Procedural Generation of Terrain with focus on Creating a Realistic World

MComp Computer Science w/ Industrial Placement (Game Engineering)

Supervisor: Lindsay Marshall

May 2021

Word Count: 9406

Lauren Jefferson Pugh (170222998)

Abstract

This dissertation researches popular techniques for procedurally generating terrain, well-known biome-type models, and suitable game engines for generating terrain in order to create an application that allows for the realistic generation of terrain. From the research, I conclude that Perlin noise is the most effective algorithm for generating my terrain procedurally and that Whittaker's biome-types model and the Unity game engine are suitable for this dissertation. I develop an application and perform testing to determine how effective it is at generating realistic terrain. Finally, I look at how successfully my dissertation was and how I could improve it.

Declaration

“I declare that this dissertation represents my own work, except where otherwise stated.”

Acknowledgements

Firstly, I would like to thank my project supervisor Lindsay Marshall for allowing this project to be undertaken, and for his continuous help and advice throughout. I would also like to thank my friends and family for supporting me throughout this dissertation.

Table of Contents

Table of Figures

Chapter 2: Background Research

Chapter 3: Design and Implementation

Chapter 4: Results and Evaluation

Chapter 1: Introduction

1.1 Purpose

As video games grow bigger and bigger, developers have started focussing on adding elements of realism into their games, making them more immersive. One way of implementing realism and immersion in your game is by making use of biome systems. A biome is a collection of plants and animals that have common characteristics for the environment they exist in.[1]

There are many video games that make use of biome systems, however, there is a lack of games that have realistic biomes near each other. For example, in Minecraft™ it is possible for a snowy taiga biome (the coldest biome in the game) to generate next to a desert biome (a dry and warm biome); it would be extremely rare for this to happen in real life.

Using realistic biome structure in video games would greatly benefit the players. It would enable video games to be much more realistic, which helps leads to video games becoming much more immersive. Having improved immersion in video games will mean that players will become more engaged with the game, allowing them to enjoy it more.[2]

1.2 Aim and Objectives

This dissertation aims to develop an application to procedurally generate terrains with a focus on generating realistic biome formations.

The aim is divided into the following objectives:

1. **Identify techniques for procedural terrain generation.**

This objective is to investigate techniques for procedural terrain generation, in order to find the most efficient technique (in terms of processing power and generation time). This objective will be achieved once I have identified the technique that I will be using in the prototype.

2. **Research existing biome-type models.**

For this objective, I will need to research existing biome-type models and analyse which would be most suitable for this project. This objective will be achieved when I have a firm understanding of how biome models work so that I can implement the most suitable model into the prototype.

3. **Identify suitable engines for procedurally generating terrains.**

This objective includes researching many different engines (e.g., Unity) to identify an engine that will include necessary features that will be required for developing the application. This objective will be achieved when I have analysed existing engines and made a choice for which I will program the prototype using, based on my analysis.

4. **Develop a prototype that will generate terrains with biomes based on the previous objectives.**

Using the engine identified from objective 3, integrate the techniques from objective 1 and the chosen biome-type model from objective 2 into an application that can be evaluated to compare processing power and generation time. This objective will be achieved when I have created a prototype that can procedurally generate terrains with biomes in a formation that follows a chosen biome-type model.

5. **Summarise and evaluate the performance of the application.**

This final objective is to assess whether the final prototype of the application was successful, which will also determine if overall my dissertation was successful. This will be achieved by evaluating the generation time and run time performance.

1.3 Work Plan

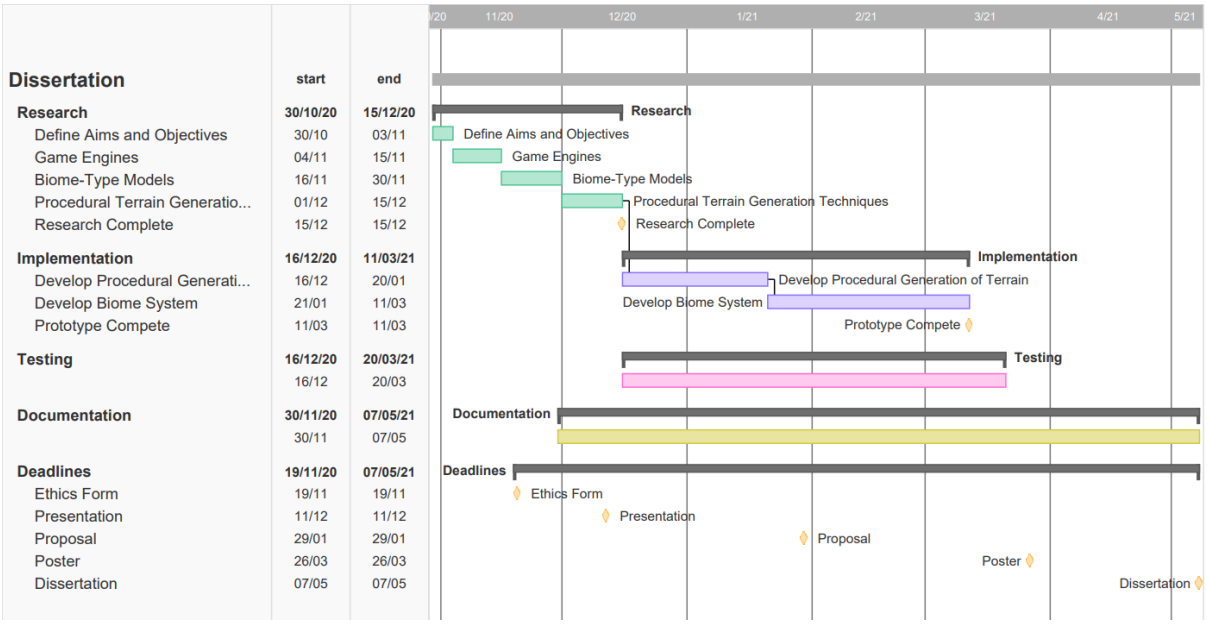

Figure 1: Work Plan

I created a Gantt chart to highlight the key tasks in my project. The chart shows how I have estimated the amount of time each task will take. I originally decided to make use of the waterfall development cycle, for the development of the project. However, I later decided that it would be more suitable to use an agile method for the development. The reason for this decision was because agile development is much more flexible and allows for changing requirements. Also, in the waterfall development cycle, testing is done in the latter phases of the project. This meant that it would be harder to know exactly where bugs were. In agile development, testing takes place concurrently with the software development, meaning that risks are reduced, and bugs are easier to spot.

Before starting any of the implementation, I focused on research; this was to ensure I was confident in my approach to the implementation. I originally decided that I would be researching game engines first. However, when I began my research, I decided that it would make more sense to know what I required my chosen game engine to do. Therefore, I change the research order to procedural generation techniques, biome-types model and then game engines last.

When starting my implementation, I had planned to develop my biome system first and develop the procedural generation of terrain second. Instead of developing them in that order, I swapped them so that the procedural terrain generation was developed first, and the biome system second. This was because I realised that the only thing that could be done for the biome system, before the terrain generation was developed, was to create a table for how the biomes would be decided. So instead, I developed the procedural generation of terrain using height, moisture, and heat maps. Then I used those generated maps to determine the biomes. Testing was done throughout the development of the application.

I had planned to add structures to my generated terrain like rocks, trees, rivers, waterfalls, and caves. However, one of the risks to the project was that I would not have enough time to implement everything, due to other modules which I have work for. I found that I was struggling with time to implement everything that I wanted, and therefore I decided to reduce the complexity of my application so that it did not generate these structures. Although I did not develop a system that generates these structures, my application would still be able to meet all of the objectives that were set. This is because the biomes are still identifiable by texture/colour and they are still generated based on the humidity and temperature of the terrain, meaning that they follow the biome-type model chosen.

1.4 Dissertation Outline

This dissertation is divided into the following chapters.

Chapter 1 – Introduction

I begin by briefly describing how realism is becoming an important factor in modern video games and how having realistic landscapes can help add to the realism. I then introduce my aims and objectives that I would like to achieve in this dissertation.

Chapter 2 – Background Research

Chapter 2 starts by describing a brief explanation of procedural generation and how it has been used in video games in the past. I look at different techniques used for procedurally generating terrain and compare them to determine which would be best for this project. I then move onto describing different ways of dividing land and some of the existing models for this purpose. Finally, I look at game engines that could be used to develop this application, outlining their advantages and disadvantages.

Chapter 3 – Design and Implementation

This chapter starts by explaining the methodology I have chosen to use for development and why. I then cover the tools and technologies that will be used to develop the application. I then outline the steps I took in the development of the application and how any problems that I encountered were tackled.

Chapter 4 – Evaluation

Here I discuss the findings from the dissertation, providing results in the form of graphs and figures of the generated terrain. Findings such as generation time, RAM usage and PGU usage were looked into.

Chapter 5 – Conclusion

This chapter presents a conclusion based on the aims and objectives that I outlined in chapter 1. I discuss whether the aim and objectives set in chapter 1 were met and how I could have further improved this project.

Chapter 2: Background Research

2.1 Research Strategy

The research was carried out using both formal and informal sources.

Formal research:

- Research papers on procedural generation techniques and game engines were found through the use of Google Scholar.
- Game engine documentation was used for discussing the suitability of the game engine for generating terrain.

Informal research:

- Forums used include sites such as reddit.com and quora.com, where posts regarding procedural generation techniques and game engines were discussed, respectively.
- Blog posts were also used for the research of procedural generation and biomes.

2.2 Background Research

2.2.1 Procedural Terrain Generation

What is procedural generation?

Procedural generation is the creation of data by computers. It is usually used to create content for video games or animated movies e.g., landscapes, 3D objects, character designs, animations, and dialogue.[3]

How procedural generation is used in video games

Rogue (1980)

An early example of procedural generation used in a video game was a game from the 1980s called ‘Rogue’, which had randomly generated ASCII maps of monster infested dungeons. It achieved this using an algorithm that divided the map into 6 parts (3x2) and placing a room in each part. It then used a corridor connection algorithm to connect the rooms.[4]

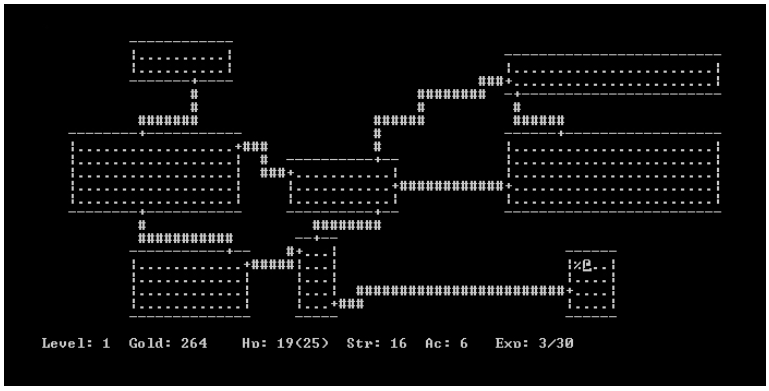

Figure 2: Rogue (1980) Gameplay [5]

The technique used in Rogue was adapted and used in later games like ‘XCOM’, ‘Diablo’ and ‘Civilization’. [6]

Minecraft (2011)

'Minecraft' is a sandbox video game developed by Mojang and was programmed using Java; it is the best-selling video game of all time. Minecraft produces massive worlds that are full of small details such as elaborate cliff faces and waterfalls. Minecraft relies on procedural generation to automatically create environments and objects that are random (yet guided by rules that maintain consistent logic). Minecraft procedurally generates worlds using Perlin noise calculations. [7]

Figure 3: Procedurally generated cliffs and waterfall in Minecraft (2011) [7]

No Man's Sky (2016)

'No Man's Sky' is a video game that presents players with a mind-bogglingly vast galaxy to explore. There are 18 quintillion (18,000,000,000,000,000,000) planets in the game and each planet is unique. It was calculated that it would take players billions of years to explore all of No Man's Sky. It was possible for the development team to create all of the planets because of the use of procedural generation. [8]

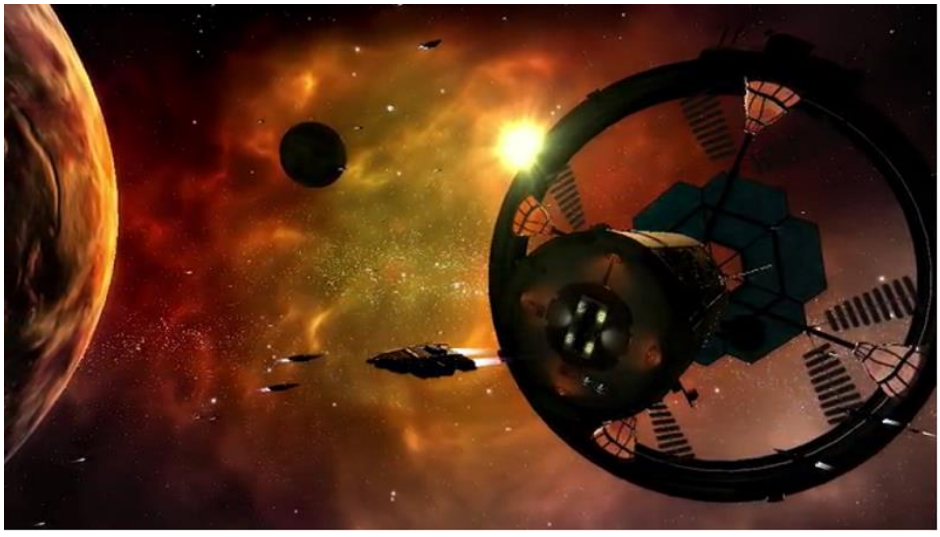

Figure 4: No Man's Sky (2016) Gameplay [8]

Procedural generation techniques

Online and offline procedural generation

There are 2 types of procedural generation modes: online and offline.

Online generation is when the procedural generation is done in real-time, while the game is running. 'Minecraft' uses online generation to generate terrain as the player moves around the world for the first time. Offline generation is when the procedural generation is performed before running the game. 'No Man's Sky' uses offline generation as all players explore the same map, except they start in different areas. [9]

Online and offline generation both have their own benefits and drawbacks.
As online generation is performed in real-time, it gives developers the ability to generate infinite worlds, which can help to add immersion to games. However, online generation is much more taxing on the hardware. This is because the algorithms for online generation are run whenever the player moves to an area that they have not discovered before, and these algorithms can be quite intense and require a lot of computing power. This can cause lower spec computers to struggle to load the world and have very low frames per second.
Offline generation does not have the ability to generate infinite worlds, as the world is generated before the game is run. However, because it generates the world beforehand, the player's computer does not have to use as much computing power. Another drawback is that the loading time before the game can be played will be longer, this is because the algorithm has to generate the entire world beforehand, rather than a small area like in online generation.

This project will not be generating terrain as the program is run, and therefore will be using offline generation.

Diamond-square algorithm

The diamond-square algorithm (DS) is a technique for generating height maps. A height map is an image that can be used to store the height value for each point of a terrain.

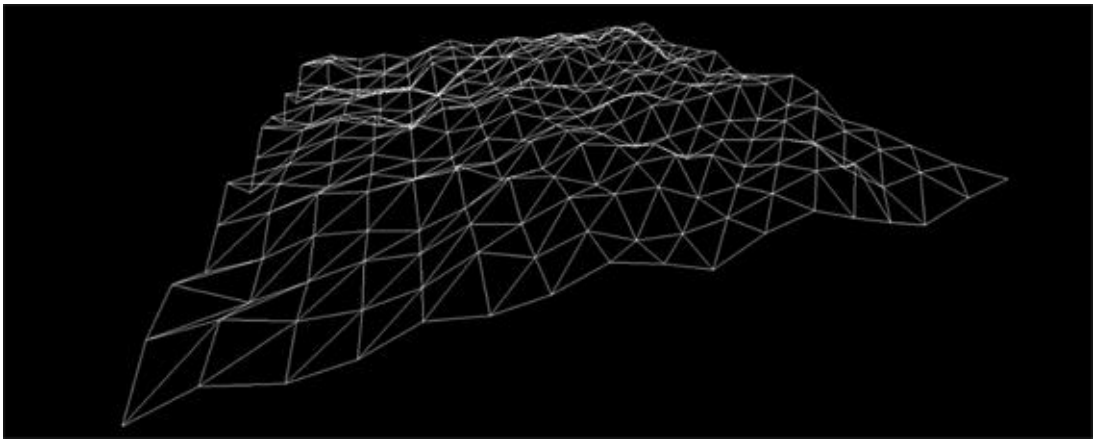

Figure 5: A wireframe generated by Diamond-Square [10]

DS requires a 2D array of size $2^n + 1$. It begins with pre-seeded values in the four corners of the array. It then loops over progressively smaller step sizes, performing a 'diamond step' and then a 'square step' until each value in the array has been set.

The diamond step finds the midpoint of a square and sets it to the average of the 4 corners plus a random value in a range. The square step finds the midpoint of a diamond and pulls in the average of the values of the points forming the 4 corners plus a random value. [10]

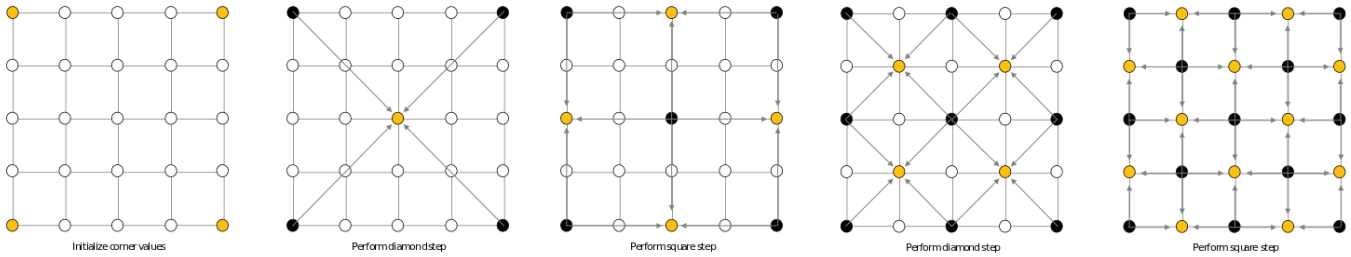

Figure 6: Diamond-Square algorithm overview [10]

Noise generation

Noise is a series of random numbers, typically arranged in a line or a grid. [11]

Terrain and texture generation can be performed utilising algorithms to produce noise. Noise produced by algorithms is useful as it has structure, rather than being completely random. Procedural generation of terrain can be hard because of the unusual shapes and patterns found in nature. When procedurally generating terrain, mimicking the way nature looks is one of the key challenges. [12]

The data produced by a noise function can be rendered directly and/or used to produce a heightmap. For example, in a greyscale height map, the data can be interpreted so that the darker the point, the higher it is. [12]

Value noise

Value noise is, as it suggests, noise based on values. The method consists of the creation of a lattice of points, which are assigned random values. Linear interpolation is then used to return a number based on all the values surrounding the lattice points.

A disadvantage of this method of noise generation is that some areas of the noise may contain little changes and others drastic changes.

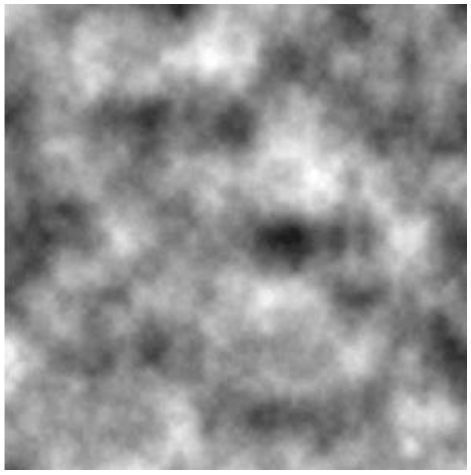

Figure 7: Value Noise rendered as a height map [12]

Perlin noise

Perlin noise is a lattice-based gradient noise type. This means that a pseudo-random gradient is set at regularly spaced points in space and are interpolated to make it smooth between those points. It is a popular algorithm used in procedural generation and can be used to generate textures and terrain.

To generate Perlin noise in one dimension, you associate a pseudo-random gradient for the noise function with each integer coordinate and set the function value at each integer coordinate to zero.[13]

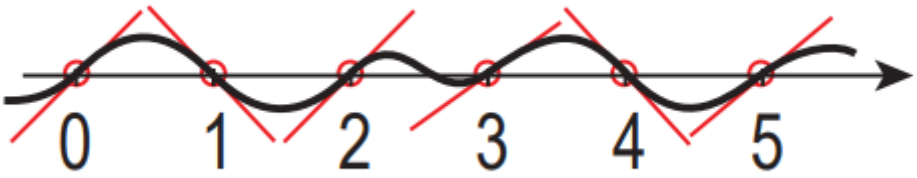

Figure 8: 1D Perlin noise [13]

As Perlin noise is a gradient implementation of noise, it is very useful for generating more natural looking results. Unlike value noise, there are no drastic changes in the values generated.

By using gradients, Perlin noise also reduces the chances of having flat curves; this is because the interpolation is calculated between tangents, rather than using values like in value noise.

Figure 9: 2D Perlin noise [14]

Simplex noise

In 2001, Ken Perlin presented 'simplex noise' as a replacement for his classic noise algorithm 'Perlin noise'. Perlin noise is the most used noise generation, however, it has quite some limitations. Perlin noise does not scale very well into higher dimensions (e.g., 4D, 5D), this is because of its complexity of $O(2^N)$. Simplex noise was designed the overcome that limitation.

The advantages of simplex noise are:

- Lower computational complexity – simplex noise scales to higher dimensions (4D, 5D, etc.) with much less computational cost. The complexity for simplex noise is $O(N^2)$ compared to Perlin noise's complexity of $O(2^N)$.
- Simplex gradient has a well-defined and continuous gradient everywhere that can be computed quite cheaply.

[13]

Worley noise (Cellular noise)

Worley noise is a noise function introduced by Steven Worley in 1996. It can be used to generate procedural textures and comes close to simulating the textures of stone, water, or biological cells.

In Worley noise, a varying number of feature points is spread in space and the distances to these points are calculated from these points to every possible point in space. [15]

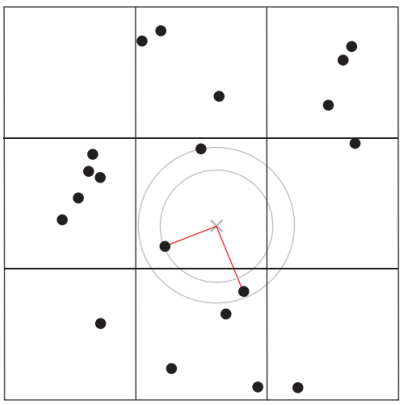

Figure 10: the feature point spread of a typical Worley noise. Sample point marked with x. [15]

Figure 10 shows a space with feature points scattered across the plane. For a position on the plane, the x in the figure, there is always a closest feature point (as well as a second and third closest etc.). The algorithm searches the plane to find them and returns the distance, the relative position, and the ID number of each feature point. [15]

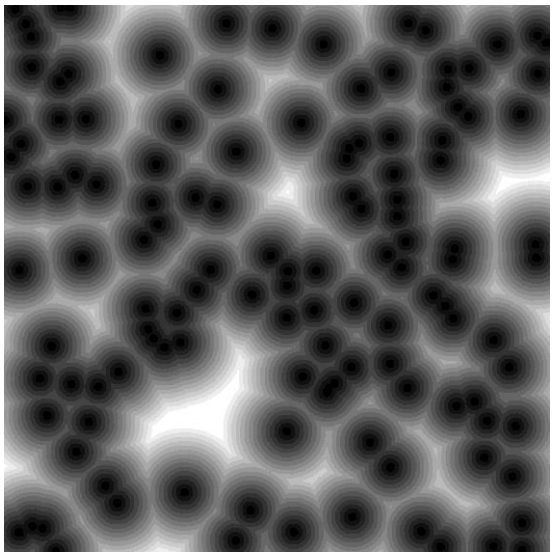

Figure 11: An example of Worley noise [15]

Comparison of noise algorithms

Algorithm	Speed	Quality	Memory Requirements
Diamond-Square Algorithm	Very Fast	Moderate	High
Value Noise	Slow - Fast*	Low - Moderate*	Very Low
Perlin Noise	Moderate	High	Low
Simplex Noise	Moderate**	Very High	Low
Worley Noise	Variable	Unique	Variable

*Depends on what interpolation function is used
**Scales better into the higher dimensions than Perlin Noise

Figure 12: Noise algorithms comparison table [12]

Figure 12 shows a table that compares the speed, quality, and memory requirements of the aforementioned noise algorithms.

The most suitable noise generation algorithm will vary between terrain generation systems. If speed is essential, the Diamond-square algorithm is the best option as it is much faster than the other algorithms tested.
If there is not much memory available, Value noise is a good alternative and is easily customisable due to its simple nature.
Worley noise is much more visually unique and is mainly desirable when producing textures for things like water, rocks, and biological cells. It is not ideal for procedurally generating terrain.
For overall quality, simplex noise is the ideal choice (although Perlin noise will be sufficient in the vast majority of circumstances). In the majority of circumstances, work will be done in 2 or 3 dimensions. In these cases, Perlin noise will be more than sufficient; however, if moving into further dimensions, the simplex noise algorithm will provide advantages. [12]

2.2.2 Biomes

What are biomes?

A biome is a type of environment that is defined by the types of organisms that live there. They are also known as 'life zones'. Dividing land up this way allows us to talk about areas that are similar, however, there are many different opinions on how exactly the land should be divided. [16]

Existing biome-type models

Holdridge life zones

The earliest biome concept was published by Leslie Holdridge in 1947. Holdridge classified climates based on the biological effects of temperature and rainfall on vegetation. The life zones system assumption is that if the climate is known, soil and vegetation can be predicted and mapped. [17]

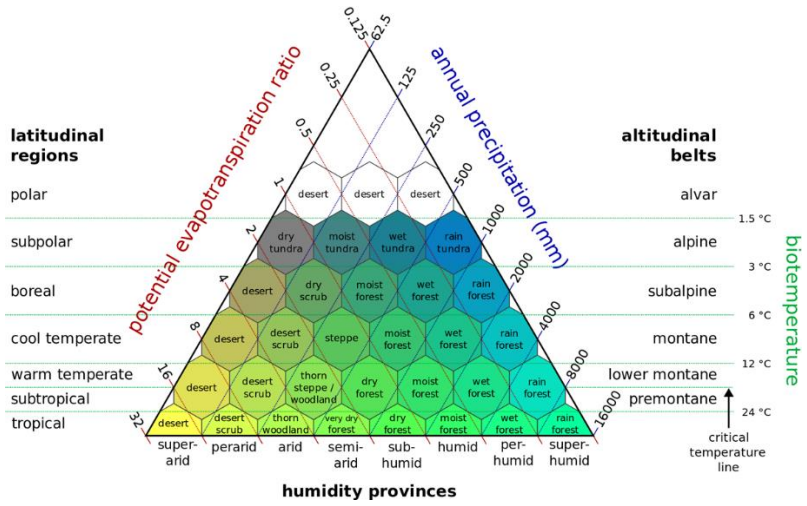

Figure 13: Holdridge life zone classification scheme [17]

Figure 13 shows the classification scheme of the Holdridge life zones model.

The 3 axes show:

- Precipitation (log annual).
- Biotemperature – this is all temperatures above freezing, with all temperatures below freezing adjusted to 0 °C.
- Potential evapotranspiration: mean total annual precipitation– this means the amount of evaporation that would occur if a sufficient water source were available. [17]

If these parameters are known for any given landmass, live zones can be mapped onto it.

Walter's zonobiomes

Heinrich Walter created a biome classification scheme based on temperature and precipitation seasonality. It finds nine major biome types with important climate traits and vegetation types.

Zonobiome	Zone Soil Type	Zone Vegetation	Biome name
I Equatorial: Always moist and lacking temperature seasonality	Equatorial brown clays	Evergreen tropical rain forest	Tropical rain forest
II Tropical: Summer rainy season and "winter" dry season	Red clays or red earths	Seasonal forest, scrub, or savanna	Tropical seasonal forest/savanna
III Subtropical (hot deserts): Highly seasonal, arid climate	Sierozems (brownish-grey above, lighter below)	Desert vegetation with considerable exposed surface	Subtropical desert
IV Mediterranean: Winter rainy season and summer drought	Mediterranean brown earths	sclerophyllous (drought-adapted) frost-sensitive shrublands and woodlands	Woodland/shrubland
V Warm Temperate: Occasional frost, often with summer rainfall maximum	Yellow or red forest soils, slightly podsolic soils	Temperate evergreen forest, somewhat frost-sensitive	Temperate rain forest
VI Nemoral: Moderate climate with winter freezing	Forest brown earths and grey forest soils	Frost-resistant, decidous, temperate forest	Temperate seasonal forest
VII Continental (cold deserts): Arid, with warm or hot summers and cold winters	Chernozems (fertile, humus-rich) to sierozems	Grasslands and temperate deserts	Temperate grassland/desert
VIII Boreal: Cold temperate with cool summers and long winters	Podsols	Evergreen, frost-hardy needle-leaved forest (taiga)	Boreal forest
IX Polar: Very short, cool summers and long, very cold winters	Tundra humus soils with solifluction (permafrost soils)	Low, evergreen vegetation, without trees, growing over permanently frozen soils	Tundra

Figure 14: Walter's Zonobiomes Scheme [17]

Whittaker biome-types

Robert Whittaker based his biome classifications on temperature and precipitation. He matched vegetation type to regional climate, to create a figure which all biomes fall into. His scheme can be seen as a simplification of Holdridge's life zones.

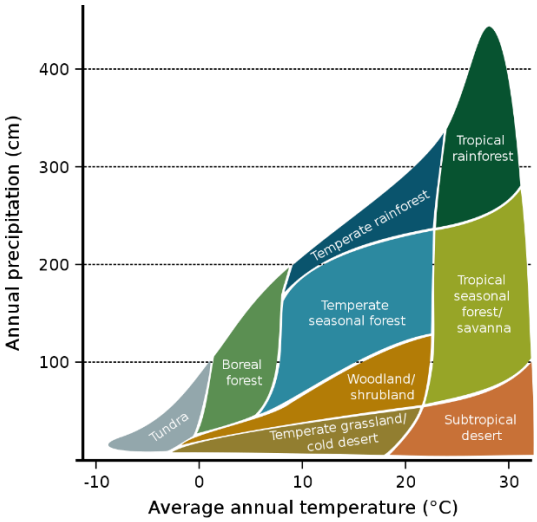

Figure 15: Whittaker biome-types model [17]

Whittaker defines a biome as a

"Grouping of terrestrial ecosystems on a given continent that is similar in vegetation structure, physiognomy, environmental features, and animal community characteristics." [17]

Biome-type model comparisons

All of the biome-type models have their own advantages and disadvantages for different situations; however, I will be comparing them based on how suitable I think they are for this project.

Holdridge's life zones is a very complex and in-depth model, however, I think that the duplication of biomes in the model could increase the complexity of the algorithm which will generate the terrain. It would also be extremely difficult to generate the values for the potential evapotranspiration of the biomes.

The Walter's zonobiomes and Whittaker's biome-types models are very similar. Both models have the same number of biomes with the same names. Both models, also, base the grouping of their biomes off of temperature and precipitation values. The advantage of Walter's zonobiomes model is the description of the soil and vegetation type of each biome; this would be useful for the textures used with my generated terrain, as I will be able to find textures that match the descriptions. The advantage of Whittaker's biome-types model is the graphical format of the model. This means that I am able to clearly see what biomes are able to be generated near each other, which would help me to the algorithm for the application.

2.2.3 Suitable Game Engines for Procedurally Generating Terrain

What is a game engine?

> *"Game engine refers to a set of tools that help the developers to create games a lot faster and easier using commands that used to require a lot of coding through plug-in or asset provider of the game engine"* [18]

One of the biggest advantages of using game engines is code recycling. In most graphical works, underlying code requires a large amount of the developer's time. With game engines, developers can reuse existing code and modify it so that they do not have to "re-invent the wheel". This saves time, lowers costs and allows the product to reach the market sooner. [19]

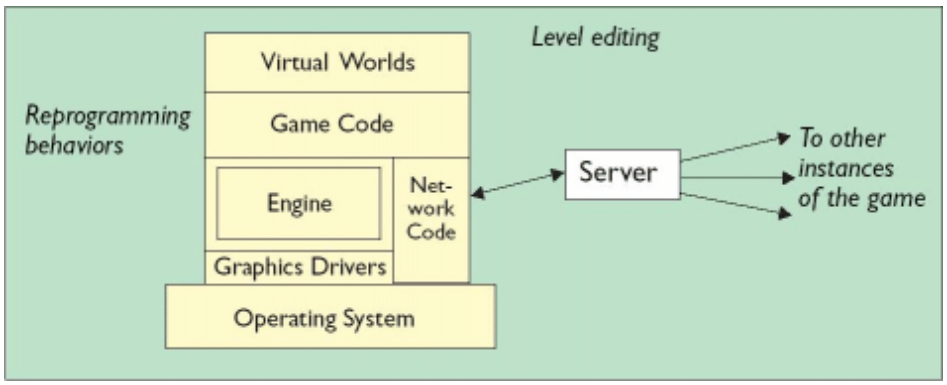

Figure 16: Modular game engine structure [18]

Leading game engines

Unreal Engine

Unreal Engine is a game engine that was developed by Epic Games in 1998. It is updated frequently, and the latest engine is Unreal Engine 4. Unreal Engine is well documented and easy to use. The programming language which it uses is C++. Not only is Unreal Engine used in industry for game development, but it is also used in industries for other purposes; for example, it is used in the automotive industry.[20]

It is capable of creating massive terrain-based worlds using powerful terrain editing tools. The Landscape tool allows the user to create terrain pieces that are optimised and can be used across many different devices.[21]

Examples of games developed with Unreal Engine are Fortnite and Mortal Kombat.

Unity

Unity is a game engine developed by Unity Technologies in 2005. Unity is a powerful game engine that supports over 25 platforms, it can render real-world graphics without using much processing power. It is also very user-friendly to both beginners and experts, uses C# for its scripts and is well-documented. Like Unreal Engine, Unity is also used outside of game development. It is often used in architecture, film, automotive and engineering. [22]

The Unity editor includes a built-in set of Terrain features that allow you to add landscapes to your game. You can adjust the appearance of the landscape and see the changes in real-time. At runtime, Unity optimises built-in Terrain rendering for efficiency. [23]

Examples of games developed with Unity are Fall Guys and Genshin Impact.

CryEngine

CryEngine is a game engine developed by Crytek in 2002. It is a cross-platform engine that can be used to create photo-realistic games. The biggest advantage of CryEngine is its Global Illumination technology. Global Illumination allows in-game light to behave extremely close to how it does in the real world. CryEngine also has landscape tools that allow the user to sculpt, paint and import terrain from heightmaps. However, CryEngine is lacking in good documentation and has performance and scalability issues. [24]

Examples of games developed with CryEngine are Ryse: Son of Rome and the Crysis series.

2.3 Discussion

When comparing the different procedural generation methods for terrain, it is clear that simplex noise or Perlin noise would be the most suitable for this project. As I will be using 2D noise, Perlin noise is more than sufficient and therefore will be used, in this project.

The biome model chosen for this project, after research, is the Whittaker biome-types model. The reason for this is that, compared to the Holdridge life zones model, there are fewer biomes to represent and therefore it will be easier to evaluate.

The game engine chosen for this project is Unity. There is a great amount of support and documentation for the Unity game engine. Its Terrain tools will be extremely useful for this project and, because of the optimisation of built-in terrain, it will be more efficient than Unreal Engine. Another reason for this decision is that many industries use Unity, and this will allow me to develop more skills and experience in Unity for the future.

Chapter 3: Design and Implementation

3.1 Planning

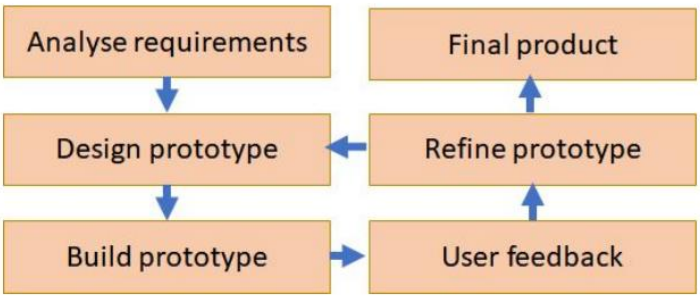

Figure 17: Agile Methodology [25]

I adopted an agile method for this project, as it allowed constant testing to ensure that it met the requirements and was high quality.

In agile development, the problem is broken down into sections that are developed in parallel. Different sections of the project can be at different stages of development. Agile methodologies aim to improve the flexibility of the software and adapt to changes in user requirements faster. The code developed is also of higher quality because it gets refined and tested every cycle, as shown in figure 17. [25]

3.2 Application Design

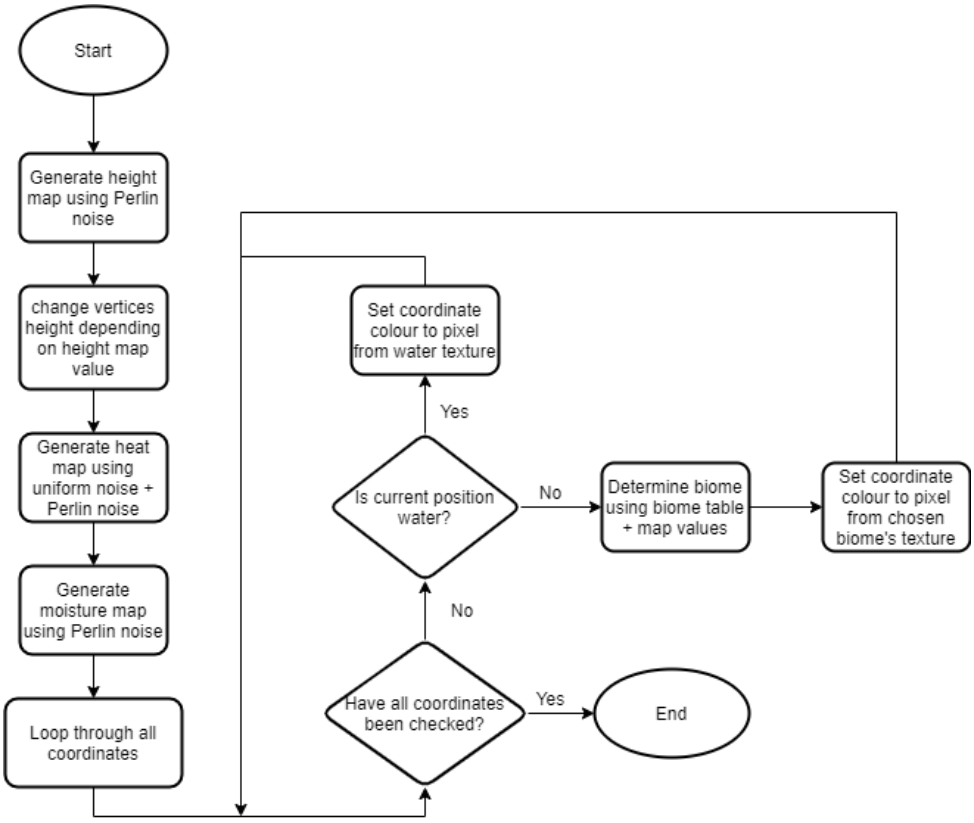

Figure 18: Flowchart to show step-by-step process of the application

I created a flowchart before starting development to show the application process step-by-step, as shown in Figure 18. The main reason for creating this was to ensure that I fully understood how I wanted the application to work before the programming started. This allowed me to clear up any misunderstandings and fix any logical errors before starting to code the application.

I followed this flowchart as much as possible. However, during development, I found that my initial way of adding textures to the biomes resulted in pixelated textures. Therefore, instead of setting the coordinate's pixel to a sample colour from the texture, I changed my method and decided to use a Unity Shader Graph with alpha maps to add textures to the biomes.

3.3 Tools and Technologies

3.3.1 Unity

As discussed in Chapter 2, Unity is a powerful game engine that can be used to create many types of applications. It is very user-friendly and well documented. It will be used for the development of this project because of its built-in terrain tools and optimisation of built-in terrain. The programming language used in Unity is C#.

3.3.2 C#

C# is a high-level, object-oriented, component-orientated programming language. It is an extension of the C programming language and was originally titled COOL, an acronym that stood for "C-like Object-Oriented Language". [26]

Advantages of C# include:

- Object-oriented – as C# is a pure object-oriented language, the user is able to create modular applications and reusable code, saving time and effort.

- Automatic garbage collection – C# has a very efficient system to erase and remove all garbage present on the system, with a strong memory backup to help prevent memory leaks.

[27]

3.3.3 Visual Studio

As it is very often used with Unity projects, Visual Studio is the IDE that I used for the development of this project. Visual Studio allows the user to inspect their code at run-time with the use of breakpoints. It also has the error highlighting feature, which shows syntax errors in the users' code before it is run.

3.4 Application Specification

Before implementation can be started, the application requirements need to be outlined.

Requirement 1 – Controllable camera

- The user must be able to move around the generated word, without issues, using a flying camera.
- The camera must have full keyboard and mouse support.

Requirement 2 – Add biomes

- The user must be able to select and add biomes that they want to be generated in the world.

Requirement 3 – Adjust appearance of biomes

- The user must be able to adjust the height of the biomes.
- The user must be able to change the textures of the biomes.

Requirement 4 – Follow the biome-type model

- The application must follow the Whittaker biome-types model when generating biomes in the world.

Fully meeting requirements 2 to 4 would mean that I have completed the aim of my dissertation which is to develop an application to procedurally generate terrains with a focus on generating realistic biome formations. The reason I have included requirement 1 is to allow for more user friendliness within the application and to allow the user to properly inspect the generated terrain.

3.5 Development

3.5.1 Generating noise

The first step to procedurally generating the terrain was to generate a noise map. As stated previously, Perlin noise is used to generate noise maps.

```
public class NoiseMapGeneration : MonoBehaviour
{

    public float[,] GenerateNoiseMap(int mapDepth, int mapWidth, float scale)
    {
        // create an empty noise map with the mapDepth and mapWidth coordinates
        float[,] noiseMap = new float[mapDepth, mapWidth];

        for (int zIndex = 0; zIndex < mapDepth; zIndex++)
        {
            for (int xIndex = 0; xIndex < mapWidth; xIndex++)
            {
                // calculate sample indices based on the coordinates and the scale
                float sampleX = xIndex / scale;
                float sampleZ = zIndex / scale;

                // generate noise value using PerlinNoise
                float noise = Mathf.PerlinNoise(sampleX, sampleZ);

                noiseMap[zIndex, xIndex] = noise;
            }
        }

        return noiseMap;
    }
}
```

Figure 19: Initial noise map generation code

Figure 19 shows the code written for the 'NoiseMapGeneration' class. This class is used for generating values for the height, moisture, and temperature of each vertex. The function works by creating a noise map using the parameters passed to it, looping through each vertex, using the built-in Perlin noise function provided by Unity to generate a value between 0 and 1 for that vertex, and then returning the noise map.

3.5.2 Generating the world

3.5.2.1 Height Map

To get started with the world generation, I first created a 'Plane' object called 'Level Tile'. The values from the noise map generated adjust the vertices of the Level Tile to either be a different height or colour. To test the noise map was generated correctly, I changed the colours of the Level Tile vertices based on the value provided from the noise map.

```
void Start()
{
    GenerateTile();
}

void GenerateTile()
{
    // calculate tile depth and width based on the mesh vertices
    Vector3[] meshVertices = this.meshFilter.mesh.vertices;
    int tileDepth = (int)Mathf.Sqrt(meshVertices.Length);
    int tileWidth = tileDepth;

    // calculate the offsets based on the tile position
    float[,] heightMap = this.noiseMapGeneration.GenerateNoiseMap(tileDepth, tileWidth, this.mapScale);

    // generate a heightMap using noise
    Texture2D tileTexture = BuildTexture(heightMap);
    this.tileRenderer.material.mainTexture = tileTexture;
}
```

Figure 20: Tile generation code

Figure 20 shows the function used to generate the tile. I generate a height map using the noise generation function. This height map is used to colour the Level Tile vertices based on the values generated.

```
private Texture2D BuildTexture(float[,] heightMap)
{
    int tileDepth = heightMap.GetLength(0);
    int tileWidth = heightMap.GetLength(1);

    Color[] colorMap = new Color[tileDepth * tileWidth];
    for (int zIndex = 0; zIndex < tileDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < tileWidth; xIndex++)
        {
            // transform the 2D map index is an Array index
            int colorIndex = zIndex * tileWidth + xIndex;
            float height = heightMap[zIndex, xIndex];
            // choose a terrain type according to the height value
            TerrainType terrainType = ChooseTerrainType(height);
            // assign the colour according to the terrain type
            colorMap[colorIndex] = terrainType.color;
        }
    }

    // create a new texture and set its pixel colours
    Texture2D tileTexture = new Texture2D(tileWidth, tileDepth);
    tileTexture.wrapMode = TextureWrapMode.Clamp;
    tileTexture.SetPixels(colorMap);
    tileTexture.Apply();

    return tileTexture;
}
```

Figure 21: Colouring the vertices

Figure 21 shows the code that tested if the noise map was generated correctly. It works by checking the height of a vertex and setting the pixel colour based on the value.

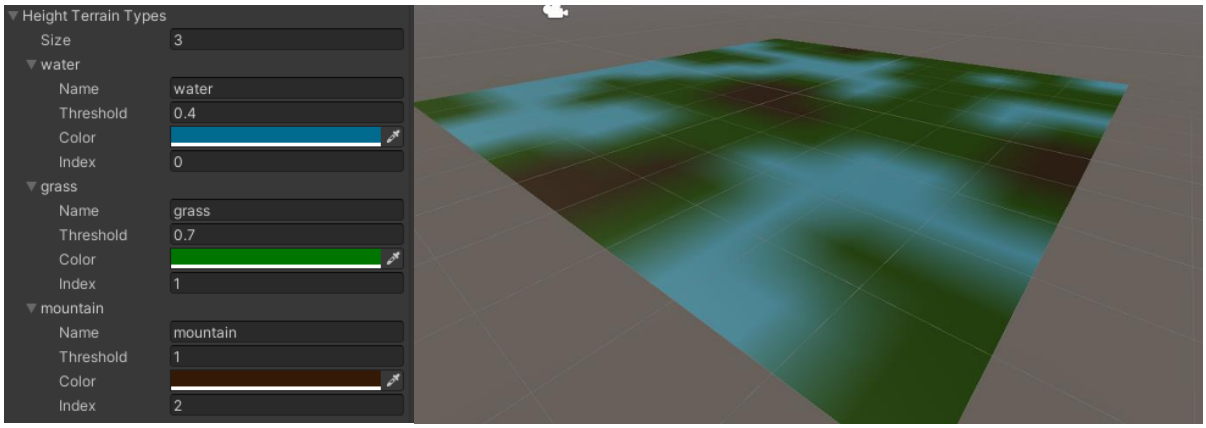

Figure 22: height parameters (left) | generation result (right)

Figure 22 shows the values I set for the test (on the left) and the generated world (on the right). From this test, I concluded that the noise maps were being generated correctly. Next, I was able to move on to adjusting the height of the Level Tile based on the noise map values.

```
private void UpdateMeshVertices(float[,] heightMap)
{
    int tileDepth = heightMap.GetLength(0);
    int tileWidth = heightMap.GetLength(1);

    Vector3[] meshVertices = this.meshFilter.mesh.vertices;

    // iterate through all the heightMap coordinates, updating the vertex index
    int vertexIndex = 0;
    for (int zIndex = 0; zIndex < tileDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < tileWidth; xIndex++)
        {
            float height = heightMap[zIndex, xIndex];

            Vector3 vertex = meshVertices[vertexIndex];
            // change the vertex Y coordinate, proportional to the height value
            meshVertices[vertexIndex] = new Vector3(vertex.x, height * this.heightMultiplier, vertex.z);

            vertexIndex++;
        }
    }

    // update the vertices in the mesh and update its properties
    this.meshFilter.mesh.vertices = meshVertices;
    this.meshFilter.mesh.RecalculateBounds();
    this.meshFilter.mesh.RecalculateNormals();
    // update the mesh collider
    this.meshCollider.sharedMesh = this.meshFilter.mesh;
}
```

Figure 23: UpdateMeshVertices function

Figure 23 shows the function that was used to change the height values of each vertex on the Level Tile. The function iterates through each coordinate on the Level Tile and changes the y coordinate to be the value of the noise value (which then gets multiplied by a height multiplier so I can change how drastic the height changes are).

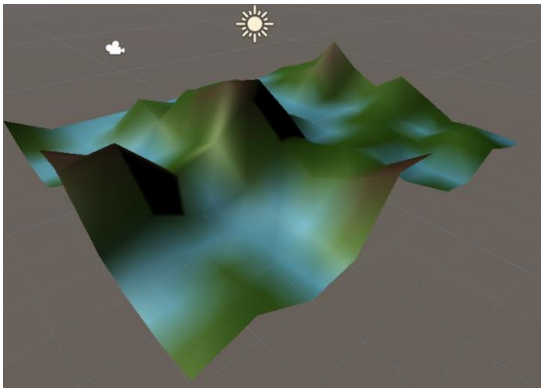

Figure 24: First attempt at changing height values

Figure 24 shows the result of using the UpdateMeshVertices function with a height multiplier of 3. The function succeeded in changing the mesh height values for each vertex, however, the water looked strange because the height changes were getting applied to it too.

Figure 25: Height curve

To fix this issue, I used a height curve, as shown in Figure 25, to make any height value below 0.4 return 0. This made it so that anywhere with water was flat.

```
private void UpdateMeshVertices(float[,] heightMap) {
    int tileDepth = heightMap.GetLength (0);
    int tileWidth = heightMap.GetLength (1);

    Vector3[] meshVertices = this.meshFilter.mesh.vertices;

    // iterate through all the heightMap coordinates, updating the vertex index
    int vertexIndex = 0;
    for (int zIndex = 0; zIndex < tileDepth; zIndex++) {
        for (int xIndex = 0; xIndex < tileWidth; xIndex++) {
            float height = heightMap [zIndex, xIndex];

            Vector3 vertex = meshVertices [vertexIndex];
            // change the vertex Y coordinate, proportional to the height value. The height value is evaluated by the heightCurve function, in order to correct it.
            meshVertices[vertexIndex] = new Vector3(vertex.x, this.heightCurve.Evaluate(height) * this.heightMultiplier, vertex.z);

            vertexIndex++;
        }
    }

    // update the vertices in the mesh and update its properties
    this.meshFilter.mesh.vertices = meshVertices;
    this.meshFilter.mesh.RecalculateBounds ();
    this.meshFilter.mesh.RecalculateNormals ();
    // update the mesh collider
    this.meshCollider.sharedMesh = this.meshFilter.mesh;
}
```

Figure 26: updated UpdateMeshVertices function

Figure 26 shows the UpdateMeshVertices function which now uses the height curve to evaluate the height value and correct it so that it returns 0 where there is water. Figure 27 shows the result after this change.

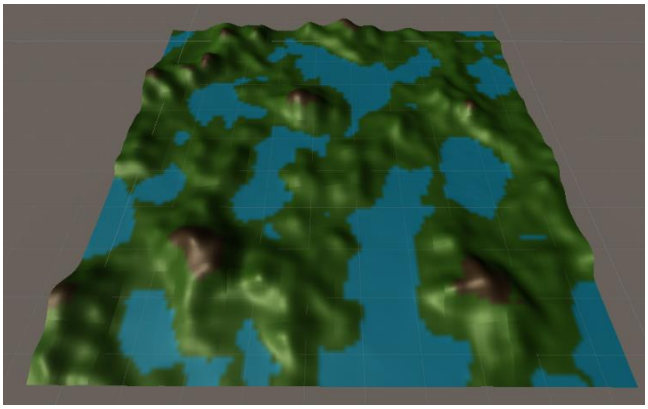

Figure 27: generation using height curve and multiple tiles

Figure 27 also shows multiple tiles being used. This was done by using the Perlin noise function with the same values as the pixels around the border of the tile. Offset parameters were also added to the function which generates the noise map, these offsets correspond to the Level Tile position so that the height is continuous across the tiles.

```
void GenerateTile()
{
    // calculate tile depth and width based on the mesh vertices
    Vector3[] meshVertices = this.meshFilter.mesh.vertices;
    int tileDepth = (int)Mathf.Sqrt(meshVertices.Length);
    int tileWidth = tileDepth;

    // calculate the offsets based on the tile position
    float offsetX = -this.gameObject.transform.position.x;
    float offsetZ = -this.gameObject.transform.position.z;

    // generate a heightMap using noise
    float[,] heightMap = this.noiseMapGeneration.GenerateNoiseMap(tileDepth,
                         tileWidth, this.mapScale, offsetX, offsetZ, waves);

    // build a Texture2D from the height map
    Texture2D tileTexture = BuildTexture(heightMap);
    this.tileRenderer.material.mainTexture = tileTexture;

    // update the tile mesh vertices according to the height map
    UpdateMeshVertices(heightMap);
}
```

```
ublic float[,] GenerateNoiseMap(int mapDepth, int mapWidth, float scale, float offsetX, float offsetZ)

    // create an empty noise map with the mapDepth and mapWidth coordinates
    float[,] noiseMap = new float[mapDepth, mapWidth];

    for (int zIndex = 0; zIndex < mapDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < mapWidth; xIndex++)
        {
            // calculate sample indices based on the coordinates, the scale and the offset
            float sampleX = (xIndex + offsetX) / scale;
            float sampleZ = (zIndex + offsetZ) / scale;

            // generate noise value using PerlinNoise
            float noise = Mathf.PerlinNoise(sampleX, sampleZ);

            noiseMap[zIndex, zIndex] = noise;
        }
    }

    return noiseMap;
```

Figure 28: Offset update

When the world is generated, it creates Level Tiles by iterating through each tile coordinate. For each tile, it calculates its position and then makes a copy of the Level Tile prefab.

3.5.2.2 Heat Map

The next step was to generate the heat map. The heat map was generated using a uniform noise map, where regions close to the centre are hotter. The reason for this was to mimic reality.

```
public float[,] GenerateUniformNoiseMap(int mapDepth, int mapWidth, float centerVertexZ, float maxDistanceZ, float offsetZ) {
    // create an empty noise map with the mapDepth and mapWidth coordinates
    float[,] noiseMap = new float[mapDepth, mapWidth];

    for (int zIndex = 0; zIndex < mapDepth; zIndex++) {
        // calculate the sampleZ by summing the index and the offset
        float sampleZ = zIndex + offsetZ;
        // calculate the noise proportional to the distance of the sample to the center of the level
        float noise = Mathf.Abs (sampleZ - centerVertexZ) / maxDistanceZ;

        // apply the noise for all points with this Z coordinate
        for (int xIndex = 0; xIndex < mapWidth; xIndex++) {
            noiseMap [mapDepth - zIndex - 1, xIndex] = noise;
        }
    }

    return noiseMap;
}
```

Figure 29: Generating a uniform noise map

Figure 29 shows the function used for generating a uniform noise map. This method works by receiving the map dimensions, the maximum distance to the centre, and the z-coordinate of the centre. The noise is generated proportional to the distance to the centre from each coordinate. To randomise the heat map, it gets mixed with a Perlin noise heat map. This is done by multiplying each coordinate value in the uniform heat map with the same coordinate value in the Perlin noise heat map. Figure 30 shows the generated heat map, using the same BuildTexture method as the height map.

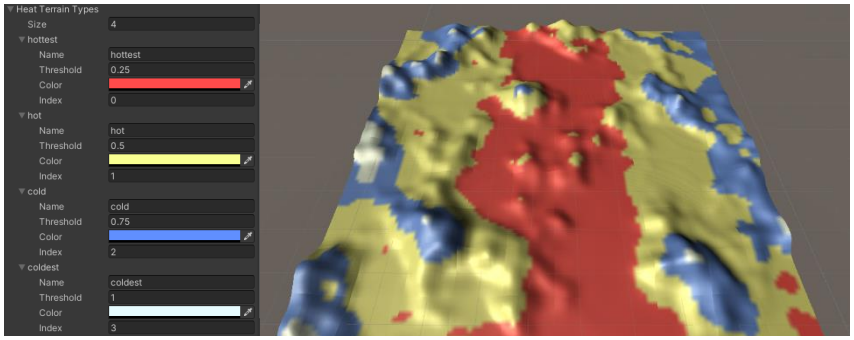

Figure 30: heat parameters (left) | generation result (right)

3.5.2.3 Moisture Map

The final noise map that needed to be generated was the moisture map, this was generated using Perlin noise. This was done in the same way as heat and height noise map generation, and also uses the BuildTexture function shown in Figure 21. Figure 31 shows the moisture map generation with the values used.

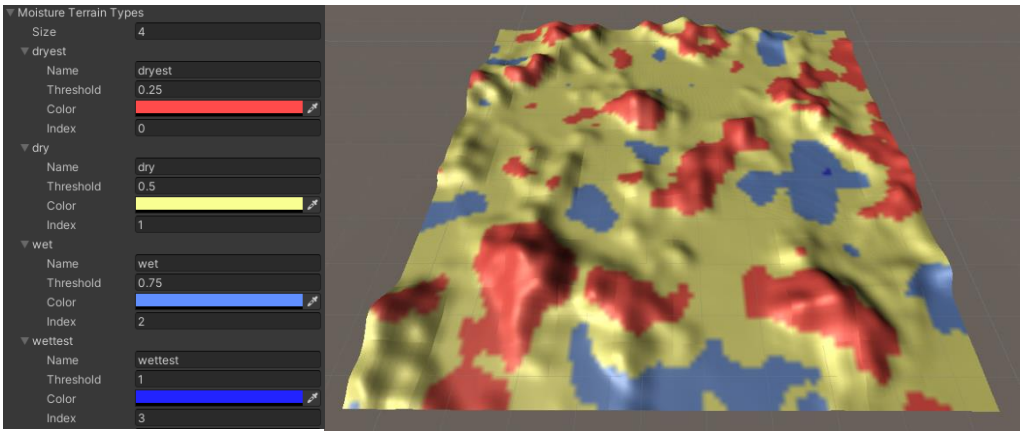

Figure 31: Moisture parameters (left) | generated result (right)

3.5.2.4 Generating Biomes

The final thing to do for generating the world was to generate the biomes. As stated before, Whittaker's biome-types model is used for this project, shown in Figure 15. Biomes are generated using the values from the heat and moisture maps that were generated. I used a table with 16 entries to determine what biome was be generated, as shown in Figure 32. This table was added as a 2D array.

	Hottest	Hot	Cold	Coldest
Driest	Desert	Grassland	Tundra	Tundra
Dry	Savanna	Shrubland	Boreal forest	Tundra
Wet	Savanna	Seasonal forest	Boreal forest	Tundra
Wettest	Tropical rainforest	Temperate rainforest	Boreal forest	Tundra

Figure 32: Biome table

Instead of using the same BuildTexture function as before, I wrote a new one for building the biome texture. This is because I would be using textures once I was sure that it was generating biomes correctly, to test that, I used colour.

To generate the biome texture, I needed to iterate through every tile coordinate and for each check: if it was a water region, the moisture value, and the heat value. A water region is determined by the height value of the vertex, all water regions have a height value of 0. If it was a water region then I skipped the biome check and set the pixel to the water colour. If it was not a water pixel then I used the 2D array with the heat and moisture values to determine the biome and set the pixel to the colour of the biome. Figure 33 shows the function for building the biome texture.

```
private Texture2D BuildBiomeTexture(TerrainType[,] heightTerrainTypes, TerrainType[,] heatTerrainTypes, TerrainType[,] moistureTerrainTypes)
{
    int tileDepth = heatTerrainTypes.GetLength(0);
    int tileWidth = heatTerrainTypes.GetLength(1);

    Color[] colorMap = new Color[tileDepth * tileWidth];
    for (int zIndex = 0; zIndex < tileDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < tileWidth; xIndex++)
        {
            int colorIndex = zIndex * tileWidth + xIndex;

            TerrainType heightTerrainType = heightTerrainTypes[zIndex, xIndex];
            // check if the current coordinate is a water region
            if (heightTerrainType.name != "water")
            {
                // if a coordinate is not water, its biome will be defined by the heat and moisture values
                TerrainType heatTerrainType = heatTerrainTypes[zIndex, xIndex];
                TerrainType moistureTerrainType = moistureTerrainTypes[zIndex, xIndex];

                // terrain type index is used to access the biomes table
                Biome biome = this.biomes[moistureTerrainType.index].biomes[heatTerrainType.index];
                // assign the colour according to the biome
                colorMap[colorIndex] = biome.color;
            }
            else
            {
                // set water colour
                colorMap[colorIndex] = this.waterColor;
            }
        }
    }

    // create a new texture and set its pixel colors
    Texture2D tileTexture = new Texture2D(tileWidth, tileDepth);
    tileTexture.filterMode = FilterMode.Point;
    tileTexture.wrapMode = TextureWrapMode.Clamp;
    tileTexture.SetPixels(colorMap);
    tileTexture.Apply();

    return tileTexture;
}
```

Figure 33: Function to build the biome texture

To make sure the biome generation worked correctly, I tested with 4 biomes. Figure 34 shows the generated result.

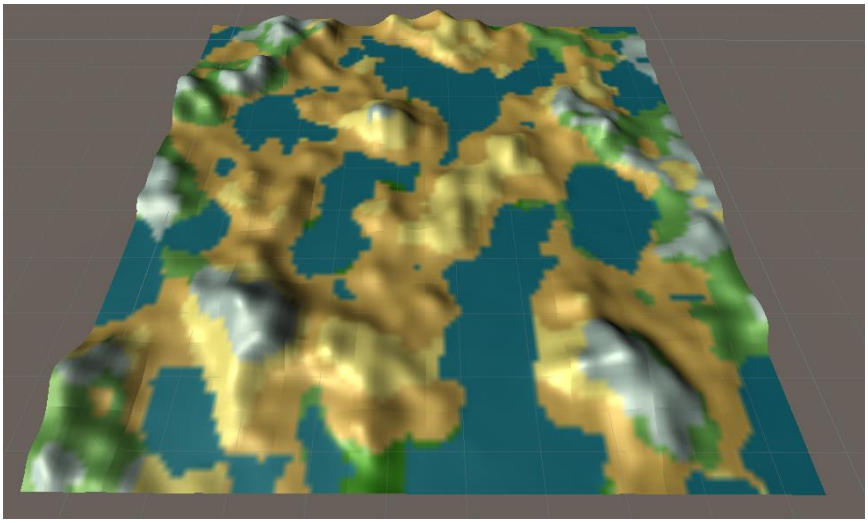

Figure 34: Generated biomes

3.5.3 Adding Textures

Now that the terrain generation worked correctly, the next step was to add textures to the biomes. I collected various textures that I thought would be suitable for the biomes from https://3djungle.net.

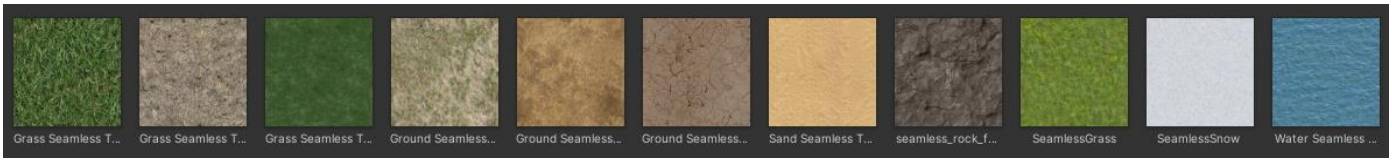

Figure 35: Textures [28]

The textures shown in Figure 35 are all seamless high-quality textures, therefore they should look good when applied to the biomes.

```
private Texture2D[] BuildBiomeTexture(float[,] heightMap, TerrainType[] terrainTypes, TerrainType[,] chosenTerrainTypes)
{
    int tileDepth = heightMap.GetLength(0);
    int tileWidth = heightMap.GetLength(1);
    int size = 5;

    float[,,] alphaMaps = new float[size, tileDepth, tileWidth];
    for (int zIndex = 0; zIndex < tileDepth; zIndex++)
    {
        for (int xIndex = 0; xIndex < tileWidth; xIndex++)
        {
            float height = heightMap[zIndex, xIndex];
            TerrainType terrainType = ChooseTerrainType(height, terrainTypes);
            alphaMaps[terrainType.index, xIndex, zIndex] = 1;

        }
    }

    Normalise(alphaMaps, heightMap, terrainTypes, chosenTerrainTypes);
    Texture2D[] textureList= new Texture2D[alphaMaps.Length];

    for (int textureIndex=0; textureIndex < alphaMaps.Length; textureIndex++) {
        Color[] colorMap = new Color[tileDepth * tileWidth];
        for (int zIndex = 0; zIndex < tileDepth; zIndex++)
        {
            for (int xIndex = 0; xIndex < tileWidth; xIndex++)
            {

                colorMap[textureIndex] = new Color(0,0,0,alphaMaps[textureIndex,xIndex, zIndex]);

            }
        }
        Texture2D tileTexture = new Texture2D(tileWidth, tileDepth);
        tileTexture.wrapMode = TextureWrapMode.Clamp;
        tileTexture.SetPixels(colorMap);
        tileTexture.Apply();
        textureList[textureIndex] = tileTexture;
    }
```

Figure 36: Code to add textures

Figure 36 shows my first attempt at adding textures to the biomes. I had an array that contained all of the textures to be used for the biomes, I then looped through each coordinate, determined the biome, and set the pixel to be a colour that is sampled from the texture for that biome. This however was not successful. It did add texture to the terrain in the correct places, however, the texture on the terrain looked extremely pixelated; as shown in Figure 37.

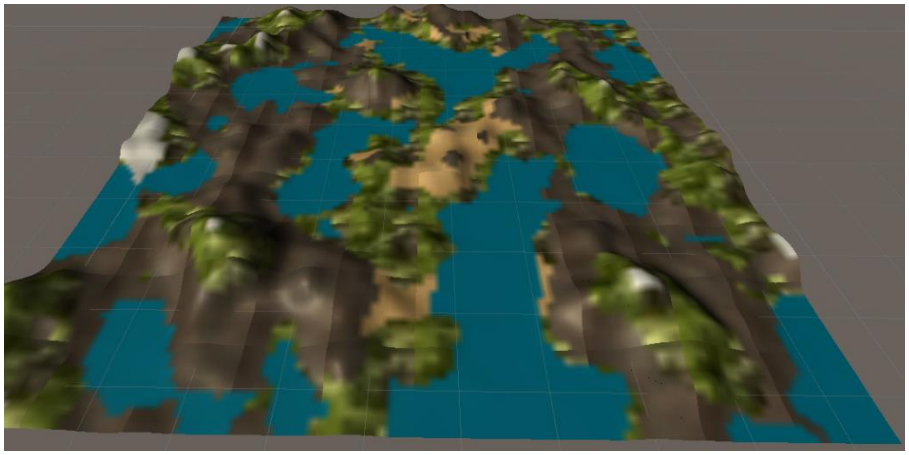

Figure 37: Pixelated texture on terrain

The reason for this is because the size of this world is not the same as the textures being used. This meant that when I took coordinates from the world and used those coordinates in the textures to grab a pixel from the same position, it was as if it was extremely zoomed-in (because the textures were bigger than the world). Instead of pursuing this method of adding texture, I decided it would be much more suitable to use a shader.

3.5.3.1 Shader Graphs

Shader Graph lets you easily author shaders by building them visually and see the results in real-time. You create and connect nodes in a network graph instead of having to write code. [29]
A shader is a program specifically made to run on a GPU that allows special effects to be added to graphical objects. Unity3D also uses them for post-processing. [30]

Using shaders will allow me to add textures with the use of alpha mapping. Alpha mapping is a technique involving the use of texture mapping to designate the amount of transparency/translucency of areas in a certain object. I have decided to use the Unity Shader Graph to write this shader, as I am not experienced in writing shaders and Shader Graphs are easy to understand for beginners.

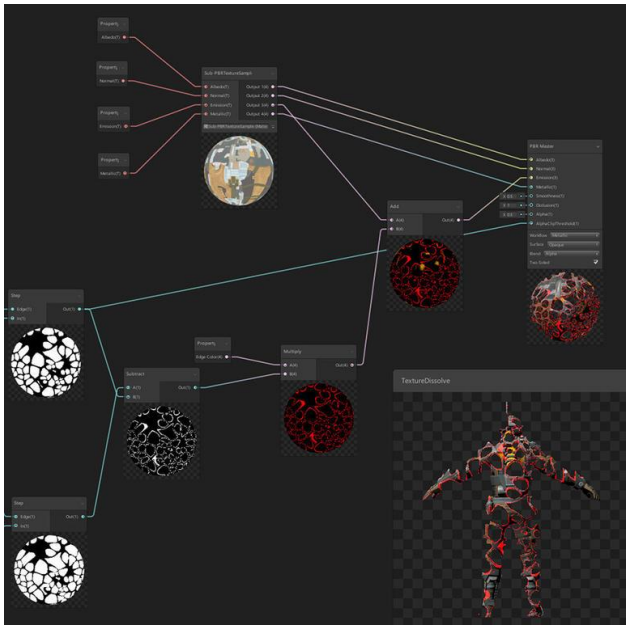

Figure 38: Shader graph example [29]

To create the alpha maps for each biome, I iterated through each coordinate and determined the biome. Once the biome was determined for that coordinate, instead of setting the vertex colour to the biome colour, I set the same coordinate in the alpha map for that biome to a colour. The texture for the biome was then passed to the shader graph. Once all coordinates had been checked, the alpha maps were passed to the shader graph. The code for this is shown in Figure 39.

```
            if (biome.name == "desert")
            {
                //add to alpha map
                alphaColorMaps[0][colorIndex] = new Color(1, 1, 0, 1);
                //pass texture to shader
                mat.SetTexture("_texture1", biome.texture);

            }
            if (biome.name=="boreal forest")
            {
                alphaColorMaps[1][colorIndex] = new Color(0, 1, 0, 1);
                mat.SetTexture("_texture2", biome.texture);
            }
            if (biome.name == "tundra")
            {
                alphaColorMaps[2][colorIndex] = new Color(1, 1, 1, 1);
                mat.SetTexture("_texture3", biome.texture);
            }
            if (biome.name == "savanna")
            {
                alphaColorMaps[3][colorIndex] = new Color(1, 0, 0, 1);
                mat.SetTexture("_texture4", biome.texture);
            }
            if (biome.name == "rainforest")
            {
                alphaColorMaps[5][colorIndex] = new Color(1, 0, 0, 1);
                mat.SetTexture("_texture6", biome.texture);
            }
            if (biome.name == "shrubland")
            {
                alphaColorMaps[6][colorIndex] = new Color(1, 0, 0, 1);
                mat.SetTexture("_texture7", biome.texture);
            }

            // save biome in chosenBiomes matrix only when it is not water
            chosenBiomes [zIndex, xIndex] = biome;

        } else {
            // water regions don't have biomes, they always have the same color
            alphaColorMaps[4][colorIndex] = new Color(0, 0, 1, 1);
            mat.SetTexture("_texture5", this.waterTexture);
        }
    }
}
Texture2D[] alphaTextures= new Texture2D[numberOfTextures];
for (int i = 0; i < alphaColorMaps.Length; i++)

{
    alphaTextures[i] = new Texture2D(tileWidth, tileDepth);
    alphaTextures[i].wrapMode = TextureWrapMode.Clamp;
    alphaTextures[i].SetPixels(alphaColorMaps[i]);
    alphaTextures[i].Apply();
}
```

Figure 39: Updated code for biome texture building

Once the textures and the alpha maps were passed to the shader graph, I used ‘combine’ nodes which allowed me to combine the red, green, and blue of the textures with the alpha values from the alpha maps. I then blended the output from each ‘combine’ node using the ‘overwrite’ mode which merged the textures so that the correct textures were showing on the correct vertices of the generated world. The output of all the ‘blend’ nodes were then fed into the ‘albedo’, which is the parameter that controls the base colour of the surface of the vertices. This is shown in Figure 40.

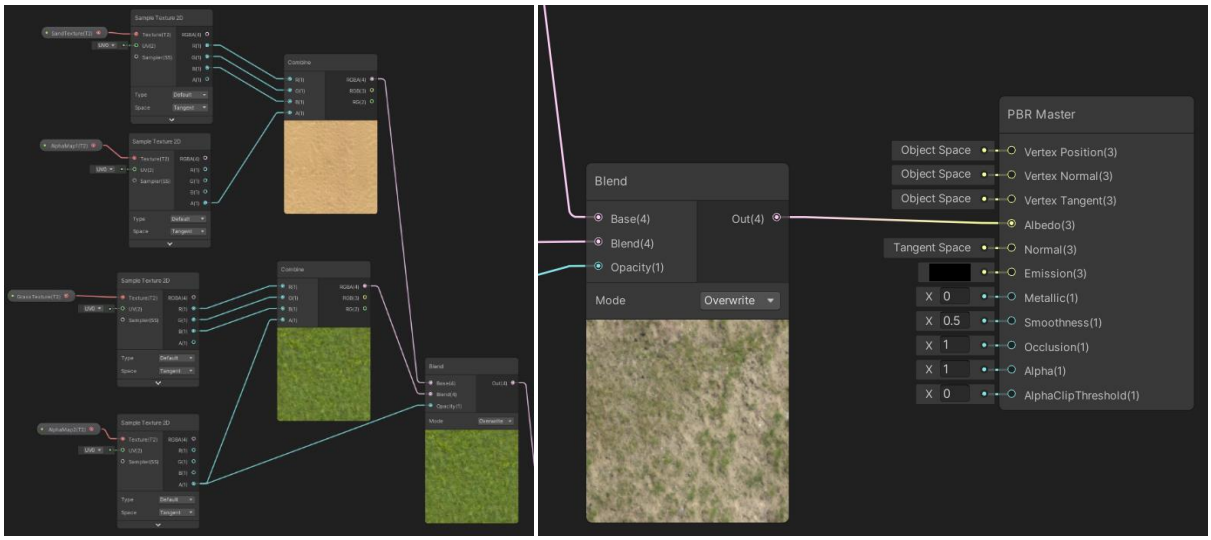

Figure 40: Shader graph for biome texturing

A limitation that was discovered with the shader graphs was that a maximum of 16 Texture2D objects were allowed to be used, this meant that only 7 biome textures plus a water texture each with their own alpha map could be passed to the shader graph. Figure 41 shows the generated world with 7 biomes.

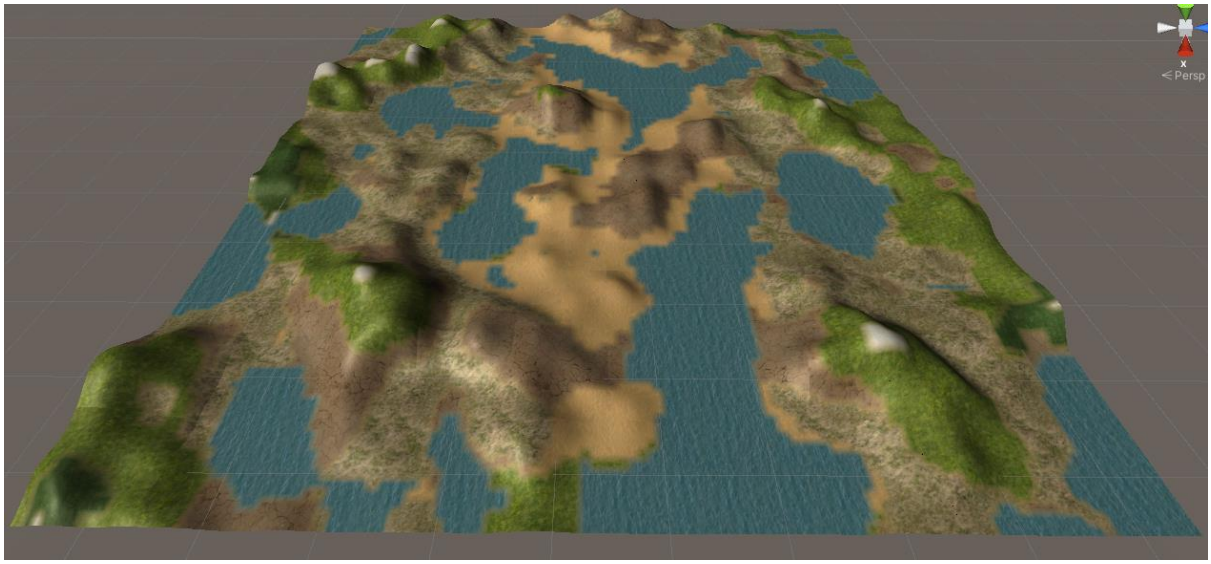

Figure 41: Generated biomes using shader graph

I, unfortunately, was unable to find a way to allow more than 16 Texture 2D parameters and therefore only 7 biomes maximum are able to be generated with textures in this application.

3.5.4 Adding a Flying Camera

The final step to the development of this application was to add a camera that the user can use to navigate around the world. Unity standard assets comes with a script called 'SimpleCameraController' which was added to the 'Main Camera' and worked perfectly in the application. It allowed full mouse and keyboard support of the camera and allowed and the user to increase the speed of the camera using the shift key. Figure 42 shows a screenshot of the camera in action.

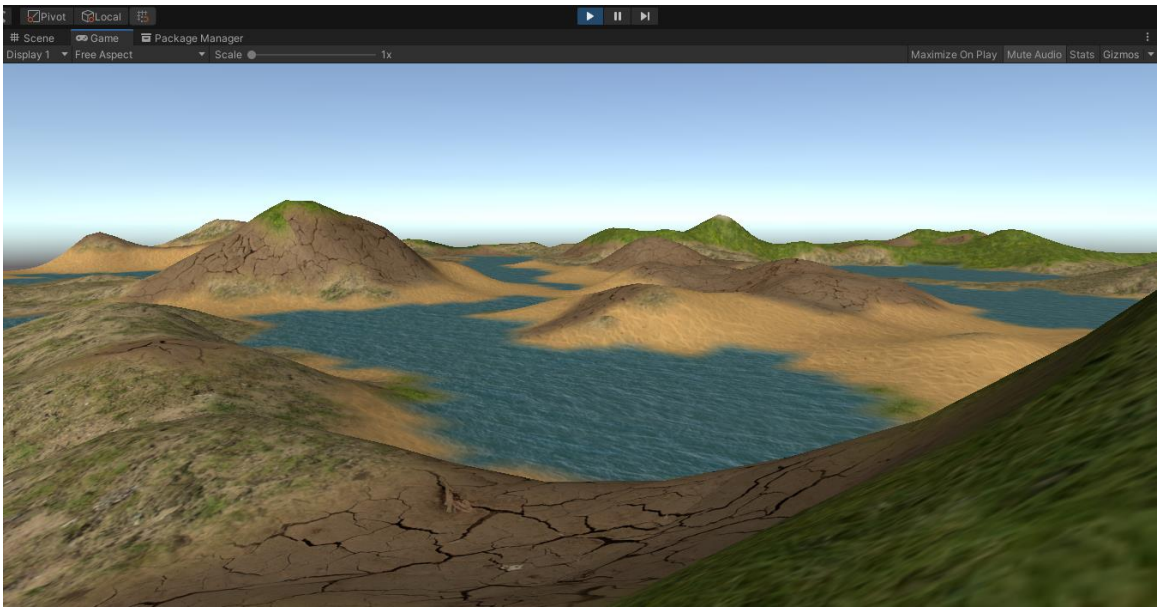

Figure 42: View from moveable in-game camera

Chapter 4: Results and Evaluation

4.1 Performance Analysis

Performance is important for the usability of this application. If the generation times of the application were too long, it would get frustrating for users that might want to generate multiple worlds or make slight adjustments after seeing the results of one of the generations.

To analyse the performance of the application, I collected a range of data. I analysed the average time taken to generate terrain within a given size. To ensure accurate results, the underlying framework of the application was the same; the only change was the size of the generated terrain.

The results are presented in the form of tables and graphs, with figures of the terrain generated.

	Generation Time (ms)					
Area of Terrain (tiles)	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Average**
1x1	8.217454	9.037495	9.414315	8.576035	8.850455	8.819151
5x5	34.43956	34.57499	35.22265	36.59570	35.42924	35.25243
10x10	106.4117	105.4038	108.9141	111.4621	106.5047	107.7393
20x20	380.4830	386.8484	391.1275	387.3270	392.7527	387.7077
50x50	2426.454	2377.181	2399.324	2408.056	2376.371	2397.477
100x100	9836.600	9822.040	9851.027	9799.701	9905.397	9842.953

Figure 43: Table to show the change in generation time when increasing the world size

Figure 43 shows the results of the performance analysis. I ran the tests 5 times and took an average to ensure that the results were as accurate as possible, and to rule out any possible outliers. Figure 43 shows the graph that was created using the averages and the "World Size". To make the graph easier to read, along the X-axis I used the World Size, which I define as the square root of the area of the terrain (in tiles).

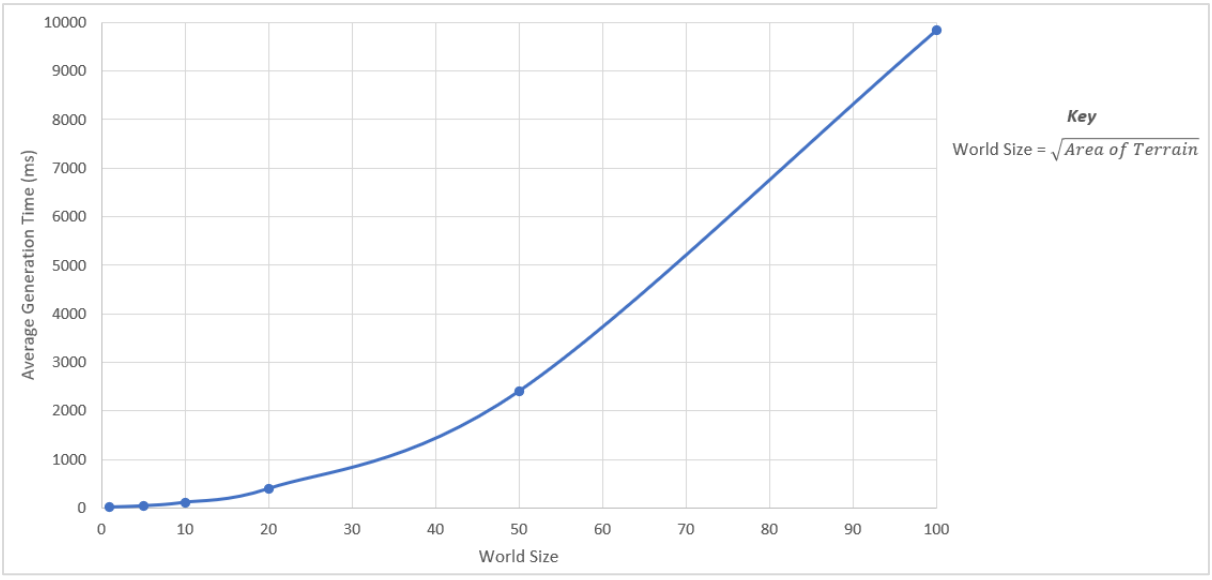

Figure 44: Graph to show average generation time for world sizes

The graph in Figure 44 shows an exponential increase in time, as the World Size increases. Therefore, the application performs the best when generating smaller sized terrain. While my

application has its limitations with generation times for bigger worlds, most games would not need worlds of 50x50 (or greater) tiles generated. However, if developers wanted to create bigger worlds, the longer generation time could be frustrating and is definitely something that needs to be improved.

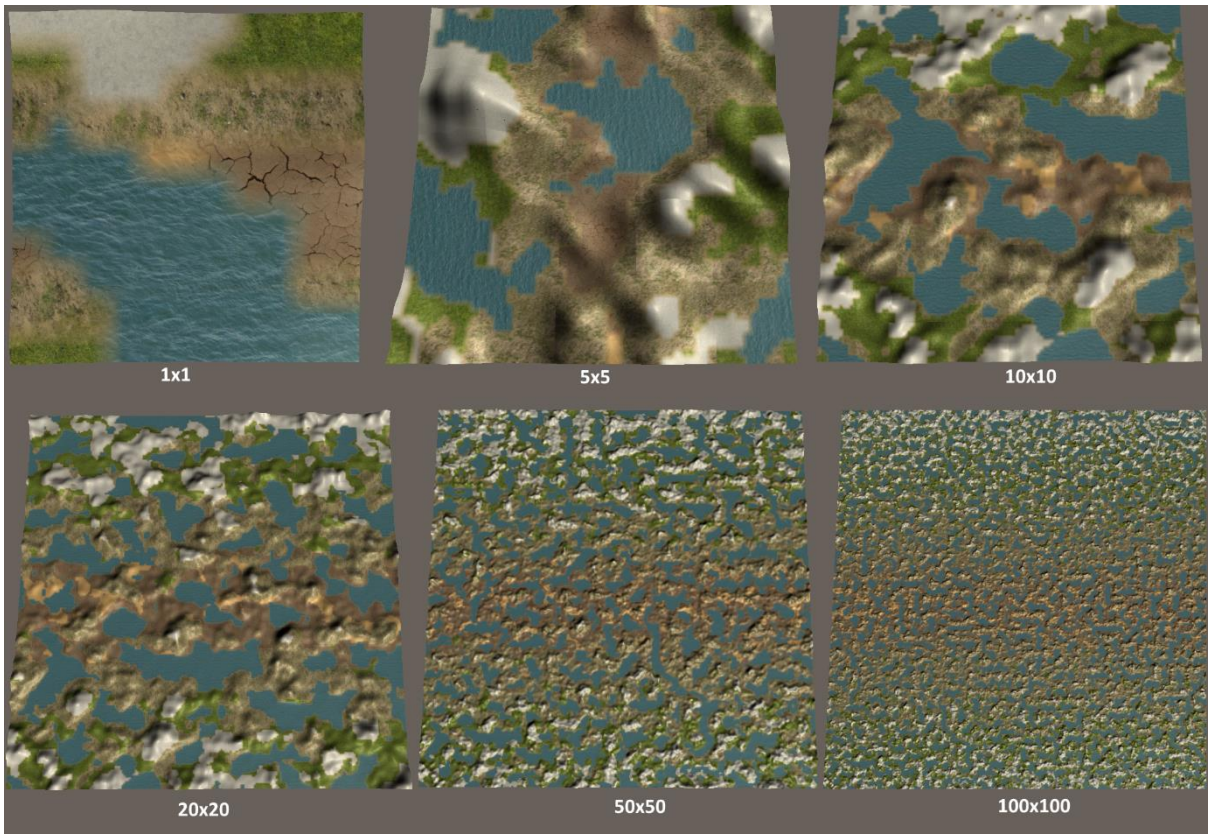

Figure 45: Performance Analysis Generated Terrain

Figure 45 shows the generated terrain for each sized terrain from the performance analysis. The same seed was used for each world to ensure accurate testing. From the larger worlds, it is clear that the uniform noise generation used for the heat map is working correctly; this is shown by the warmer biomes positioned around the centre of the terrain.

4.2 Quality Evaluation

The aim of this section is to analyse the quality of the application. To determine whether the application is of good quality, I must compare it against the application requirements outlined before development started, which were:

Requirement 1 – Controllable camera

- The user must be able to move around the generated word, without issues, using a flying camera.
- The camera must have full keyboard and mouse support.

Requirement 2 – Add biomes

- The user must be able to select and add biomes that they want to be generated in the world.

Requirement 3 – Adjust appearance of biomes

- The user must be able to adjust the height of the biomes.
- The user must be able to change the textures of the biomes.

Requirement 4 – Follow the biome-type model

- The application must follow the Whittaker biome-types model when generating biomes in the world.

4.2.1 Requirement 1 – Controllable camera

The application features a movable camera that was implemented using the Unity standard assets pack, which fully supports keyboard and mouse control. This allows users to fly around the world to inspect the generated terrain.

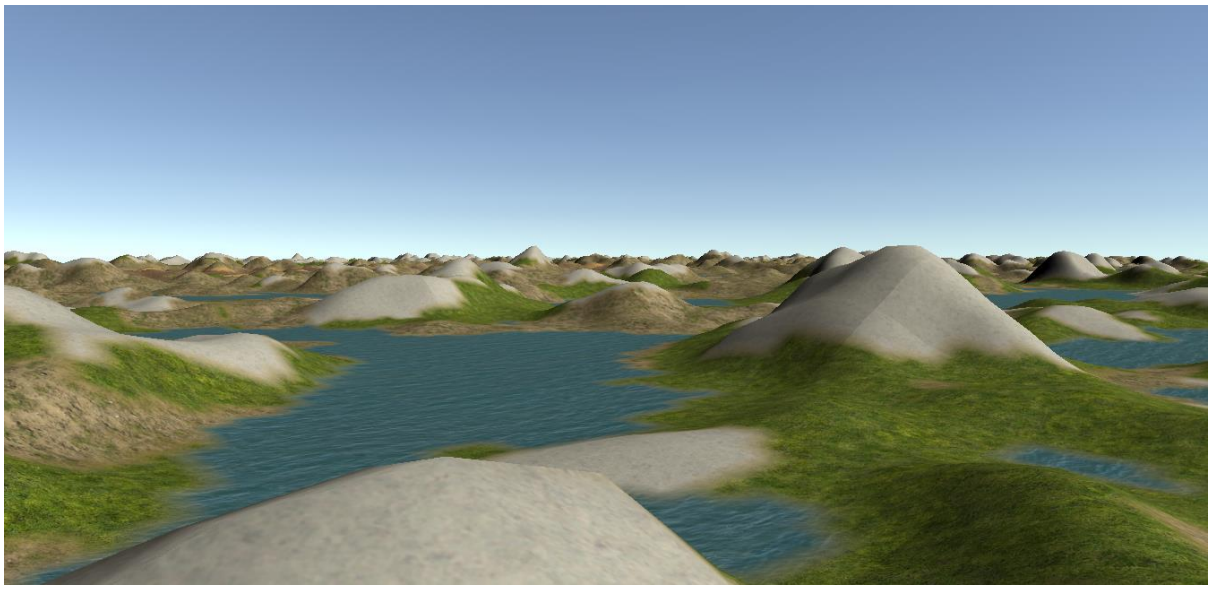

Figure 46: View from moveable camera

4.2.2 Requirement 2 – Add biomes

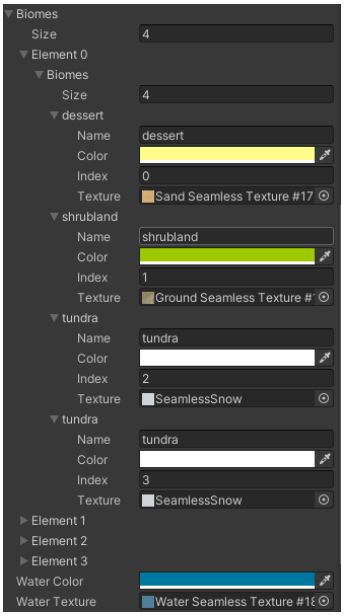

Figure 47: Biome customisation options

Figure 47 shows the options available for the user to customise the biome properties. Here users can change the number of columns and rows in the biome table (Figure 32) by changing the size variables. The outer size variable changes the number of rows and the inner size variable inside the

element tab changes the number of columns for that row. The user can use this area to add more types of biomes.

4.2.3 Requirement 3 – Adjust appearance of biomes

Biome appearance

As shown in figure 47, the user is able to change the colour and texture of the biomes. The reason for both options is because of the limitation with Unity Shader Graphs. This limitation means that a maximum of 8 textures can be used in the generation of the terrain.

Height customisation

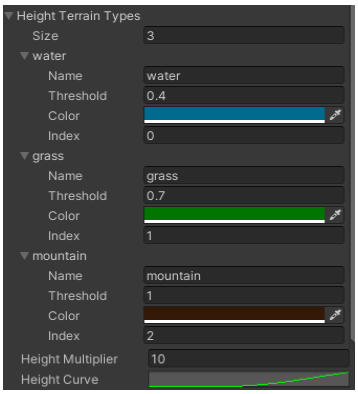

Figure 48: Height customisation options

Figure 48 shows the options available for customising the height of the terrain. It allows users to adjust the existing height threshold values which are used to help determine the biome to be generated. Users can add more height types by increasing the size variable at the top. Users can also adjust the height multiplier to create terrain with more drastic looking mountains or flatter terrain.

4.2.4 Requirement 4 – Follow the biome-type model

The terrain is randomly generated using Perlin noise values. Using the Perlin noise values alongside the biome table that I set up which follows the Whittaker biome-types model (shown in Figure 32), it is impossible for biomes with completely different characteristics to generate next to each other. This is because Perlin noise is a gradient-type noise function, meaning there are no sudden changes in the values. To give an example of how this is used in the application, if the values of moisture on an area if the terrain is the same (in this example I will use “driest” as the moisture value) but the temperature of that area ranged from “hottest” to “coldest”, it would be impossible for a desert to generate next to a tundra biome. This is because the temperature values of the area must transition to “hot” and “cold” before reaching “coldest”, meaning that a grassland biome would generate between the desert and the tundra biomes. Therefore, this application follows Whittaker’s biome-types model.

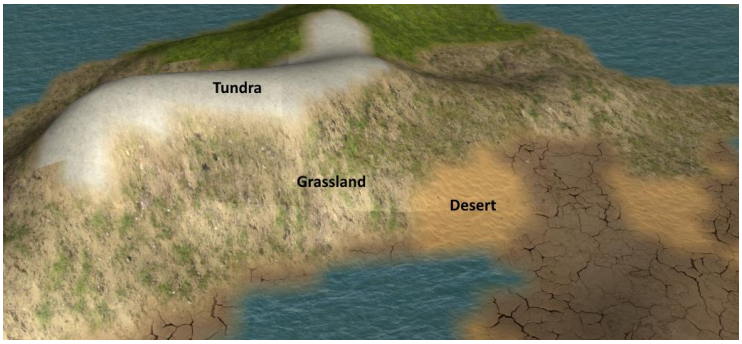

Figure 49: Transition from desert to tundra

4.3 Summary

From the data collected, I know that my application is most suited to generating smaller sized worlds, however, it is not as suitable for creating larger sized worlds due to longer generation time which could be frustrating for the users. The application meets all of the requirements that were outlined, and that the application is effective at generating terrain which follows Whittaker's biome-types model. To improve the application, further development could be done to reduce generation times, to allow for the generation of structures (like trees, caves, and rivers) and to allow for more textures to be used.

Chapter 5: Conclusion

5.1 Satisfaction of the aims and objectives

The aim of this dissertation was to develop an application to procedurally generate terrains with a focus on generating realistic biome formations. I broke down the aim into objectives that needed to be met so that the overall aim could be met.

1. **Identify techniques for procedural terrain generation.**

This objective was met. I identified the diamond-square algorithm, value noise, Perlin noise, simplex noise, and Worley noise as techniques that are used in industries. Once identified, I compared the procedural generation techniques to find the one which would be most suited to my dissertation.

2. **Research existing biome-type models.**

This objective was met. I researched Holdridge's life zones, Walter's zonobiomes, and Whittaker's biome-types, which are all pre-existing biome-type models that are used in modern literature.

3. **Identify suitable game engines for procedurally generating terrains.**

I looked into the current most popular game engines (which were Unreal Engine, Unity and CryEngine) and identified their advantages and disadvantages. I compared the identified game engines to find the one most suitable for this dissertation. Therefore, this objective was met.

4. **Develop a prototype that will generate terrains with biomes based on the previous objectives.**

This objective was fully met. The prototype was developed and meets all of the application requirements set in Chapter 3, allowing the user to: navigate around the generated terrain, add/remove biomes to be generated, and customise the appearance of the generated terrain. The prototype uses the methods researched in objectives 1, 2 and 3, and successfully generates terrain which follows the chosen biome-type model.

5. **Summarise and evaluate the performance of the application.**

A performance analysis was undertaken on the application prototype. This looked at the generation time of the terrain, as the terrain size increased. A quality evaluation then took place, which assessed whether the prototype met all of the application requirements that were set. Therefore, this objective was fully met and, subsequently, the overall aim of the dissertation was met.

5.2 What went well

Overall, I think that this project went well and that the research and techniques that I used for the terrain generation were the best that I could have used for this dissertation. I believe that the performance testing was to a high standard as I ensured that testing was fair and comparable. The constant testing that I performed throughout development reduced many future risks and allowed me to identify the problem areas when errors were encountered. The chosen biome-type model and game engine worked well for this dissertation; however, I would like to branch out and try some of the other options in the future like Unreal Engine.

5.3 What could have been done better

The quality evaluation is based solely on if the application requirements were met and my opinion. If I could do this dissertation again, I would get external people to test the prototype and ask their opinion on if the prototype met the application requirements and if it met them well. I was unable to get external testers due to current circumstances with lockdown and COVID-19, meaning that most people did not have powerful enough computers which are required to be able to run my application. From this, I could have tested computers of different specifications and determined the minimum and recommended computer specifications required to run my application.

A limitation that I found during development was that only a maximum of 16 Texture2D objects could be passed to the Unity Shader Graph. Because of the way that I applied textures to the terrain vertices, this meant that I could only use 8 textures maximum. This was very disappointing, as the biome-type model I was following (Whittaker's biome-types) used 9 biomes; therefore, I was unable to add textures to all of the biomes.

5.4 What was learnt

Before starting this dissertation, I had little experience with Unity and Shader Graphs. During this dissertation, I have greatly improved my knowledge of Unity and I am now confident in creating Shader Graphs to improve the appearance of objects.

I had no experience with procedural generation techniques prior to this dissertation. I now know some of the most used techniques and their reasons for use. In relation to generating realistic terrain, I learned that gradient-type noise algorithms are the best at creating natural-looking terrain and that the most used method in industry is Perlin noise.

5.5 What could be done in the future

Additional steps could be taken to allow for structures to be generated alongside the terrain (e.g., trees, caves, rivers, and waterfalls). I would also like to explore different ways of applying texture to terrain, to enable me to use more than 8 textures on the generated terrain. I could also perform additional testing to cover different performance aspects such as frames per second, RAM usage and GPU usage. I would also like to get external users to test the application and give their feedback. I would also like to determine the minimum and recommended computer specifications required to run the application, and the performance differences when running on different operating systems and hardware.

References

[1] Berkeley.edu. (2019). The world's biomes. [online] Available at: https://ucmp.berkeley.edu/exhibits/biomes/index.php. [Accessed 07 04 2021].

[2] D. Wilcox-Netepczuk (2013). Immersion and realism in video games - The confused moniker of video game engrossment. In Proceedings of CGAMES'2013 USA (pp. 92-95).

[3] Van Brummelen, J. and Chen, B., (2021). Procedural Generation. [online] Mit.edu. Available at: http://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html. [Accessed 11 04 2021].

[4] fuzavella (2018). Original Rogue (1980) duneon generation. [online] Available at: https://www.reddit.com/r/roguelikedev/comments/99b5fe/original_rogue_1980_duneon_generation/ [Accessed 11 Apr. 2021].

[5] britzl.github.io. The Rogue Archive. [online] Available at: https://britzl.github.io/roguearchive/ [Accessed 11 04 2021].

[6] The New Stack. (2017). The New Gaming Frontier of Procedurally-Generated Universes. [online] Available at: https://thenewstack.io/new-crop-games-built-procedurally-generated-universes/. [Accessed 11 04 2021]

[7] Engadget. (2015). Here's how "Minecraft" creates its gigantic worlds. [online] Available at: https://www.engadget.com/2015-03-04-how-minecraft-worlds-are-made.html [Accessed 11 Apr. 2021].

[8] Baker, C. and Baker, C. (2016). "No Man's Sky": How Games Are Building Themselves. [online] Rolling Stone. Available at: https://www.rollingstone.com/culture/culture-news/no-mans-sky-how-games-are-building-themselves-104779/. [Accessed 11 04 2021].

[9] Academy, B. and Bontchev, B. (2016). Serdica Journal of Computing MODERN TRENDS IN THE AUTOMATIC GENERATION OF CONTENT FOR VIDEO GAMES. ACM Computing Classification System, [online] 10(2), pp.133–166. Available at: https://core.ac.uk/download/pdf/156902177.pdf. [Accessed 12 04 2021].

[10] jmecom.github.io. (2015). Procedural Terrain Generation: Diamond-Square. [online] Available at: http://jmecom.github.io/blog/2015/diamond-square/. [Accessed 12 04 2021].

[11] http://www.redblobgames.com/articles/noise/introduction.html, "Noise Functions and Map Generation," Red Blob Games, 31 08 2013. [Online]. Available at: http://www.redblobgames.com/articles/noise/introduction.html. [Accessed 12 04 2021].

[12] T. J. Rose, and A. G. Bakaoukas 2016. Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques. In 2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES) (pp. 1-2).

[13] Gustavson, S. (2015). Simplex noise demystified. [online] https://weber.itn.liu.se. Available at: https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf [Accessed 15 04 2021].

[14] rtouti.github.io. Perlin Noise: A Procedural Generation Algorithm. [online] Available at: https://rtouti.github.io/graphics/perlin-noise-algorithm. [Accessed 14 04 2021].

[15] Rosén, C.-J. (2006). Cell Noise and Processing. [online] http://www.carljohanrosen.com. Available at: http://www.carljohanrosen.com/share/CellNoiseAndProcessing.pdf. [Accessed 15 04 2021].

[16] kazilek (2013). Biomes of the World | Ask A Biologist. [online] Asu.edu. Available at: https://askabiologist.asu.edu/explore/biomes. [Accessed 16 04 2021].

[17] Bio.miami.edu. 2021. BIL 330 - Lecture 9. [online] Available at: http://www.bio.miami.edu/dana/330/330F19_9.html. [Accessed 16 04 2021].

[18] Ruzinoor Che Mat, Abdul Rashid Mohammed Shariff, Abdul Nasir Zulkifli, Mohd Shafry Mohd Rahim, and Mohd Hafiz Mahayudin 2014. Using game engine for 3D terrain visualisation of GIS data: A review. IOP Conference Series: Earth and Environmental Science, 20, p.012037.

[19] www.exsquared.com. (2014). Game Engines are Serious Business | EX Squared. [online] Available at: https://www.exsquared.com/blog/game-engines-are-serious-business/. [Accessed 18 04 2021].

[20] Perforce Software. (2019). What Are the Most Popular Game Engines? [online] Available at: https://www.perforce.com/blog/vcs/most-popular-game-engines. [Accessed 18 04 2021].

[21] docs.unrealengine.com. (2014). Creating Landscapes. [online] Available at: https://docs.unrealengine.com/en-US/BuildingWorlds/Landscape/Creation/index.html. [Accessed 18 04 2021].

[22] Juego Studio. (2020). 9 Major Advantages of Unity 3D Game Engine: An Ultimate Game Developing Tool. [online] Available at: https://www.juegostudio.com/blog/advantage-of-unity-3d-game-development/. [Accessed 18 04 2021].

[23] Technologies, U. (2021). Unity - Manual: Terrain. [online] docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/script-Terrain.html [Accessed 18 Apr. 2021].

[24] www.quora.com. (2019). What are the pros and cons of Unreal Engine and CryEngine? - Quora. [online] Available at: https://www.quora.com/What-are-the-pros-and-cons-of-Unreal-Engine-and-CryEngine/. [Accessed 18 04 2021].

[25] PMT. (2021). OCR Computer Science A-level: Software and Software Development Revision. [online] Available at: https://www.physicsandmathstutor.com/computer-science-revision/a-level-ocr/software-and-software-development/ [Accessed 21 04 2021].

[26] Pluralsight.com. (2019). Everything you need to know about C# | Pluralsight. [online] Available at: https://www.pluralsight.com/blog/software-development/everything-you-need-to-know-about-c-. [Accessed 21 04 2021].

[27] Asad (2016). 10 Advantages of C# Programming Language. [online] ProProgramming. Available at: http://proprogrammershub.blogspot.com/2016/04/top-10-advantages-of-c.html. [Accessed 21 04 2020].

[28] 3djungle.net. (2016). 3D models, textures and materials for design and visualisation. [online] Available at: https://3djungle.net. [Accessed 27 04 2021].

[29] Technologies, U. Unity Shader Graph | Build Your Shaders Visually with Unity | Rendering & Graphics | Unity. [online] unity.com. Available at: https://unity.com/shader-graph. [Accessed 27 04 2021].

[30] Alan Zucconi. (2015). A gentle introduction to shaders in Unity - Shader tutorial. [online] Available at: https://www.alanzucconi.com/2015/06/10/a-gentle-introduction-to-shaders-in-unity3d/. [Accessed 27 04 2021].