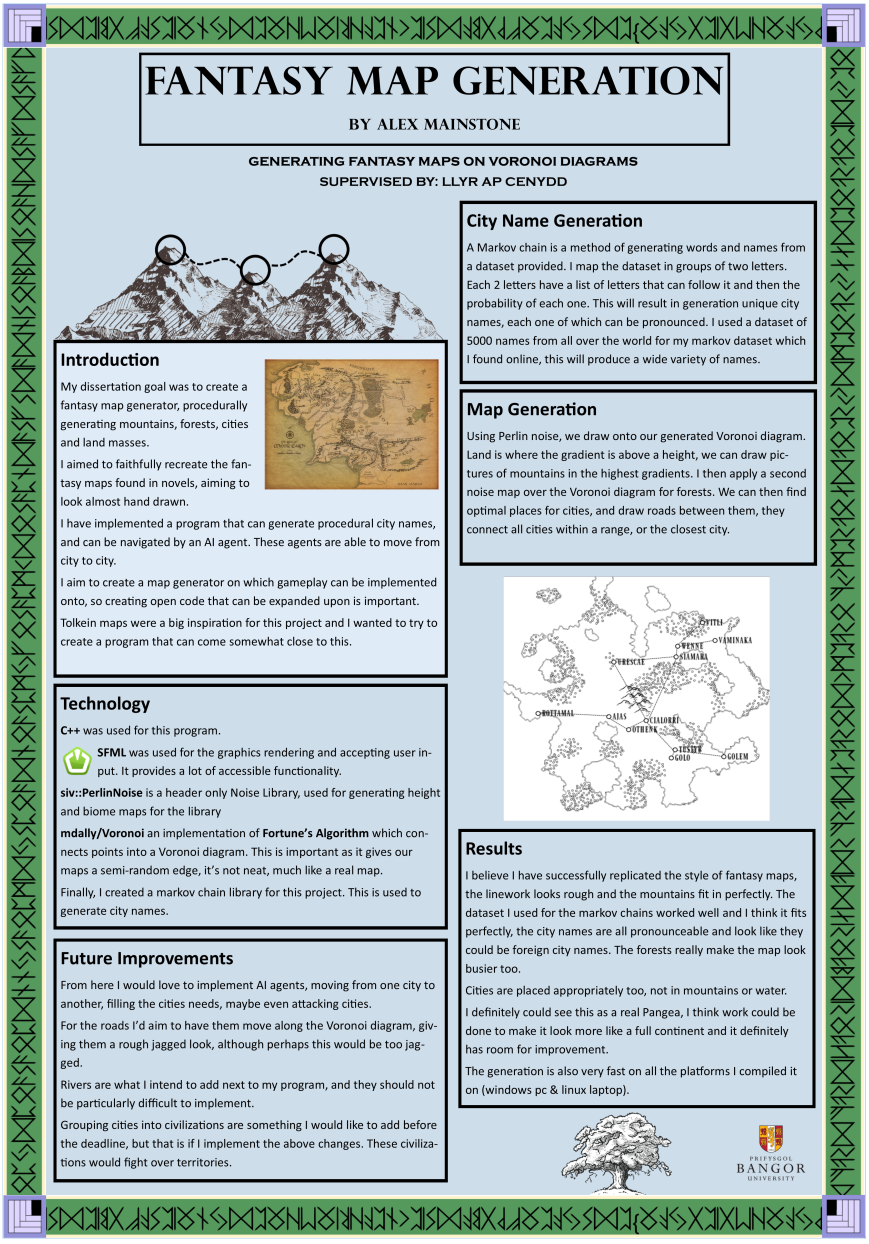
.2.1 Markov.hpp

```
#pragma once

#include <string>
#include <map>
#include <vector>
#include <utility>
#include <sstream>
#include <locale>
#include <iostream>

using namespace std;

class Markov
{
public:
    Markov(int n, std::string input) : n(n)
```




```
{
    srand(time(0));
    add(input);
}

void add(std::string input)
{
    // To lower
    std::string intext = "";
    for (int i = 0; i < input.length(); i++)
```

```
{
    char s = tolower(input.at(i));
    if (s < 97 || s > 122)
    {
        continue;
    }
    intext += s;
}

std::stringstream test_stream(intext);
std::string word;
for (int i = 0; test_stream >> word; i++)
{

    // iterate through word
    for (int j = 0; j < word.length() - n; j++)
    {
        std::string subs = word.substr(j, n);
        char next = word.at(j + n);
        if (markov_map[subs].empty())
        {
            std::pair<char, int> p;
            p.first = next;
            p.second = 1;
            markov_map[subs].push_back(p);
        }
        else
        {
            bool exist = false;
            for (auto &a : markov_map[subs])
            {
                if (a.first == next)
                {
```

```
                        exist = true;
                        a.second++;
                    }
                }
                if (!exist)
                {
                    std::pair<char, int> p;
                    p.first = next;
                    p.second = 1;
                    markov_map[subs].push_back(p);
                }
            }
        }
    }
}

std::string generate(std::string beginning = "", int min = 2, int max =
{
    if (beginning == "")
    {
        beginning = "nn";
        while (!(isvowel(beginning.at(0)) || isvowel(beginning.at(1))))
        {
            auto it = markov_map.begin();
            std::advance(it, rand() % markov_map.size());
            beginning = it->first;
        }
    }
    std::string result = beginning;

    for (int i = 0; i < rand() % max + min; i++)
    {
        auto v = markov_map[result.substr(result.length() - 2)];
```

```
            if (v.empty())
            {
                if (result.length() <= min)
                {
                    return generate();
                }
                break;
            }

            std::vector<char> dev;
            for (int j = 0; j < v.size(); j++)
            {
                for (int c = 0; c < v[j].second; c++)
                {
                    dev.push_back(v[j].first);
                }
            }
            result += dev[rand() % dev.size()];
        }
        return result;
    }

private:
    bool isvowel(char c) { return (c == 'a' || c == 'e' || c == 'i' || c ==
    std::map<std::string, std::vector<std::pair<char, int>>> markov_map;
    const int n;
};
```

.2.2 MapGenerator.hpp

```
#pragma once

#include <SFML/Graphics.hpp>
#include "VoronoiDiagramGenerator.h"
```

```
#include "PerlinNoise.hpp"
#include "World/Map.hpp"

struct MapCell
{
    float altitude;
    unsigned int tile;
    MapCell(float altitude, unsigned int tile) : altitude(altitude), tile(ti
    MapCell() : altitude(0), tile(0) {}
};

class MapGenerator
{
public:
    MapGenerator();
    Map *generate(unsigned int seed = 0, sf::Vector2f size = sf::Vector2f(12

private:
    std::map<Cell *, MapCell *> initCellProperties(Diagram *diagram);

    sf::Texture generateTexture(Diagram *diagram, std::map<Cell *, MapCell *
    sf::VertexArray generateConvexShapes(std::map<Cell *, MapCell *> mapprop

    VoronoiDiagramGenerator vge;
};
```

.2.3 MapGenerator.cpp

```
#include "World/MapGenerator.hpp"
#include "Jezov/Markov.hpp"
#include <fstream>

MapGenerator::MapGenerator()
{
```

```
    vge = VoronoiDiagramGenerator();
}

Map *MapGenerator::generate(unsigned int seed, sf::Vector2f size, unsigned i
{
    if (seed == 0)
    {
        seed = time(0);
    }
    srand(seed);

    // Generate random points
    std::vector<Point2> sites;
    sites.reserve(nodes);
    for (int i = 0; i < nodes; i++)
    {
        Point2 p;
        p.x = 1 + (rand() / (double)RAND_MAX) * (size.x - 2);
        p.y = 1 + (rand() / (double)RAND_MAX) * (size.y - 2);
        sites.push_back(p);
    }

    // Generate voronoi
    vge.compute(sites, BoundingBox(0, size.x, size.y, -1));
    Diagram *diagram = vge.relax();

    // Initialize Map properties
    auto mapprop = initCellProperties(diagram);

    // Generate Texture
    sf::Texture map_texture = generateTexture(diagram, mapprop, size);
    Map *map = new Map(map_texture, generateConvexShapes(mapprop, diagram));
```

```
siv::PerlinNoise noise(time(0) + 1);
// Mountains & trees
for (auto c : diagram->cells)
{
    if (mapprop[c]->altitude > 0.1)
    {
        sf::Vector2f newpos(c->site.p.x, c->site.p.y);

        bool spaced = true;
        for (int i = 0; i < map->getMountainSize(); i++)
        {
            sf::Vector2f mtnpos = map->getMountain(i);
            float dist = std::sqrt(std::pow(newpos.x - mtnpos.x, 2) + st
            if (dist < 20)
            {
                spaced = false;
            }
        }

        if (spaced)
        {
            map->addMountain(newpos);
        }
    }
    if (mapprop[c]->tile != 0 && mapprop[c]->altitude <= 0.1 && noise.no
    {
        map->addTrees(sf::Vector2f(c->site.p.x, c->site.p.y));
    }
}

// Add Cities
std::ifstream t("../res/names.txt");
std::string str((std::istreambuf_iterator<char>(t)),
```

```
                std::istreambuf_iterator<char>());

Markov markov(2, str);

// Road Generation
sites.clear();

for (int i = 0; i < rand() % 10 + 10; i++)
{
    auto randmap = diagram->cells[rand() % diagram->cells.size()];
    while (mapprop[randmap]->tile == 0)
    {
        randmap = diagram->cells[rand() % diagram->cells.size()];
    }
    map->addCity(City(sf::Vector2f(randmap->site.p.x, randmap->site.p.y)
    sites.push_back(Point2(randmap->site.p.x, randmap->site.p.y));
}

delete diagram;
diagram = vge.compute(sites, BoundingBox(0, size.x, size.y, -1));
map->setRoads(diagram);

for (auto it = mapprop.begin(); it != mapprop.end(); it++)
{
    delete it->second;
}
mapprop.clear();
// delete (diagram);

t.clear();
t.close();
str.clear();
```

```
    return map;
}

std::map<Cell *, MapCell *> MapGenerator::initCellProperties(Diagram *diagram
{
    std::map<Cell *, MapCell *> cells;

    siv::PerlinNoise noise(time(0));
    for (auto c : diagram->cells)
    {
        float dist = std::sqrt(std::pow(c->site.p.x - (1280 / 2), 2) + std::
        // if(dist < 200) {
        //     cells[c] = new MapCell(1, 0);
        // } else {
        //     cells[c] = new MapCell(0, 0);
        // }

        float n = noise.noise(c->site.p.x / 100, c->site.p.y / 100) - (dist
        if (n > -0.9)
        {
            cells[c] = new MapCell(n, 1);
        }
        else
        {
            cells[c] = new MapCell(n, 0);
        }
    }

    return cells;
}

sf::Texture MapGenerator::generateTexture(Diagram *diagram, std::map<Cell *,
{
```

```
sf::RenderTexture render_texture;
render_texture.create(size.x, size.y);
// Draw Cells
for (Cell *c : diagram->cells)
{

    auto pos = c->site.p;
    sf::Color col(sf::Color::White);
    if (mapprop[c]->altitude <= -0.9)
    {
        col = sf::Color(200, 200, 255);
    }
    else if (mapprop[c]->altitude > 0.15)
    {
        col = sf::Color(255, 255, 255);
    }
    else if (mapprop[c]->altitude > -0.1)
    {
        col = sf::Color(150, 150, 150);
    }
    else
    {
        col = sf::Color(200, 255, 200);
    }
    sf::ConvexShape shape;
    shape.setPointCount(c->halfEdges.size() * 2);

    for (int i = 0; i < c->halfEdges.size(); i++)
    {
        shape.setPoint(i * 2, sf::Vector2f(c->halfEdges[i]->edge->vertA-
        shape.setPoint(i * 2 + 1, sf::Vector2f(c->halfEdges[i]->edge->ve
    }
    shape.setFillColor(col);
```

```
        render_texture.draw(shape);
    }

    render_texture.display();
    return render_texture.getTexture();
}

sf::VertexArray MapGenerator::generateConvexShapes(std::map<Cell *, MapCell
{
    std::vector<sf::Vector2f> templines;

    for (auto c : diagram->cells)
    {
        for (auto e : c->halfEdges)
        {
            if (e->edge->lSite == nullptr || e->edge->rSite == nullptr)
            {
                continue;
            }
            if (mapprop[e->edge->lSite->cell]->tile != mapprop[e->edge->rSite
            {
                // define points
                float x1 = e->edge->vertA->x;
                float y1 = e->edge->vertA->y;
                float x2 = e->edge->vertB->x;
                float y2 = e->edge->vertB->y;

                // Add line
                templines.push_back(sf::Vector2f(x1, y1));
                templines.push_back(sf::Vector2f(x2, y2));

                // Parallel line
                float L = std::sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
```

```
                float offset = 2;

                float x1p = x1 + offset * (y2 - y1) / L;
                float y1p = y1 + offset * (x1 - x2) / L;
                float x2p = x2 + offset * (y2 - y1) / L;
                float y2p = y2 + offset * (x1 - x2) / L;
                // Add line
                templines.push_back(sf::Vector2f(x1p, y1p));
                templines.push_back(sf::Vector2f(x2p, y2p));
            }
        }
    }

    sf::VertexArray lines(sf::Lines, templines.size());
    for (int i = 0; i < templines.size(); i++)
    {
        lines[i].position = templines[i];
        lines[i].color = sf::Color::Black;
    }
    return lines;
}
```

.2.4 Map.hpp

```
#pragma once

#include <SFML/Graphics.hpp>
#include "Diagram.h"
struct City
{
    sf::Vector2f pos;
    std::string name;
    bool drawName;
    City(sf::Vector2f pos, std::string name) : pos(pos), name(name) { drawNa
```

```
};

class Map
{
public:
    Map(sf::Texture texture, sf::VertexArray lines);

    void addCity(City city) { cities.push_back(city); }
    void addLandMass(sf::ConvexShape shape);

    void addMountain(sf::Vector2f pos) { mountains.push_back(pos); }
    std::size_t getMountainSize() { mountains.size(); }
    sf::Vector2f getMountain(int i) { return mountains[i]; }

    void addTrees(sf::Vector2f pos) { trees.push_back(pos); }

    void update(float dt);
    void handleEvent(sf::Event e);

    void draw(sf::RenderWindow &window);

    void setRoads(Diagram *diagram);

private:
    void drawMountain(sf::Vector2f pos, sf::RenderWindow &window);
    void drawTrees(sf::Vector2f pos, sf::RenderWindow &window);

    sf::VertexArray roads;

    sf::Texture texture;
    sf::Sprite sprite;
    sf::VertexArray lines;
```

```
    sf::Texture mountainTexture;

    // Font loading
    sf::Font font;
    sf::Text text;
    sf::CircleShape citydot;

    std::vector<City> cities;
    std::vector<sf::Vector2f> mountains;
    std::vector<sf::Vector2f> trees;
};
```

.2.5 Map.cpp

```
#include <World/Map.hpp>
#include <math.h>

Map::Map(sf::Texture texture, sf::VertexArray lines)
{
    this->texture = texture;
    sprite.setTexture(this->texture);
    this->lines = lines;

    text.setColor(sf::Color::Black);
    citydot.setOutlineColor(sf::Color::Black);
    citydot.setFillColor(sf::Color::White);
    citydot.setOutlineThickness(2.0f);
    citydot.setRadius(5);
    citydot.setOrigin(5, 5);

    font.loadFromFile("../res/fonts/ferrum.otf");
    text.setFont(font);

    mountainTexture.loadFromFile("../res/images/mountain.png");
```

```
}

void Map::update(float dt)
{
}

void Map::handleEvent(sf::Event e)
{
    if (e.type == sf::Event::MouseMoved)
    {
        sf::Vector2f pos = sf::Vector2f(e.mouseMove.x, e.mouseMove.y);
        for (int i = 0; i < cities.size(); i++) {
            if(std::sqrt(std::pow(pos.x - cities[i].pos.x, 2) + std::pow(pos
                cities[i].drawName = true;
            } else {
                cities[i].drawName = false;
            }
        }
    }
}

void Map::drawMountain(sf::Vector2f pos, sf::RenderWindow &window) {
    sf::Sprite mountain;
    mountain.setTexture(mountainTexture);
    mountain.setOrigin(mountain.getGlobalBounds().width/2, mountain.getGloba
    mountain.setScale(0.02, 0.02);
    mountain.setPosition(pos);
    window.draw(mountain);
}

void Map::drawTrees(sf::Vector2f pos, sf::RenderWindow &window) {
    sf::CircleShape triangle(2);
    triangle.setPosition(pos);
```

```
    triangle.setOutlineColor(sf::Color::Black);
    triangle.setOutlineThickness(1);
    window.draw(triangle);
}

void Map::setRoads(Diagram *diagram) {
    int road_size = diagram->edges.size();
    roads = sf::VertexArray(sf::Lines, road_size * 2);

    for(int i = 0; i < road_size; i++) {
        auto line = diagram->edges[i];
        if(line->rSite != nullptr) {
            roads[i*2].position = sf::Vector2f(line->lSite->p.x, line->lSite
            roads[i*2].color = sf::Color::Red;
        }

        if(line->rSite != nullptr) {
            roads[i*2+1].position = sf::Vector2f(line->rSite->p.x, line->rSi
            roads[i*2+1].color = sf::Color::Red;
        }
    }
}

void Map::draw(sf::RenderWindow &window)
{
    // render the coloured texture
    // window.draw(sprite);

    window.draw(lines);
    window.draw(roads);

    for (auto t : trees) {
        drawTrees(t, window);
```

```
    }

    for(auto m : mountains) {
        drawMountain(m, window);
    }

    for (auto c : cities)
    {
        text.setPosition(c.pos.x + 10, c.pos.y - 22);
        citydot.setPosition(c.pos);
        text.setString(c.name);
        window.draw(text);
        // if(c.drawName) {window.draw(text);}
        window.draw(citydot);
    }
}
```

.2.6 Game.hpp

```
#pragma once

#include <SFML/Graphics.hpp>
#include "World/Map.hpp"
#include "World/MapGenerator.hpp"

class Game
{
public:
    Game(sf::RenderWindow &window);

    void handleEvent(sf::Event &e);
    void update(float dt);
    void render();
```

```
private:
    void cameraZoom(int mousex, int mousey, float m);

    sf::RenderWindow &window;
    Map *map;
    float zoom;
};
```

.2.7 Game.cpp

```
#include <Scenes/Game.hpp>
#include <stdlib.h>
#include "Point2.h"
#include <iostream>

Game::Game(sf::RenderWindow &window) : window(window)
{
    // Create map

    MapGenerator *mgen = new MapGenerator();
    map = mgen->generate();
    zoom = 1;
    delete mgen;
}

void Game::handleEvent(sf::Event &e)
{
    if (e.type == sf::Event::KeyPressed)
    {
        if (e.key.code == sf::Keyboard::R)
        {
            MapGenerator *mgen = new MapGenerator();
            delete map;
            map = mgen->generate();
```

```
            delete mgen;
        }
    }
    else if (e.type == sf::Event::MouseWheelScrolled)
    {
        if (e.mouseWheelScroll.delta < 0)
        {
            cameraZoom(e.mouseWheelScroll.x, e.mouseWheelScroll.y, 0.1);
        }
        else if (e.mouseWheelScroll.delta > 0)
        {
            cameraZoom(e.mouseWheelScroll.x, e.mouseWheelScroll.y, -0.1);
        }
    }
    map->handleEvent(e);
}

void Game::cameraZoom(int mousex, int mousey, float m)
{
    sf::Vector2i pixel(mousex, mousey);
    const sf::Vector2f before(window.mapPixelToCoords(pixel));
    sf::View view(window.getView());
    ;
    view.zoom(zoom += m);
    window.setView(view);
    const sf::Vector2f after(window.mapPixelToCoords(pixel));
    const sf::Vector2f finalCoord(before - after);
    view.move(finalCoord);
    window.setView(view);
}

void Game::update(float dt)
{
```

```
}

void Game::render()
{
    map->draw(window);
}
```

Main.cpp