Procedural Terrain Generation

Xiang Li (190471866)
May 2022
BSc Computer Science (Game Eng)
Supervisor – Dr Graham Morgan

Word Count: 11,297

Abstract

The procedural terrain generation is one of the core techniques in modern game development. The idea of this technique is to create game contents, such as terrain, in an automated way and they look as good as when designers do it. This dissertation develops a procedural terrain generation in Unity engine, which allows user to automatically generate terrain of any random shapes, different textures, and infinite size in real-time. The project firstly focused on generating pseudorandom height maps for terrains through Perlin Noise and filled those height maps with multiple triangles and vertices. Then after developing the endless terrain system and texture shader, a custom UI were implemented at the end to help user easily adjust various features through my procedural terrain generation. Lastly, test and results are presented for its performance evaluation.

Acknowledgements

I would like to thank my supervisor, Dr Graham Morgan, for the advice and support provided through the entire project. I also want to thank Dr Gary Ushaw and Dr Rich Davison, who have also provided guidance when I needed help.

Declaration

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

Table of Contents

Chapter 1: Introduction 6
1.1 Context 6
1.2 Purpose 6
1.3 Aim and Objectives 7
1.3.1 Aim 7
1.3.2 Objectives 7
1.4 Dissertation structure (write more later) 8
Chapter 2: Background 8
2.1 Research Strategy 8
2.2 Background Research 8
2.2.1 Procedural Terrain Generation 8
2.2.2 Noise functions 12
2.2.3 Unity Documentation 15
2.3 Summary 16
Chapter 3: Method/Methodology 17
3.1 Overview/Introduction 17
3.2 Planning/design 17
3.3 Tools 18
3.4 Functionality Outline (write more later) 18
3.5 Development/Implementation 19
3.5.1 Introduction 19
3.5.2 pseudorandom terrain map 19
3.5.3 Terrain Mesh Generator 25
3.5.4 Endless terrain system 28
3.5.5 Shader 33
3.6 Summary 34
Chapter 4: Testing & Evaluation 34
4.1 Overview 34
4.2 Testing tool 34
4.3 Testing Solution 35
4.4 Results Evaluation 38
4.5 Summary 41
Chapter 5: Conclusion 42
5.1 Satisfaction of the Aim and Objectives 42
5.2 What went well 43
5.3 What could have been done better 43

5.4 Personal development 43
5.5 Future work 44
References 45
Appendices 48

Table of Figures

Figure 1: No Man’s Sky [8] 9
Figure 2: Houdini Tool in Ghost Recon: Future Soldier [9] 10
Figure 3 [10] 10
Figure 4 [12] 11
Figure 5: Left is the heigh map from noise. Right is shaded terrain based on height map [13] 11
Figure 6: 1D coherent-noise function n(x) [14] 12
Figure 7: 2D Value Noise [15] 13
Figure 8: 1D comparation between Perlin noise and Value Noise [17] 13
Figure 9: 2D Perlin Noise [15] 14
Figure 10: Whorley Noise [18] 15
Figure 11: Unity documentation: Graphics (Left) and Scripting API (Right) [19] 15
Figure 12: Source code of Mathf.PerlinNoise [20] 16
Figure 13: Waterfall development process in Gantt chart [24] 17
Figure 14: 2D Perlin noise at 25 scale 20
Figure 15: Left is when Octave equals 2, Right is when Octave equals 8 [14] 21
Figure 16: Under the same values with S=25, O=5, P=0.5. Left is when L=1. Right is when L=2. 22
Figure 17: Under the same values with S=25, O=5, L=2. Left is when P=0.5. Right is when P=1 22
Figure 18: Under the same value with S=25, O=5, L=2, P=0.5. Left is when Seed=1. Right is when Seed=2. 23
Figure 19: Editor page for setting terrain and colour 24
Figure 20: Left is height map from Perlin noise. Right is its early stage of terrain map 25
Figure 21: square grid is created by two triangles 26
Figure 22: 2D terrain mesh 27
Figure 23: 3D terrain mesh at height of 15 27
Figure 24: Left is the height map curve. Right is a 3D terrain mesh affected by the curve. 28
Figure 25: Boundary between loaded chunk and unloaded chunk [29] 29
Figure 26: Ideal binary tree [30] 29
Figure 27: Mesh in LOD = 0. The white circles represent vertices (also can be referred as the width and height of the mesh) [31] 30
Figure 28: Mesh in LOD = 1. The black circles represent new vertices in LOD1. [31] 30
Figure 29: Left is LOD = 0. Middle is LOD = 2. Right is LOD = 4. 31
Figure 30: Level propagation in Minecraft [29] 31
Figure 31: Endless terrain. Centre has the LOD of 0, visible distance of 200. Sides have the LOD of 4 to 6 and the visible distance of 400 to 600. 33
Figure 32: Left is the terrain only with Colour shader. Right is the terrain with Texture shader override on the Colour shader. 34
Figure 33: Unity Profiler [33] 35

Figure 34: Top is terrain map setting. Middle is terrain mesh setting. Bottom is LOD and visible distance setting....36
Figure 35: 3D terrain with no shader and texture....36
Figure 36: 3D terrain at early stage only with single colour pixel....37
Figure 37: 3D terrain with colour shader....37
Figure 38: 3D terrain at final stage with colour and texture shader....37

Table of Code

Code 1: MapGenerator class....19
Code 2: Initial Implementation of Perlin Noise in Unity....20
Code 3: Apply octaves into noise map....21
Code 4: Implementation of pseudorandom generator for noise map....23
Code 5: Final Implementation of Perlin Noise in Unity....23
Code 6: Colour map method....24
Code 7: MeshData struct in MeshGenerator....25
Code 8: Setting vertices and UVs to the mesh....26
Code 9: Setting triangles to the mesh in colockwise....26
Code 10: Setting height map curve....28
Code 11: Calculation of chunk size, visible distance, and chunk coordinate....32
Code 12: LODMesh class for storing mesh and LOD data....32
Code 13: Updating the terrain chunks through different LOD in specific visible distance....32
Code 14: surf function. The key function of Surface shader....33

Table of Chart

Chart 1: Frame rate comparison of different testing objects....38
Chart 2: Comparison of memory consumption of different testing objects....40
Chart 3: Comparison of rendering time of different terrain outputs....41

Chapter 1: Introduction

This chapter aims to introduce the project to the reader, providing the purpose of the project along with the aim and objectives. Last but not the least, the overall structure of this dissertation will also be presented.

1.1 Context

In this digital age, gaming becomes one of the most popular activities all around the world. Especially during the global COVID-19 pandemic, more and more people are looking for joy and excitement in various virtual world.

> *"Looking back at gamer spending in 2020, worldwide digital gaming spending on in-game content and paid downloads has increased by 12 percent and 21 percent respectively, highlighting the growth of digital revenues." [1]*

Lots of gamer love spending their time on exploring randomness and challenges in the realistic open world of modern games, such as *Minecraft*, a game which can generate 2.8 trillion different possible worlds in only a few hundred megabytes [2]. Moreover, people never get bored when driving spaceship around over 18 quintillion possible planets in the universe of *No Man's Sky*. [3]

Hence, to create those gigantic and vivid virtual worlds successfully, terrain generation has been introduced as one of the core techniques in modern game development.

1.2 Purpose

As the game industry grew, the time and cost of developing games became longer and more expensive. Although lots of game engines contain build-in set of Terrain features that allow user to create environment easily, generating terrain manually is a finite resource and difficult to create the vast landscapes with different types of infinite and random terrain demanded by modern gamers. For example, the Terrain features in Unity editor still need to import additional terrain toolbox for a variety of painting on complex-looking terrain. [4] When requiring a large endless surface, users also need to create neighbour terrains and redesign a new shape of that terrain. Therefore, instead of spending large number of resources on employing many artists and designers to create contents, more and more game studios prefer to seek cheaper and more efficient ways to develop games.

However, the technical challenge here is to create game contents, such as terrain, in an automated way and they look as good as when designers do it. This is where the concept of procedural generation be introduced. Procedural generation is a method of creating contents using a certain logic or algorithm instead of doing it manually. [5]

> *"One of the main costs of developing a videogame is content creation. Procedural Content Generation (PCG) can help alleviate that cost by algorithmically generating some of the content a human would normally produce." [6]*

This project intends to implement a procedural terrain generation in Unity engine, which will allow user to automatically generate terrain of any random shapes, different textures, and infinite size in real-time. The main idea is to create pseudorandom content by computer instead of human.

1.3 Aim and Objectives

1.3.1 Aim

The aim of this project is to implement a procedural and dynamic terrain generation using Unity engine that can help modern games populate vast surface areas. It will be able to render an “endless” scrolling surface in real-time with different types of terrains and textures.

1.3.2 Objectives

The aim is divided into the following objectives

1. **Explore the usage of noise function in creating terrain and identify the best function that can be implemented in Unity to generate random heightmaps**
   The research of different noise function in creating terrain will provide a better understanding on which noise can allow changes to occur gradually and generate pseudorandom height maps for terrains without feeling chaotic. Analysing height maps using Noise will help generate random terrain.

2. **Research the data of mesh in Unity and create a terrain mesh by generating heightmaps**
   It is important to understand what Unity engine uses to make up a mesh and how those data, such as vertices and triangles, affect objects. Specifically, those mesh data fill the height map with multiple triangles and set the height values of individual vertices to generate different terrains.

3. **Understand the usage of Level of Detail (LOD) and implement the endless terrain system for terrain generation.**
   This step will be useful to implement endless terrain. The main idea of endless in this project is to limit the player to the visible chunks of terrain. Those chunks will be hidden once they are no longer visible. Therefore, it is important to understand the usage of LOD in Unity when large maps make up multiple meshes. The LOD can render fewer triangles and vertices at a certain distance to make the game running smoother. [8] This will change the LOD dynamically based on how far away the chunks are from the player. Once the endless terrain implements successfully, this objective will be achieved.

4. **Implement custom terrain shader and UI for better visualization**
   For this objective, adding supports on colour and texture to terrain generation will provide a better visual effect.

5. **Summarise and evaluate the performance of terrain generation**
   Last but not the least, in the final stage of this project, the terrain generation will be tested whether it can render infinite and random terrain smoothly at standard running frame rates. In addition, rendering time need to be measured for the comparison of performance speed. Moreover, when the complexity of the terrain increases, the computational cost, memory cost and framerate also need to be evaluated visually. The objective will also be used to decide if the aim would be achieved.

1.4 Dissertation structure

Chapter one not only elaborates on the context and motivation of the whole project, but also provides the aim and objectives to achieve the final implementation.

Chapter two reviews the relevant research strategies and background of the project in detail and examines the objectives established in Chapter one.

Chapter three specifically explains the methodologies behind the implementation of the project and describes how those methods match with the functionality requirements.

Chapter four provides a test solution to identify the performance of each result and analyse the efficiency of the project base on the test result.

Chapter five presents a conclusion to explain whether the project met the aim and objectives from Chapter one or not. In addition, the project will also be summarised and evaluated critically about what knowledge author has learned and what section can be improved in the future.

Chapter 2: Background

This chapter mainly elaborates on the fundamental concepts and research of the project. The various methods of procedural terrain generation techniques are introduced along with definition of different noise functions. Finally, the ideas for generating endless terrain and some Unity documentation approaches are also included.

2.1 Research Strategy

The following research was presented by a mix of scientific and technical approaches. The main focus is on how the contribution and analysis of the article relate to this project

Scientific approach

Research paper on the concept of PCG, the analysis of algorithm on terrain generation and the usage of noise functions were all found through google scholar and free published academical reports. In addition, pictures for better understanding are collected from other major game development websites and posts.

Technical approach

Tool based explanation on Unity were all found through Unity Documentation

2.2 Background Research

The technical challenge here is to automatically generate terrain in Unity and they look as good as when designers do it. The research undertaken intends to find the solutions.

2.2.1 Procedural Terrain Generation

This section looked at the idea of procedural content generation and how it be used as technique for procedurally generating terrain.

2.2.1.1 Procedural Content Generation (PCG)

As PCG be mentioned in Chapter one, it is a programmatic generation of game content using random or pseudo-random processes. The idea of using procedural content generation is to reduce the resources and cost of game development.

> *"One of the main costs of developing a videogame is content creation, with some researchers estimating it to be around 30%–40% of the US$20M–US$150M average budget for AAA games. This content production requires highly specialized personnel, limiting the possibilities of small studios and increasing costs in bigger companies. Fortunately, PCG techniques enable the algorithmic creation of content without (or with limited) human effort."* [6]

The PCG can be mainly distinguished into two types, Online and Offline [7]. To be specific, online means content generation is performed at the runtime of the game. The generation process can be executed continuously during loading or while the game is running. It automates the game changes and greatly speeds up development time. For example, in Figure 1, every game contents in No Man's Sky are procedurally generated at runtime.

Figure 1: No Man's Sky [8]

On the other hand, offline means content generation is created and utilized by artists. It is often generated in layers. The bottom-level data provides a framework for the overall contents that can support the opportunity for designer to add complex details at the top layer. It makes the game more good-looking but time consuming. For example, in Ghost Recon: Future Soldier, the wires were placed by artists through the tool in Figure 2 instead of generating by the editor.

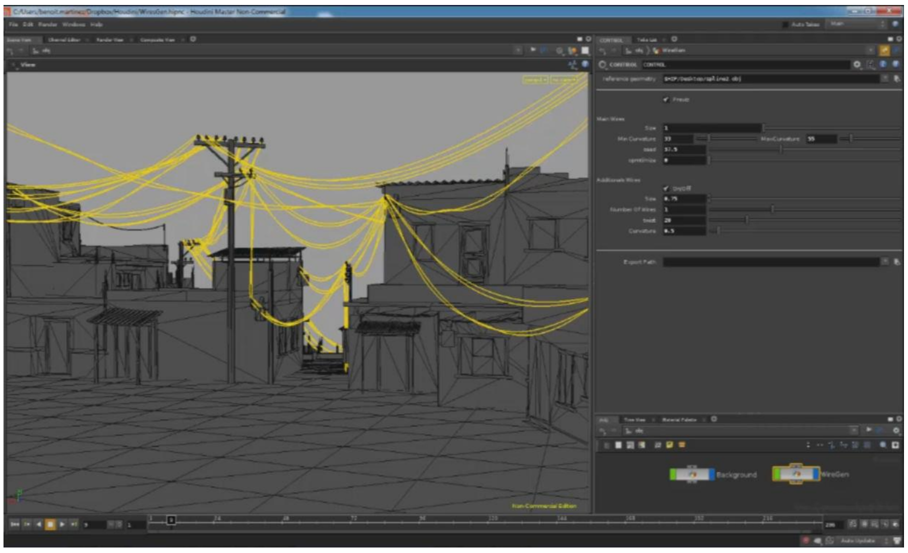


Figure 2: Houdini Tool in Ghost Recon: Future Soldier [9]

As a result, based on the concept of PCG, this project is going to use online procedural generation process to create terrain in real-time. Hence, it is also important to look at various techniques of procedural terrain generation, in which can be easily achieved to automatically generate a random and infinite terrain.

2.2.1.2 Different algorithms for procedurally generating terrain

Midpoint displacement algorithm

The first algorithm for procedurally generating terrain is midpoint displacement algorithm. This is one of the easiest and simplest method for automatically creating large landscape. Basically, the algorithm begins with a straight line between two points. Then it takes a point in the middle of the line that becomes the average of two outer points plus a random value.

As shown in the Figure 3, by displaying a square grid as a simple terrain on the X-Z plane, the algorithm first divides the four sides of the square into four grids by taking a midpoint. The resulting four midpoints and one center point are then displaced vertically by a random value in the Y-direction. Finally, whenever a new square is added, repeat the above steps.

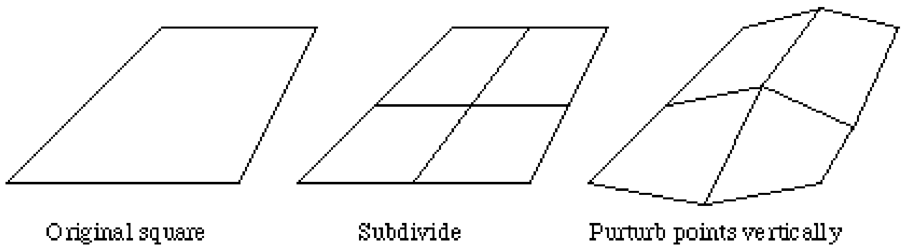


Figure 3: Example of square grid [10]

Although the algorithm can render at the fastest speed and retain acceptable quality, it consumes too much memory and has distinct angles to the image. [11]

Diamond-square algorithm

The diamond-square algorithm is a better version of midpoint displacement algorithm on generating terrain. It is also known as the random midpoint displacement fractal. This algorithm is generally divided into two steps, the diamond step and square step. In the diamond step, the midpoint of each square is determined using the average of the four points of the surrounding square plus a random value. While in square step, the other midpoints from diamond were determined by setting the midpoint of that diamond to the average of the four corner points plus another random value. Finally, these steps will be iterated until all the points on the terrain are given proper values. The diamond-square algorithm can be shown graphically in Figure 4.

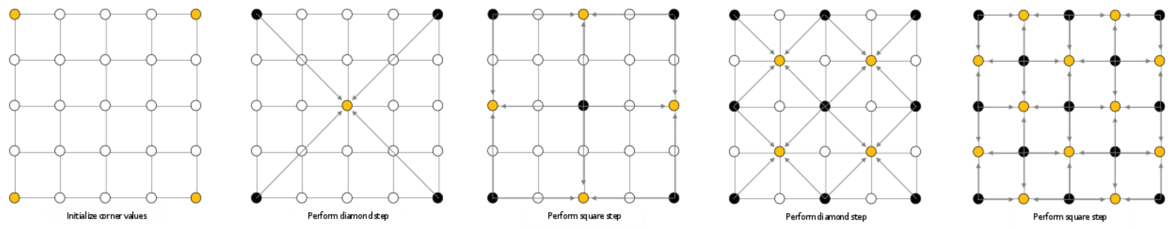


Figure 4: Diamond-square algorithm [12]

Hence, the diamond-square algorithm at each iteration is:

> *"Displacement amount = avg + scale*random" [10]*

The avg is the average value of the corner points, and the magnitude of random value should be multiplied by the scale of 1/2H, where H controls the roughness of terrain. Overall, the diamond-square algorithm is essentially the same as previous algorithm rendering at almost the same speed. Even though, it still costs lots of memory but provides a better quality. [11]

Noise algorithms

> *"Procedural Content Generation (PCG) almost always uses some form of noise, and games especially are benefiting from noise. Games use noise to add realism to hand-crafted models, but realistic landscapes generated procedurally are finding greater use." [11]*

The noise algorithm is one of the most popular terrain generation algorithms, which focuses on adding different layers of coherent noise and creating more realistic pseudo-random terrain types. Although noise by itself is just series of random data, it often be used to quickly generate elevation (height map). This process can show as an example in Figure 5.

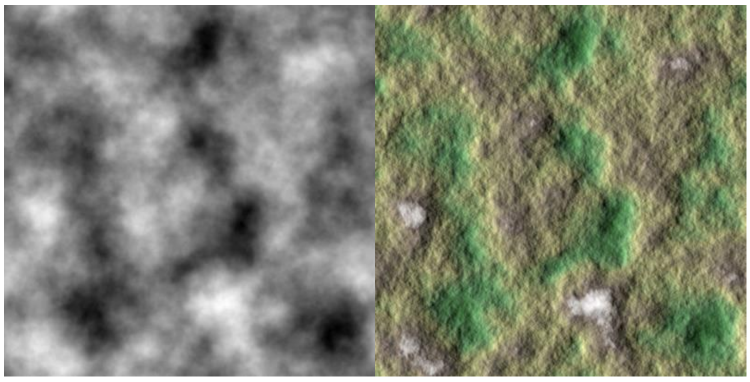


Figure 5: Left is the heigh map from noise. Right is shaded terrain based on height map [13]

By taking a slice of noise map, each at different frequencies and amplitudes, it will get a section of 1-dimensional coherent-noise function, in which be shown in Figure 6. Then, each section is commonly referred to as octaves and can be combined to increase different details to the terrain by adjusting the lacunarity and persistence of octave.

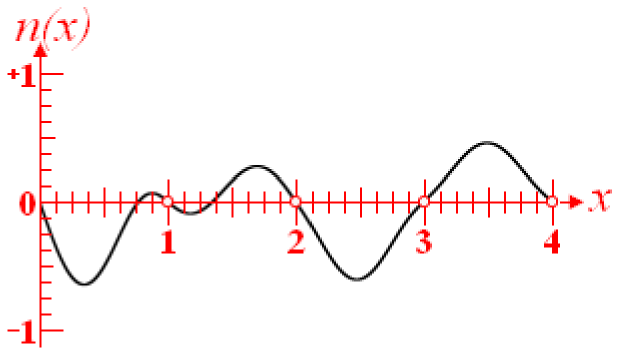


Figure 6: 1D coherent-noise function n(x) [14]

Comparing to other algorithms in a 3-dimensional surface, the main difference of using noise algorithm is how the height value be calculated through different noise functions. Although rendering speed of this algorithm is slower than Diamond-square algorithm, the overall memory consumption is minimal [11]. More importantly, Noise can allow changes to occur gradually and generate pseudorandom height maps for terrains without feeling chaotic.

> *"The basis of almost every procedurally generated landscape is a noise algorithm. Without one, every mountain, valley, rock, and pebble must be crafted by hand. This is certainly possible, but not very feasible for larger maps. Noise creates all this much faster than a human can dream of, and with greater detail" [11]*

As a result, based on the above research of algorithms for procedurally generating terrain, this project will use noise algorithms to help create complex and random terrain. Therefore, it is necessary to explore some of the noise functions and find out which function can be best implemented in this project.

2.2.2 Noise functions

This section looked at the usage of each noise functions

Value Noise

Value noise is the easiest function to understand and implement. It also be considered as regular noise. Basically, on a grid-filled surface, every grid point is assigned a value. Each pixel will randomly select a grid point and interpolates its value. The noise function then returns the interpolated number based on the values of the surrounding grid points. Result of value noise can be shown in Figure 7.

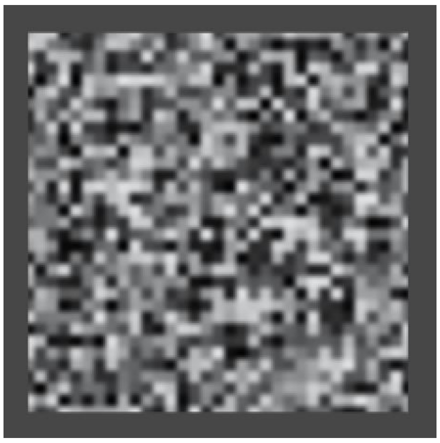


Figure 7: 2D Value Noise [15]

Due to the independency between each pixel, value noise provides various randomness. However, it doesn’t contain natural artifacts for creating terrain since there are lots of cubes and squares.

> *“While Value noise is smooth, it has a blocky appearance. The patterns look random but are clearly constrained to a grid, which is undesirable when trying to create a more chaotic or natural-looking surface.” [15]*

Perlin Noise
Perlin noise is a type of coherent noise, also called gradient noise, which allows changes occur gradually. It is a standard in the noise functions developed by Ken Perlin.

> *“To Ken Perlin for the development of Perlin Noise, a technique used to produce natural appearing textures on computer generated surfaces for motion picture visual effects.*
> *The development of Perlin Noise has allowed computer graphics artists to better represent the complexity of natural phenomena in visual effects for the motion picture industry.” [16]*

Generally, Perlin noise is similar to Value noise except the random values were selected from the gradient rather than a certain point. To be specific, in the same grid-filled surface with same value grid points, each pixel is pushed by a pseudo-random gradient to toward different directions and interpolate a smooth function between those points. To compare with Value noise in Figure 8, instead of using height value, Perlin noise uses a gradient vector at each grid point.

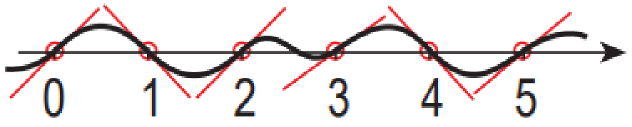


Figure 8: 1D comparation between Perlin noise and Value Noise [17]

As a result, in Figure 9, it is obvious to see that there are more corners and movable space than Value noise. This reflects that Perlin noise not only presents a better natural artifact, but also can generate pseudorandom height maps for terrains without feeling chaotic.

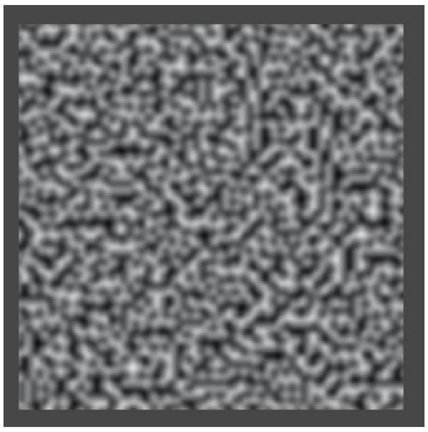


Figure 9: 2D Perlin Noise [15]

Simplex Noise
Although Perlin noise can perfectly create high-quality terrain, it gets slow when generating in higher dimensions (4D, 5D). Hence, the Simplex noise was developed by Ken Perlin again to solve the limitation of noise functions in higher dimensions. Technically, due to the complexity of Simplex is $O(n^2)$ for n dimensions rather than $O(n*2^n)$ of Perlin noise, Simplex noise scales to higher dimensions with much less computational cost.

> *"Simplex Noise solves this by uses simplices instead of hypercubes. A simplex is the shape with the fewest corners in a dimension. In 2D it is a triangle, in 3D a pyramid, and in n-dimensions it is a shape with n + 1 corners." [11]*

By comparing with Perlin noise, the lower computational cost in higher dimensions makes Simplex noise the best quality function. However, the function itself is hard to understand and debug. More importantly, the project in this dissertation mainly focuses on generating terrain in 3D. Higher-dimensional implementation will not be considered.

Whorley noise

> *"Cell Noise, also called Whorley Noise, alone is not very useful for terrain generation. It creates very artificial structures, usually with harsh jagged lines or bubbly surfaces." [11]*

As shown in Figure 10, the main idea of Whorley noise is to simulate textures of stone, water, or biological cells. Basically, it chooses random points on the surface, the value of any given pixel is the distance from d_n to the nth-closest point. The Whorley noise is an interesting noise, but not fit to generate terrain.

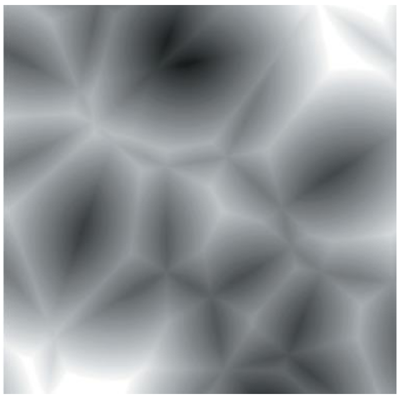


Figure 10: Whorley Noise [18]

Overall, through the above discussion of noise functions. Value noise is the easiest to implement but lacks natural artifacts. While Simplex noise provides the best quality but goes beyond what was originally planned in a 3D world. At last, the final Whorley noise does not apply to terrain generation. Only Perlin noise can effectively render pseudo-random terrain and present a realistic visual effect for modern game industry. Therefore, this project is going to use Perlin noise to analyse height maps and generate complex terrain. In addition, the build-in functions and class of Perlin noise in Unity documentation will also be looked at later below.

2.2.3 Unity Documentation

This section will explore the build-in functions and class in Unity documentation, especially focus on the Perlin Noise function and the usage of Meshes in Unity.

As the main development environment of this project is based on Unity Engine, it is important to understand the how Unity manages those features. In *Unity documentation: Graphics and Scripting API* (Figure 11), it not only contains many explanations of build-in functions and class in Unity but also provides lots of examples, ranging from Mathf.PerlinNoise to Mesh API and from Textures to Shaders.

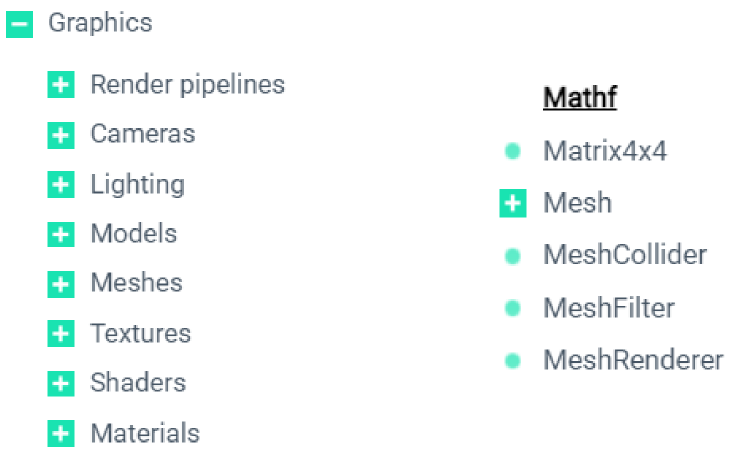
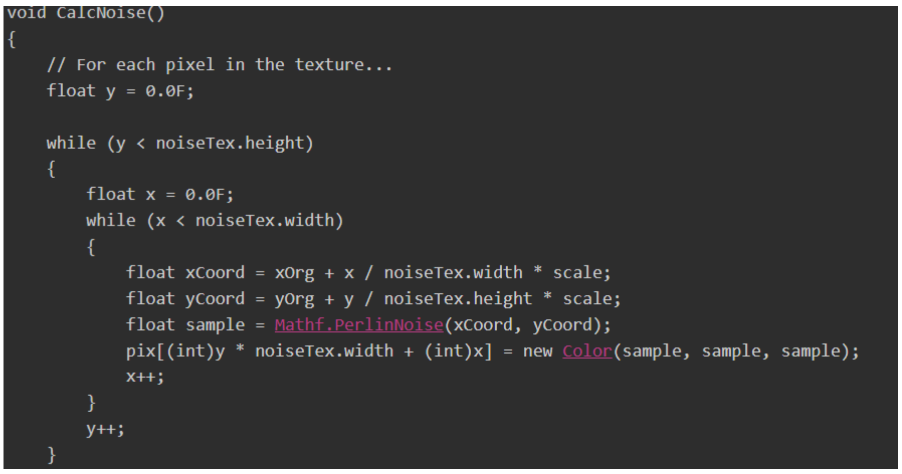


Figure 11: Unity documentation: Graphics (Left) and Scripting API (Right) [19]

Through the document, it provides a clear understanding on the implementation of the key features in terrain generation. For example, in Figure 12, the source code of PerlinNoise function helps to generate random 2D height maps by adjusting its octaves, lacunarity and persistence.



```
void CalcNoise()
{
    // For each pixel in the texture...
    float y = 0.0F;

    while (y < noiseTex.height)
    {
        float x = 0.0F;
        while (x < noiseTex.width)
        {
            float xCoord = xOrg + x / noiseTex.width * scale;
            float yCoord = yOrg + y / noiseTex.height * scale;
            float sample = Mathf.PerlinNoise(xCoord, yCoord);
            pix[(int)y * noiseTex.width + (int)x] = new Color(sample, sample, sample);
            x++;
        }
        y++;
    }
```

Figure 12: Source code of Mathf.PerlinNoise [20]

In addition, the Mesh class in this document is also what the project intends to implement. It has made a clear structure about how to fill the height map with multiple triangles and set the height values of individual vertices to generate complex terrains.

> *"The Mesh class is the basic script interface to an object's mesh geometry. It uses arrays to represent the triangles, vertex positions, normals and texture coordinates and also supplies a number of other useful properties and functions to assist mesh generation." [21]*

Moreover, the LOD scheme in Unity helps to change the LOD dynamically based on how far away the terrain chunks are from the player.

> *"The LOD technique allows Unity to reduce the number of triangles it renders for a GameObject based on its distance from the Camera. To use it, a GameObject must have a number of meshes with decreasing levels of detail in its geometry. These meshes are called LOD levels. The farther a GameObject is from the Camera, the lower-detail LOD level Unity renders. This technique reduces the load on the hardware for these distant GameObjects and can therefore improve rendering performance." [22]*

As this project will implement endless terrain, large maps will make up multiple meshes. Hence, it is necessary to measure the speed and consumption when rendering complex terrain. The usage of LOD in Unity provides some idea for the project about how to achieve fast rendering in the terrain generation.

2.3 Summary

With the development of technology, computer performance has reached a new level in recent years. Its processing power and memory consumption became cheaper and faster. As a result, modern game development is no longer limited by hardware, but instead using different technologies and tools to find ways to create a virtual world as complex and varied as in reality. Hence, procedural terrain generation is one of the powerful technologies for generating realistic landscape.

In general, based on the above conceptual understanding of PCG, the exploration of terrain generation algorithm, the analysis of various noise functions and the technical explanation of Unity features, using Perlin noise to create pseudorandom height maps at run-time and fill them with meshes in Unity is the best way to solve the technical challenge.

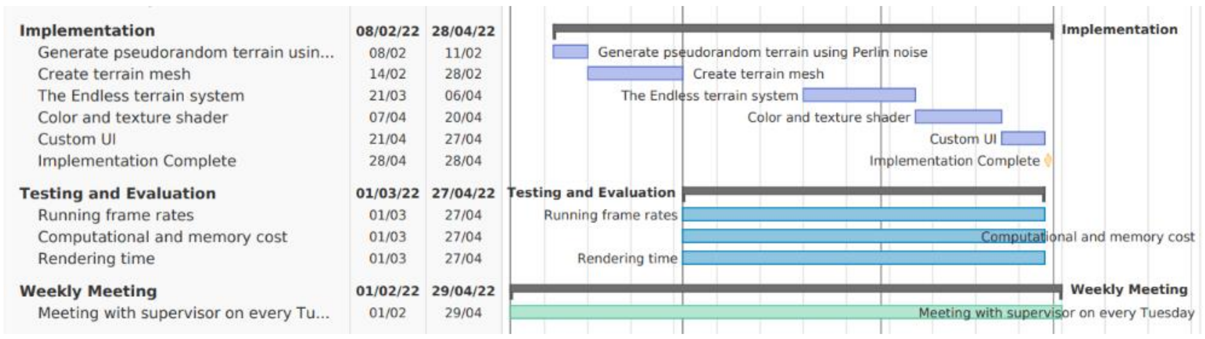
Chapter 3: Method/Methodology

3.1 Overview/Introduction

This section details what work was done, how it was completed and why. It also covers various aspects of project development, including the project plan, the tools been used and the outline of project functionality. Eventually, the final implementation will be explained as well as the methods and techniques used to solve the technical challenges.

3.2 Planning/design

This project intends to implement a procedural terrain generation in Unity engine, which will allow user to automatically design terrain of any random shapes, different textures, and infinite size. Based on the project objectives in Chapter 1, the main idea is to create pseudorandom content by computer instead of human. Thus, the first step of this project will focus on generating pseudorandom height maps for terrains through Perlin Noise and fill those height maps with multiple triangles and vertices. Then the endless terrain system will be implemented with a texture shader for the same visual effects as the designer did. At last, a custom UI will also be implemented to help user easily adjust various features through procedural terrain generation.



Implementation	**08/02/22**	**28/04/22**
Generate pseudorandom terrain usin...	08/02	11/02
Create terrain mesh	14/02	28/02
The Endless terrain system	21/03	06/04
Color and texture shader	07/04	20/04
Custom UI	21/04	27/04
Implementation Complete	28/04	28/04
Testing and Evaluation	**01/03/22**	**27/04/22**
Running frame rates	01/03	27/04
Computational and memory cost	01/03	27/04
Rendering time	01/03	27/04
Weekly Meeting	**01/02/22**	**29/04/22**
Meeting with supervisor on every Tu...	01/02	29/04

Figure 13: Waterfall development process in Gantt chart [24]

This is Gantt chart for this project plan which displays the key tasks throughout the development process from February to April. As shown in Figure 13, the waterfall development process was perfectly designed for this project. [22] The chart consists of three core stages with different sub-tasks, including Implementation, Testing and Evaluation, and Weekly meeting. Each stage has a clearly defined starting time and conclusion, which makes implementation easy to monitor. In Implementation stage, it contains all the development of functionalities for procedural terrain generation. During the meeting stage, if there are some problems in this project, it is easy to ask feedback from supervisor to improve the implementation. Lastly, the Testing and Evaluation stage is already detailed in the technical specification of the requirements phase, which makes the testing process easier and more transparent.

3.3 Tools

This section will introduce the tool and IDE used throughout the whole project.

Unity Engine

As modern game development technology continues to grow, more and more game engines are introduced to developer for creating various types of games, such as Unity engine, one of the most popular engines in the game industry. Hence, this project is developed in C# by Unity engine for good reasons.

First, the multiplatform and portability. Unity engine is free for everyone and can support over 25 platforms ranging from Mac OS to Windows, from Android and iOS to consoles and web. Developer also can have the opportunity to create not only 2D and 3D contents but also allow the creation in VR and AR. [25] Therefore, due to a large group of users, it is easy to find the resources and solutions from the community when facing some technical issues during the development. More importantly, the implementation of procedural terrain generation from this project can be widely built and expanded into various game areas in the future.

Secondly, the graphic. Unity engine is known for high-quality graphics and visual effects. Its highly customizable rendering technologies as well as a variety of intuitive tools, make Unity engine easy to create more realistic games. [26] As a result, many of the terrain features explored from background research in Chapter 2 can be implemented agilely and efficiently in Unity.

Lastly, the powerful scripting. Object behaviour in Unity is not limited to the built-in modules. Conversely, Unity supports complex operations defined in different programming languages, such as C#. In addition, these languages can be used simultaneously in the project for people with different technical backgrounds. Moreover, all of programming languages are written in scripted form, which allow for fast compilation, iteration, and flexibility. [27] Hence, since C# is the main programming language in this project, it is easy to track and debug in Unity if there are some problems.

Visual Studio

The reason for choosing Visual Studio as IDE in this project is quite simple. Firstly, the C# language can be compiled directly in Visual Studio, and it is highly compatible with unity. Secondly Visual Studio supports integration with GitHub. It allows this project to be easily backed up to online code repository by executing pull/push requests.

3.4 Functionality Outline

This section lists the all the functional requirement for this project.

Functionality one: Adjustable random terrain

- Users are free to change any style of terrain they want.
- All the changes will be automatically generated.

Functionality two: Fully customized texture shader

- Users can apply or add any colour and textures for different terrain types.
- All the changes will be automatically generated.

Functionality three: Infinite world

- Endless terrain can be automatically generated at run time when users are moving around the map

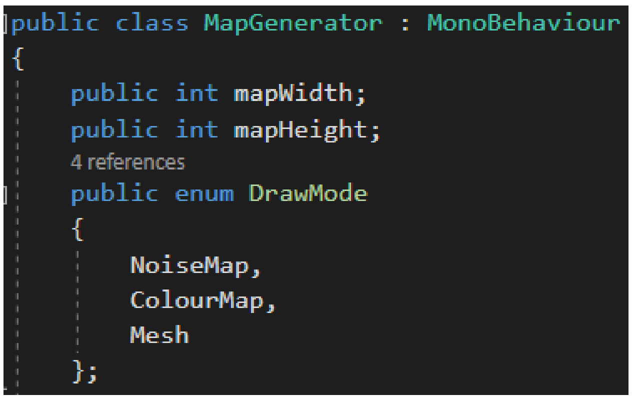
3.5 Development/Implementation

3.5.1 Introduction

This section will detailly explain how procedural terrain generation deploy the concepts into development.

3.5.2 pseudorandom terrain map

Firstly, as stated earlier, before filling the terrain with meshes, using Perlin noise to analyse height maps is the best way to generate pseudorandom terrain without feeling chaotic. To achieve this technique, the previous research of Unity documentation provides a build-in *Mathf.PerlinNoise* function, which can easily solve this technical challenge. Hence, the initial implementation of this project will be split in two parts in MapGenerator class, Noise map and Colour map. The initial size of all the map is also controlled by variables *mapWidth* and *mapHeight.* See in Code 1.



```
public class MapGenerator : MonoBehaviour
{
    public int mapWidth;
    public int mapHeight;
    4 references
    public enum DrawMode
    {
        NoiseMap,
        ColourMap,
        Mesh
    };
```

Code 1: MapGenerator class

The *NoiseMap* here creates a grid of Perlin noise values and render them to the screen as a texture. While in the *ColourMap*, it signs colour to the specific height values and generates different terrain types from noise. Lastly, the entire terrain maps will be filled with multiple triangles and vertices in *Mesh.*

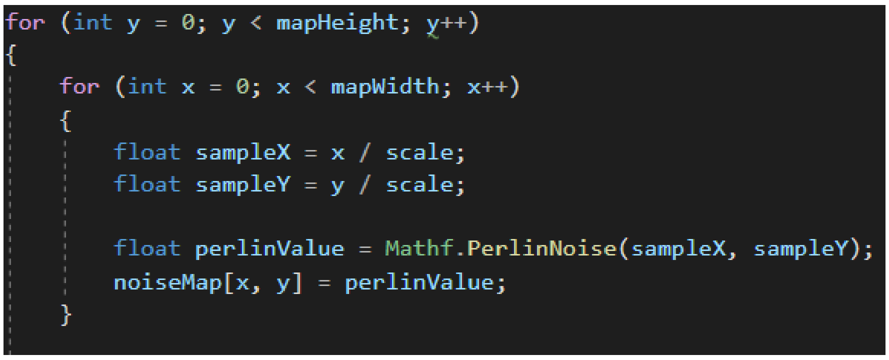
3.5.2.1 Noise map

Perlin Noise

Previous background research on Perlin Noise concluded that it was efficient and easy to implement. This noise function is well documented partly because it is widely used in procedural generation and is the first noise function to use gradient-value techniques. Therefore, Unity provides a built-in implementation in the maths class called *Mathf.PerlinNoise*, in which shown in Chapter 2: Unity documentation.

‘*public static float Mathf.PerlinNoise(float x, float y);*’ [20]

The main idea of this method is to generate a noise by returning a grid of values between 0 and 1 in a 2D array of float values, and then sampling the height values to any points on the plane by passing the appropriate X and Y coordinates. However, the problem is if the noise value is taken in integral coordinates, the same sample value will always be returned. This situation shouldn't be considered as randomness. To avoid repetition, the scale of the noise can be divided to get non-integer values at each coordinate. Full solution seen in Code 2.



```
for (int y = 0; y < mapHeight; y++)
{
    for (int x = 0; x < mapWidth; x++)
    {
        float sampleX = x / scale;
        float sampleY = y / scale;

        float perlinValue = Mathf.PerlinNoise(sampleX, sampleY);
        noiseMap[x, y] = perlinValue;
    }
```

Code 2: Initial Implementation of Perlin Noise in Unity

As a result, the noise map created by Perlin noise function will be converted to texture and applied on a 2D plane through TextureGenerator class. Then, each pixel is interpolated with a random point. In Figure 14, a perfect Perlin noise map is generated.

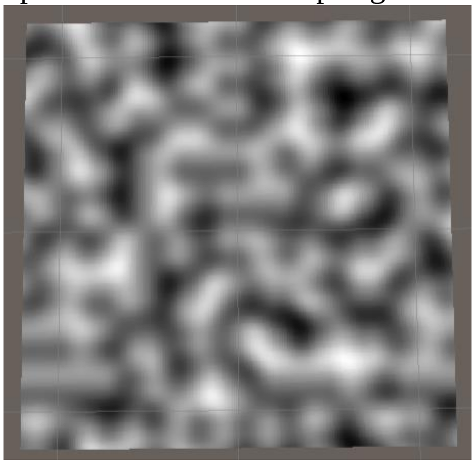


Figure 14: 2D Perlin noise at 25 scale

Although a scaleable noise map can now be created accurately, the noise pattern looks too smooth and cannot be changed randomly. This goes against the main purpose of a natural-looking pseudorandom terrain. Therefore, to solve the issue, the following functions will explain how to add layers of multiple levels of Perlin noise to enhance its details and diversity.

Octaves

As mentioned above in Chapter 2, Perlin noise is a type of coherent noise. It is the sum of several coherent noise functions that increase in frequency with the X-axis and decrease in amplitude with the Y-axis. Each coherent-noise function that is part of a Perlin noise function is commonly called Octaves. With the number of different octaves, the project can simply control the amount of detail of Perlin noise map. The more octaves increase, the more details were added. Examples of comparison can be presented in 1D Perlin noise function n(x) in Figure 15.

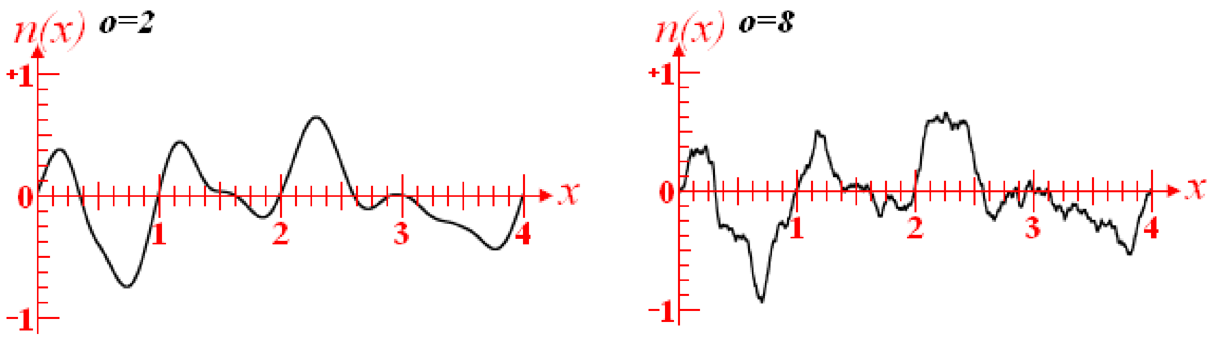
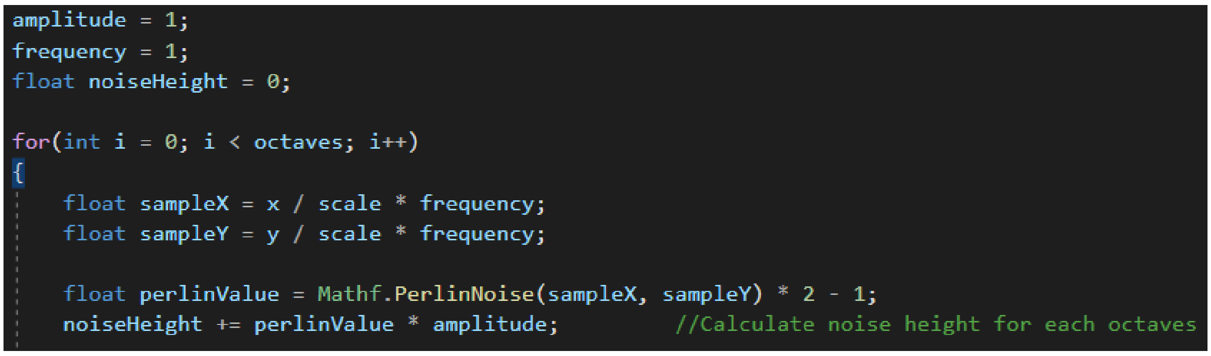


Figure 15: Left is when Octave equals 2, Right is when Octave equals 8 [14]

To be specific, as shown in Code 3, instead of setting the Perlin values directly to all the points in noise map, the height values of each octave should be signed to the noise map by adjusting its amplitude and frequency on sample x-y coordinates between value of -1 and 1. Therefore, with multiple octaves, it provides more natural objects and makes the noise map change more rapidly.



```
amplitude = 1;
frequency = 1;
float noiseHeight = 0;

for(int i = 0; i < octaves; i++)
{
    float sampleX = x / scale * frequency;
    float sampleY = y / scale * frequency;

    float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
    noiseHeight += perlinValue * amplitude;         //Calculate noise height for each octaves
```

Code 3: Apply octaves into noise map

However, octave only refers as the individual layers of Perlin noise in this function. To add more details and effects in each subsequent octave, the variables lacunarity and persistence also need to be implemented.

Lacunarity
Lacunarity generally controls the new frequency increases for each successive octave in Perlin noise function. It can be calculated from previous octave’s frequency and the lacunarity value. The formula is as follows.

The frequency of octave = previous frequency * lacunarity.

As each octave increases in detail through lacunarity, its influence on some of the features should diminish. For example, when generating a mountain, the smaller the rock, the less effect it occurs on the outline of the mountain. This is because the lacunarity essentially affect the value of x-axis. Thus, the comparison results are shown in Figure 16. With the same number of scales (S), octaves (O), and persistence (P) on the noise map, increasing the lacunarity (L) values of octave will increase the detail of small features.

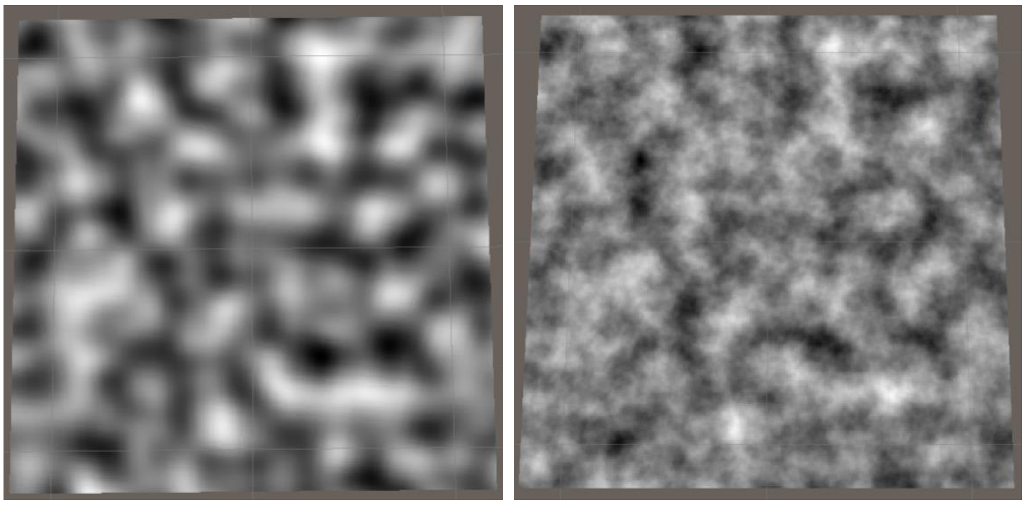


Figure 16: Under the same values with S=25, O=5, P=0.5. Left is when L=1. Right is when L=2.

Persistence
The persistence has the similar property to lacunarity. The main difference is persistence controls the amplitudes decrease for each successive octave in Perlin noise function. It can also be calculated from previous octave's amplitudes and the persistence value. The formula is as follows.

The amplitude of octave = previous amplitude * persistence

Using a persistence value with the range between 0 and 1, the function affects how rapidly the amplitude decreases with each octave. Therefore, the comparison results are shown in Figure 17. With the same number of scales (S), octaves (O), and lacunarity (L) on the noise map, increasing the persistence value (P) will produces rougher noise map and affects how much the small features influence the overall shape of the map. Eventually, the map will become sharp and crispy before the value is out of range.

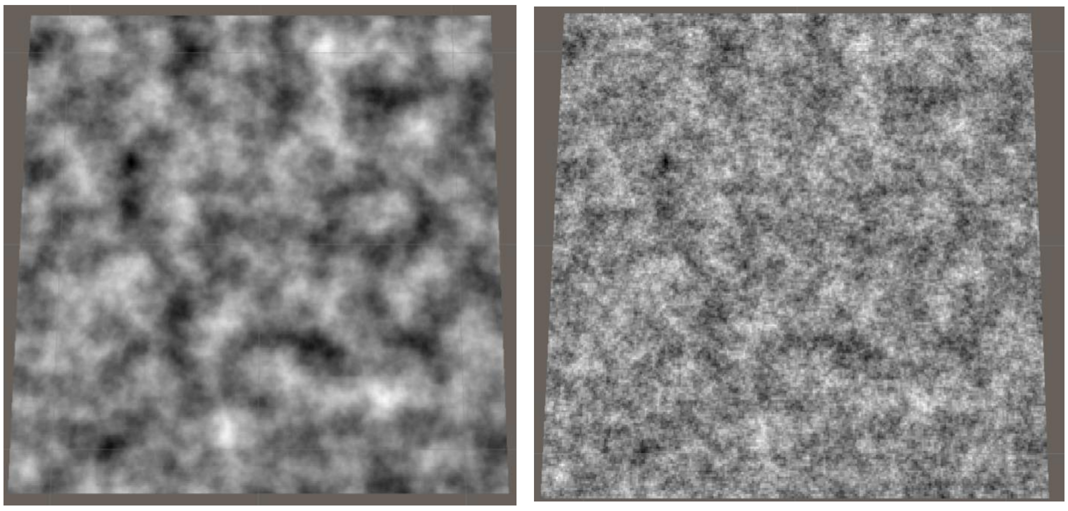
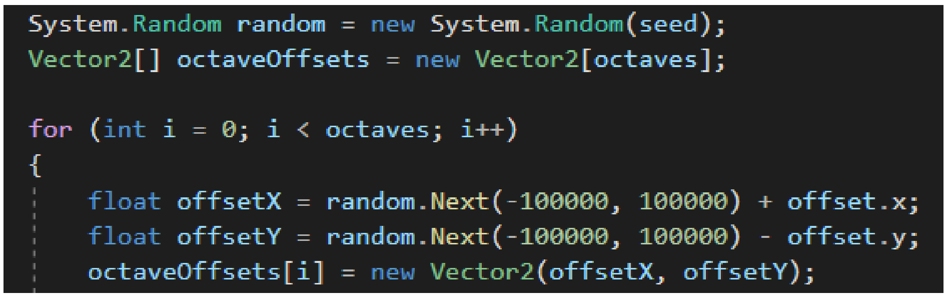


Figure 17: Under the same values with S=25, O=5, L=2. Left is when P=0.5. Right is when P=1

Seed

Seed is a value that changes the output of a coherent noise function. It provides the functionality to automatically generate many pseudorandom noise maps. As shown in Code 4, the key implementation technique is sampling each octave radically from different coordinate location with the range between -100000 and 100000.



```
System.Random random = new System.Random(seed);
Vector2[] octaveOffsets = new Vector2[octaves];

for (int i = 0; i < octaves; i++)
{
    float offsetX = random.Next(-100000, 100000) + offset.x;
    float offsetY = random.Next(-100000, 100000) - offset.y;
    octaveOffsets[i] = new Vector2(offsetX, offsetY);
```

Code 4: Implementation of pseudorandom generator for noise map

In contrast to the results shown in Figure 18, although all the map values remain the same, the structure of noise map was created differently as the seeds changes in Perlin noise function.

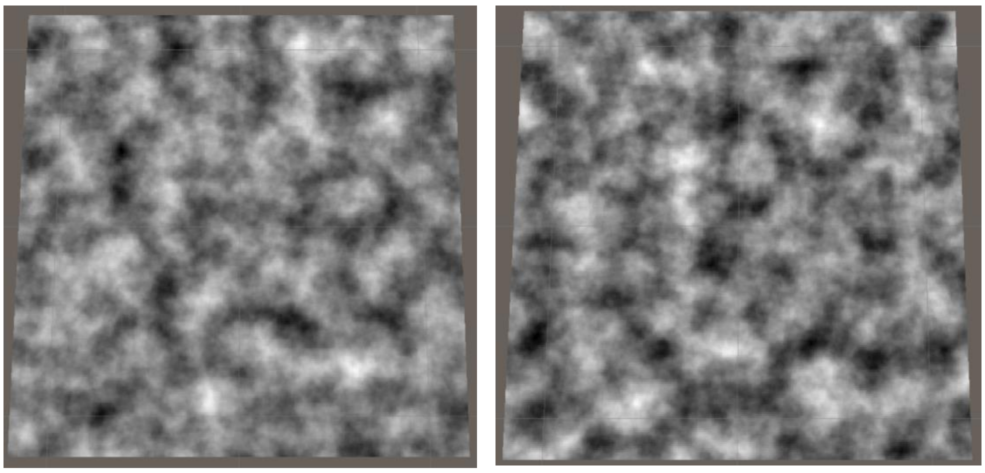
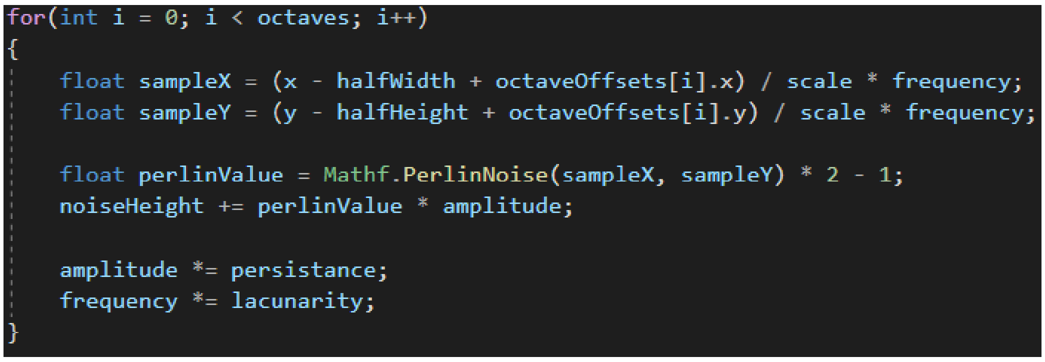


Figure 18: Under the same value with S=25, O=5, L=2, P=0.5. Left is when Seed=1. Right is when Seed=2.

Overall, by combining the lacunarity and persistence of multiple octaves, the NoiseMap method can generate a far more natural and pseudorandom height map from Perlin noise. Based on all the analysis above, the final core implementation is shown in Code 5.

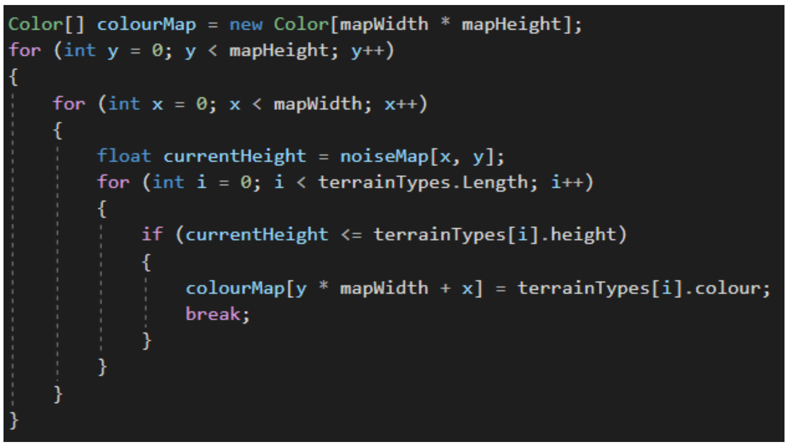


```
for(int i = 0; i < octaves; i++)
{
    float sampleX = (x - halfWidth + octaveOffsets[i].x) / scale * frequency;
    float sampleY = (y - halfHeight + octaveOffsets[i].y) / scale * frequency;

    float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
    noiseHeight += perlinValue * amplitude;

    amplitude *= persistance;
    frequency *= lacunarity;
}
```

Code 5: Final Implementation of Perlin Noise in Unity

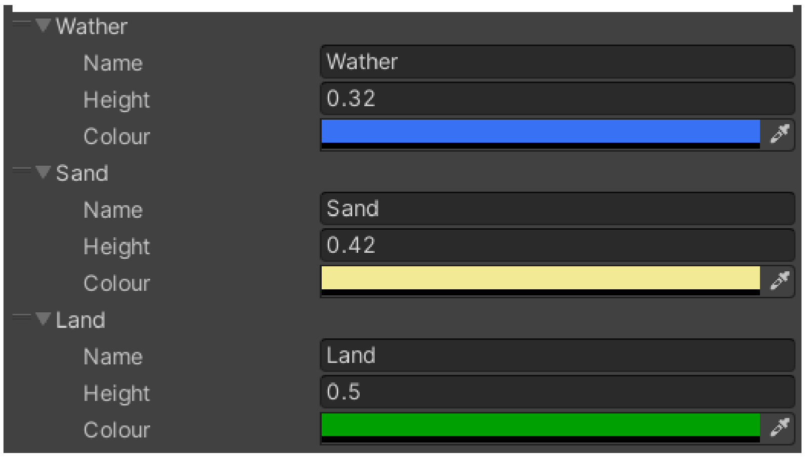
3.5.2.2 Colour map

Since the height maps are successfully analysed by Perlin noise function, it is necessary to implement initial functionality for the division of different terrain types. The main idea of this method is to simply design different terrain types by assigning colours to specific height values. As shown in Code 6, by looping through the height map from Perlin noise, the height of the current map will be given values from coordinates x-y, followed by a comparison of the terrain heights designed from the editor, in which shown in Figure 19. If the current height value is equal to or less than the height of these terrains, each type of terrain will be set to its corresponding height. Lastly, the colour of each point will be stored in different terrain through a 2D array.



```
Color[] colourMap = new Color[mapWidth * mapHeight];
for (int y = 0; y < mapHeight; y++)
{
    for (int x = 0; x < mapWidth; x++)
    {
        float currentHeight = noiseMap[x, y];
        for (int i = 0; i < terrainTypes.Length; i++)
        {
            if (currentHeight <= terrainTypes[i].height)
            {
                colourMap[y * mapWidth + x] = terrainTypes[i].colour;
                break;
            }
        }
    }
}
```

Code 6: Colour map method



▼ Wather	
Name	Wather
Height	0.32
Colour	🟦
▼ Sand	
Name	Sand
Height	0.42
Colour	🟨
▼ Land	
Name	Land
Height	0.5
Colour	🟩

Figure 19: Editor page for setting terrain and colour

The initial Colour map method provides guidance for the implementation of complex shader in the future development. When an endless terrain system is implemented later, it will be faster to determine the size of the chunk directly rather than measuring the height by the width and height of height maps. As a result, by combing Noise map and Colour map methods, a pseudorandom terrain map is successfully generated. The example results are shown below in Figure 20.

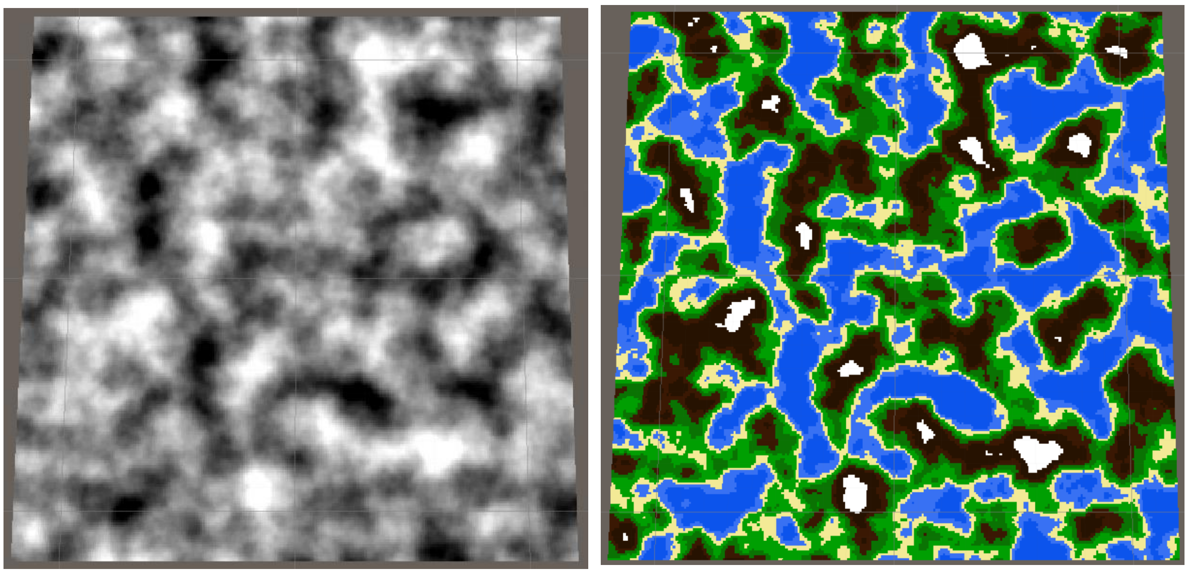
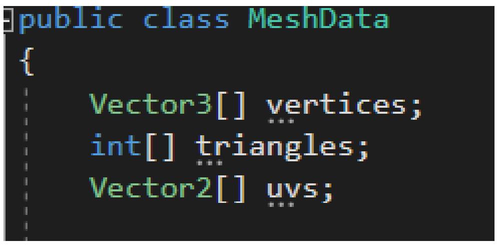


Figure 20: Left is height map from Perlin noise. Right is its early stage of terrain map

3.5.3 Terrain Mesh Generator

Currently, the procedural terrain generation only allows to create a 2D plane map. To generate a 3D terrain, the mesh generator is an essential technique in this project. As stated in Objective 2, the main concept of this method is to fill the entire terrain map with meshes and set the heights of individual vertices. According to the exploration in Chapter 2, the Mesh class in Unity can easily modify meshes. To be specific, meshes in Unity contain multiple vertices and triangle arrays. Each triangle is made by three indices and each index is an array of vertices. In addition, for every vertex, it can also be set to normal or UV. Then, by using a MeshData struct in Code 7, the mesh generator will be able to access, process and create Meshes based on those geometry data.


```
public class MeshData
{
    Vector3[] vertices;
    int[] triangles;
    Vector2[] uvs;
```
Code 7: MeshData struct in MeshGenerator

Hence, the calculation of each data must be explained detailly.

Setting Vertices
Before adding triangles to a grid, the generator needs to know how many triangles are created. The best way to calculate the amount is to first know the number of vertices of each triangle. As shown in Figure 21, each square grid has two triangles and each consisting of three vertices. The overlapping vertices will be ignored.

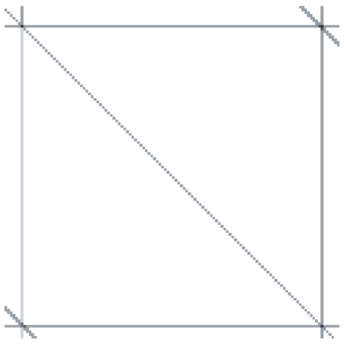
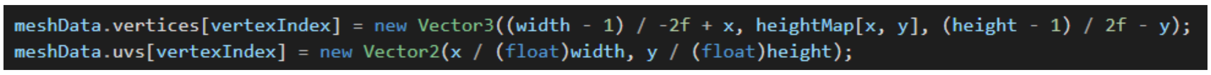


Figure 21: square grid is created by two triangles

Thus, the calculation can be done as follows.

Vertices = width of mesh * height of mesh

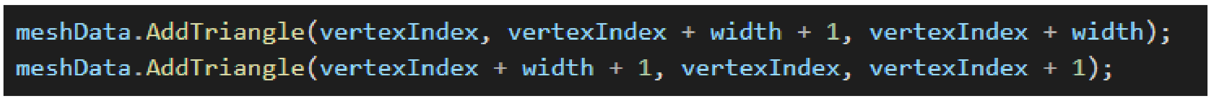
With this formular, a vertex array will be assigned in the generator to control the number of vertices and place them as index to the meshes. However, one problem here is that all the vertices are added from the edge rather than center. To solve this, all the points on the x and z axis will have to offset by -1/2 and 1/2 separately. Therefore, the full solution of setting vertices is shown in Code 8, which can also be used for adding UVs.


```
meshData.vertices[vertexIndex] = new Vector3((width - 1) / -2f + x, heightMap[x, y], (height - 1) / 2f - y);
meshData.uvs[vertexIndex] = new Vector2(x / (float)width, y / (float)height);
```
Code 8: Setting vertices and UVs to the mesh

Adding Triangles
Since the vertices are placed perfectly on the meshes and each square grid contains two triangles with total six vertices, the length of triangle array can be calculated as follows.

Triangle = (width of mesh – 1) * (height of mesh - 1) * 6

When creating the triangles with this formula, each row will have one more vertex than the square grid. It is necessary to increase the vertex index after each row and square. To be specific, if the indices of vertices are referred as i, i+1 means the next index. Then the next row of vertex index will start at i+width of the mesh and end at i+width of the mesh+1. Hence, as shown in Code 9, the generator will be able to add triangles to the meshes through these vertex indices.


```
meshData.AddTriangle(vertexIndex, vertexIndex + width + 1, vertexIndex + width);
meshData.AddTriangle(vertexIndex + width + 1, vertexIndex, vertexIndex + 1);
```
Code 9: Setting triangles to the mesh in clockwise

As a result, in Figure 22, the entire terrain map is filled with vertices and triangles by the mesh generator.

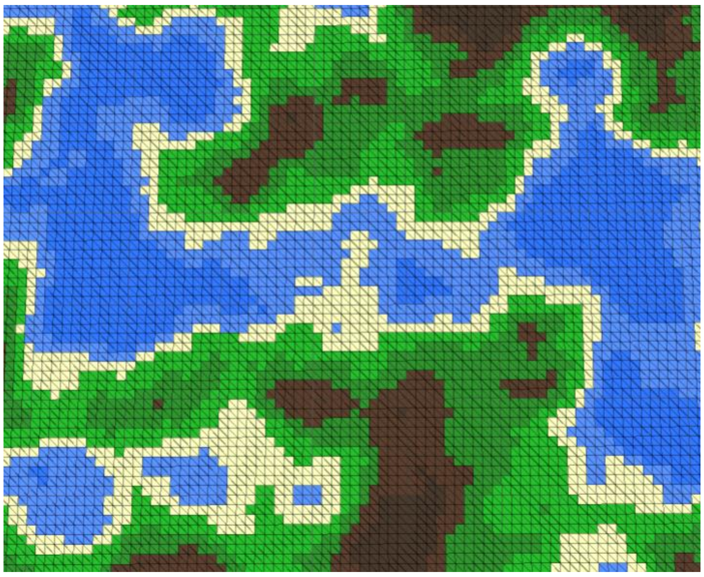


Figure 22: 2D terrain mesh

Then, by changing the height values on y-axis of each vertex index through a multiplier, a 3D terrain mesh can be simply generated. Seen in Figure 23.

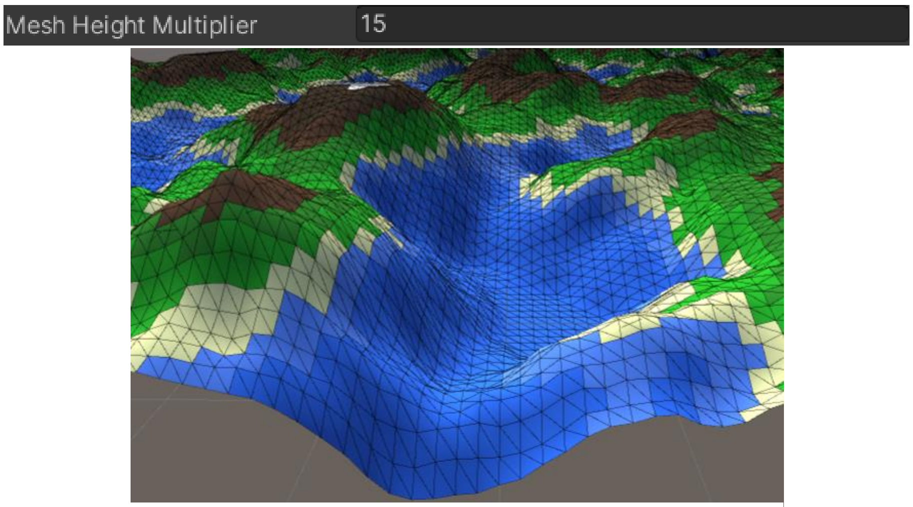
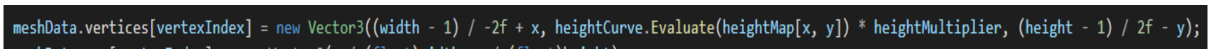

Mesh Height Multiplier 15

Figure 23: 3D terrain mesh at height of 15

Although there are some shape changes on the current terrain map, different terrain types should apply with different height values for more natural looking. For example, water and sand should be as parallel to the horizon as possible while as the plains should be flatter than mountains. To improve this, the height map curve needs to be introduced.

Height map curve

The main idea of using height map curve in mesh generator is to specify the effect of different height values on multiplier. As the value in the height map is always a float number between 0 and 1, each point can be multiplied by a curve to create more realistic terrain. In Unity, the AnimationCurve class is commonly used to smooth the changes of the keyframe at index. [28] To be specific, by using the height value on x-axis from the terrain map, the mesh generator will be able to get the corresponding y-value from the height curve. Seen in Code 10.



```
meshData.vertices[vertexIndex] = new Vector3((width - 1) / -2f + x, heightCurve.Evaluate(heightMap[x, y]) * heightMultiplier, (height - 1) / 2f - y);
```

Code 10: Setting height map curve

The result of multiplying these values will not only remain the same height, but also can lower the already low points. If the value of multiplier is changed in the Editor, the height value will also be assigned to the highest point. Therefore, the curve can be adjusted to form the desired look by adding some keys. The example result is shown in Figure 24.

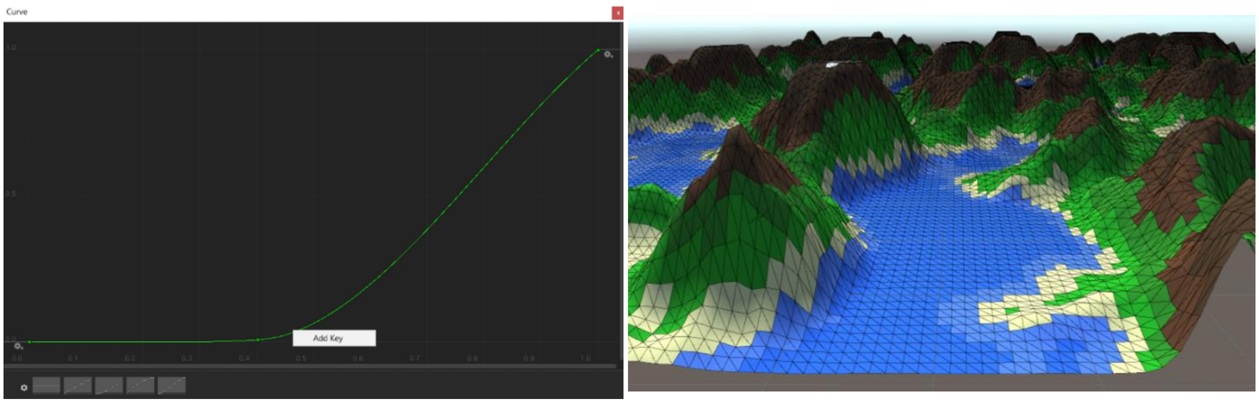


Figure 24: Left is the height map curve. Right is a 3D terrain mesh affected by the curve.

Comparing to the Figure 23 above, the terrain in Figure 24 highlights the steepness of the mountains while the water and sand are quite flat. As a result, the terrain has become more realistic now.

3.5.4 Endless terrain system

As a single terrain mesh can be generated perfectly, the endless terrain system will be the next key requirement of this project. Seen in Objective 3

> *"Understand the usage of Level of Detail (LOD) and implement the endless terrain system for terrain generation."*

The main inspiration for implementing the endless terrain system essentially came from a famous game called Minecraft. The infinite overworld in Minecraft is generated by using chunk system, which divides the map into manageable pieces. Each chunk is 16 blocks wide, 16 block long and 384 blocks high. [29] Overall, all these chunks are generated around player and new chunks will be formed as players move around the map. To prevent the game from overloading and freezing while it is running, Minecraft will not process the unloaded chunks and lower the game aspects when loaded chunks are far away from players. [29] The example is shown in Figure 25.

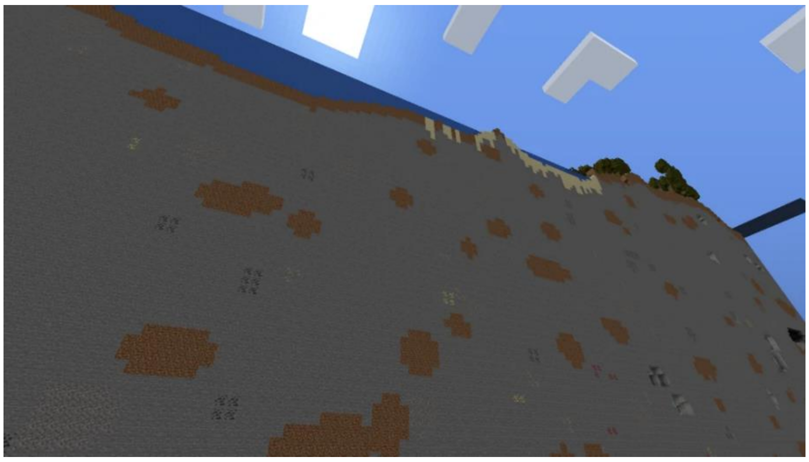


Figure 25: Boundary between loaded chunk and unloaded chunk [29]

Currently, the terrain generation in this project uses lots of meshes to render topography. If the chunk system is developed, it may consume more resources and memory to generate terrain. Eventually, the performance of the game will be affected. Therefore, before implementing the chunk system and rendering large terrain meshes, the usage of level of detail (LOD) needs to be introduced.

3.5.4.1Level of detail (LOD)

Based on the previous research in Chapter 2 about the LOD scheme in Unity, it is a technique reduces multiple mesh resolutions in certain distance. As a player moves away from the chunks, terrain meshes will be rendered with fewer vertices and triangles to keep the game running smoothly. In other words, a terrain should have several meshes with decreasing levels of detail in its geometry. With Figure 26 as reference, the idea of implementing LOD is similar to a standard binary hierarchy tree. As the n levels in the trees are increasing, the total number of the root nodes m in each level will be 2^n and the number of comparisons between each level will be 2^n-1. The more root nodes are counted, the more binary tree grows, which implies the different levels of details.

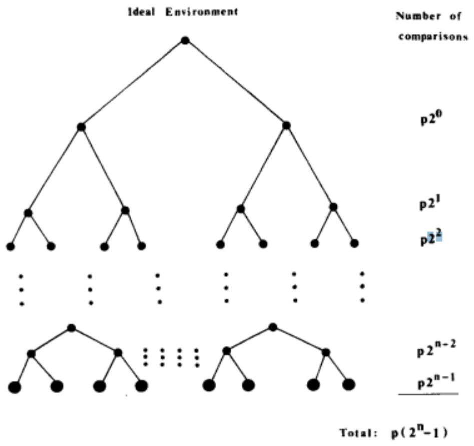


Figure 26: Ideal binary tree [30]

To be specific, each level of detail will be considered as L. As shown in Figure 27, the number of vertices in each row at the original level (LOD start at L= 0) is 5. Unlike binary hierarchy calculation, LOD in this project is designed to achieve fast rendering by reducing the detail in every mesh. In other words, lower levels represent higher resolution. Therefore, the number of comparisons per level will be 2^L. This means the number of vertices in next level of detail (L=1) is 3, which is the black circles in Figure 28.

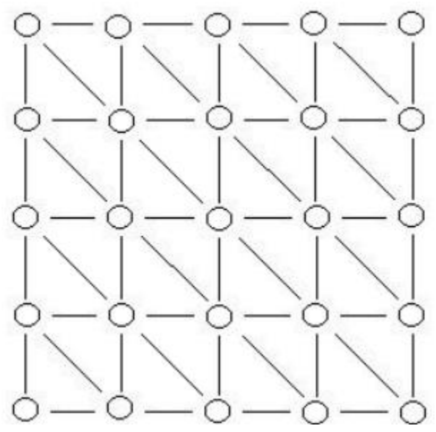


Figure 27: Mesh in LOD = 0. The white circles represent vertices (also can be referred as the width and height of the mesh) [31]

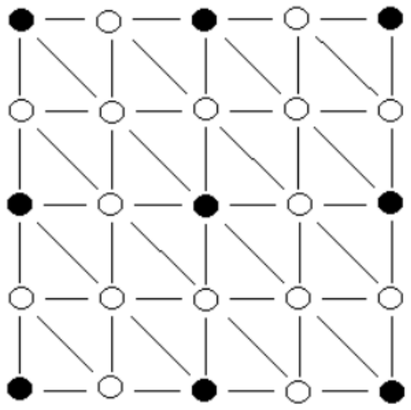


Figure 28: Mesh in LOD = 1. The black circles represent new vertices in LOD1. [31]

Thus, under the new level of details, the number of vertices in new meshes can be calculated as follow. V represents the number of vertices per row. L represents the level of detail.

$$V = (V - 1) / 2^L + 1$$

As a result, in Figure 29, when the LOD increases, the number of vertices and triangles in the meshes decreases. The complex terrain becomes rough and simple.

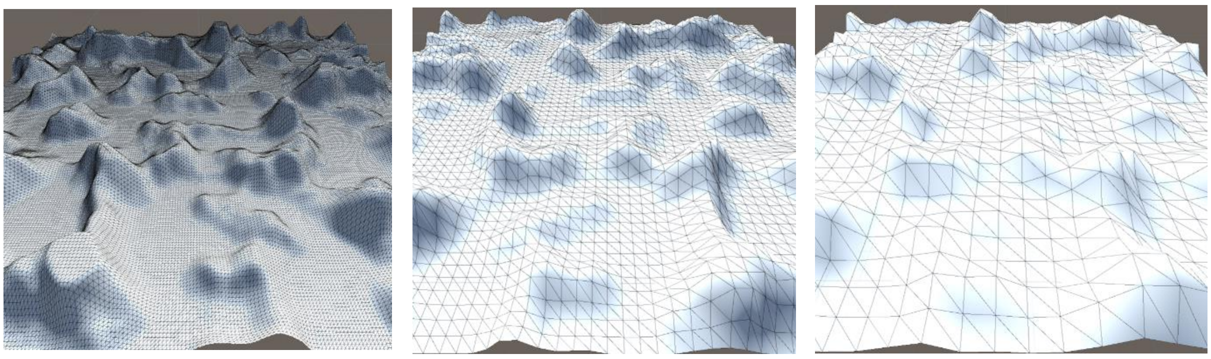
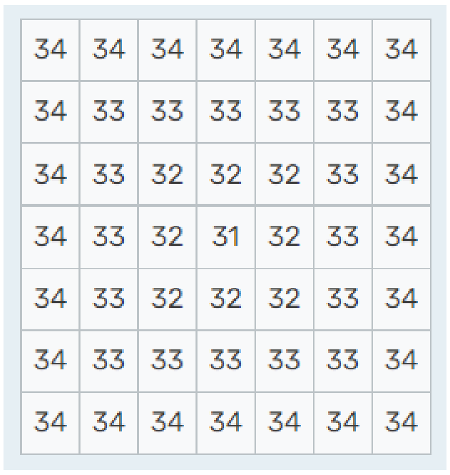


Figure 29: Left is LOD = 0. Middle is LOD = 2. Right is LOD = 4.

3.5.4.2 Chunk system
With the implementation of level of detail, the procedural terrain generation in this project will be able to easily render large terrain chunks with multiple meshes in distance. As stated earlier, Minecraft also use levels to determine what load type the chunk is. In Minecraft, there are mainly four types of chunks, and each chunk has different level and properties [29]. Seen in Figure 30.

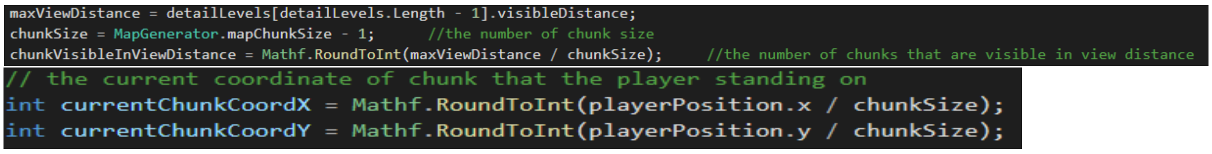


34	34	34	34	34	34	34
34	33	33	33	33	33	34
34	33	32	32	32	33	34
34	33	32	31	32	33	34
34	33	32	32	32	33	34
34	33	33	33	33	33	34
34	34	34	34	34	34	34

Figure 30: Level propagation in Minecraft [29]

Generally, all the game details will be generated at level 31 and below while as other level from 32 to 34 will decrease the aspects rapidly until the maximum of 44. The player usually starts in the middle of this layer and the surrounding levels, such as level 32 or level 33, are generated around the player according to its rendering distance.

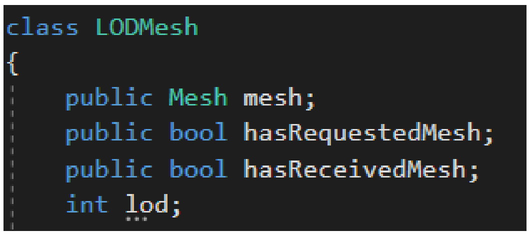
Hence, the basic idea of implementing chunk system in this project is also similar to Minecraft. The visible chucks will only spawn around the player based on the view of distance. Note that regardless of which terrain chunk the player is placed on, the chunk in visible distance simply refers to the terrain meshes that are generated around the player at run time. As soon as these chucks are out of the visible distance, they will be unloaded by the generator and the new visible chunks will be automatically formed again based on the distance from the closest point in its parameter to player's coordinates. The main calculation of all these variables is shown in Code 11.



```
maxViewDistance = detailLevels[detailLevels.Length - 1].visibleDistance;
chunkSize = MapGenerator.mapChunkSize - 1;      //the number of chunk size
chunkVisibleInViewDistance = Mathf.RoundToInt(maxViewDistance / chunkSize);     //the number of chunks that are visible in view distance
// the current coordinate of chunk that the player standing on
int currentChunkCoordX = Mathf.RoundToInt(playerPosition.x / chunkSize);
int currentChunkCoordY = Mathf.RoundToInt(playerPosition.y / chunkSize);
```

Code 11: Calculation of chunk size, visible distance, and chunk coordinate

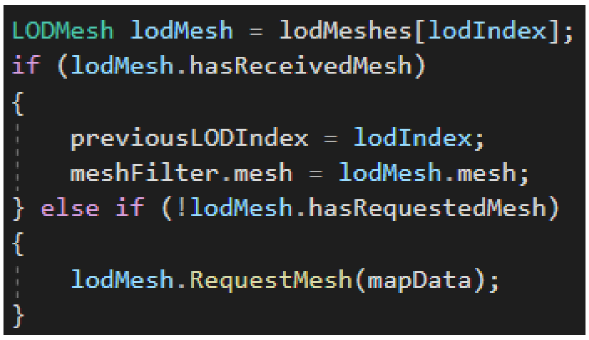
In addition, to improve optimization, the level of detail will also increase dynamically as chunks move far away from the player. To be specific, the level of details from each mesh chunk can be first stored in the generator sequentially through the LODMesh class. Seen in Code 12.



```
class LODMesh
{
    public Mesh mesh;
    public bool hasRequestedMesh;
    public bool hasReceivedMesh;
    int lod;
```

Code 12: LODMesh class for storing mesh and LOD data

Then, in Code 13. By comparing the distance to the player's nearest edge with the distance to each detail level, the generator will be able to request mesh with the specific level of detail.



```
LODMesh lodMesh = lodMeshes[lodIndex];
if (lodMesh.hasReceivedMesh)
{
    previousLODIndex = lodIndex;
    meshFilter.mesh = lodMesh.mesh;
} else if (!lodMesh.hasRequestedMesh)
{
    lodMesh.RequestMesh(mapData);
}
```

Code 13: Updating the terrain chunks through different LOD in specific visible distance

As a result, the endless terrain system is implemented perfectly as shown in Figure 31. Different kinds of terrain maps can be automatically created when player moves around. For more realistic looking topography, texture shader will be the next priority techniques.

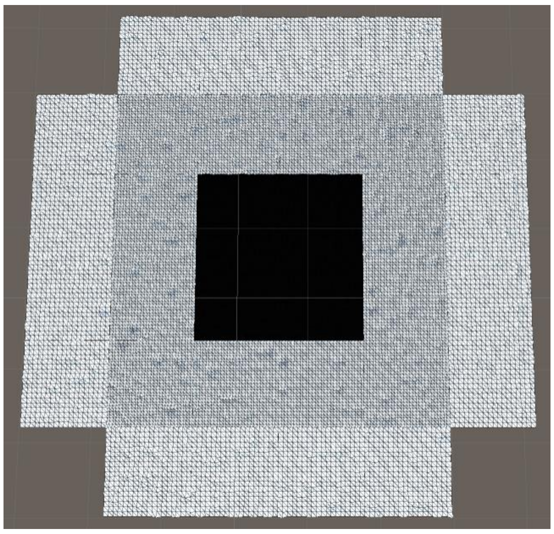


Figure 31: Endless terrain. Centre has the LOD of 0, visible distance of 200. Sides have the LOD of 4 to 6 and the visible distance of 400 to 600.

3.5.5 Shader

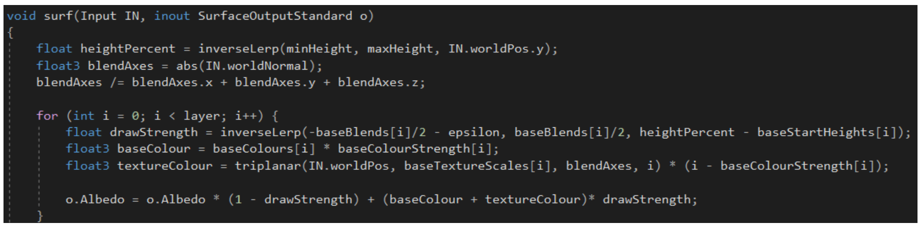
Generally, shader is a GPU-based program that process the specific part of a graphic pipeline, such as calculating the basic light, shadow, and colour of a 3D scene. Although shader can make object more realistic, it is also an isolated program which only allows data input and output. To implement shader in this project, Unity engine provides two method of programming shaders.

One common way is using Vertex and Fragment Shader to calculate the colour and texture of each vertex and fragment for the screen. They are normally written in HLSL language for different render pipeline in Unity, such as URP and HDRP. [32] As this project depends on built-in render pipeline, there is another method can be achieved, which is Surface Shader.

> *"Surface Shaders is a code generation approach in built-in render pipeline that makes it much easier to write lit shaders than using low level vertex/pixel shader programs." [32]*

It will start by collecting any useful data and UVs as input and fills in output structure, such as albedo colour, and then render the actual pixel.

Therefore, the main for implementing shader in this project is to use the surface shader. Initially, the *surf* function will check every pixel in the visible terrain and store them as input data. Then, the colour and texture of the surface at each particular point will be set by adjusting its albedo property through *SurfaceOutputStandard* parameter. The full solution is shown in Code 14.



```
void surf(Input IN, inout SurfaceOutputStandard o)
{
    float heightPercent = inverseLerp(minHeight, maxHeight, IN.worldPos.y);
    float3 blendAxes = abs(IN.worldNormal);
    blendAxes /= blendAxes.x + blendAxes.y + blendAxes.z;

    for (int i = 0; i < layer; i++) {
        float drawStrength = inverseLerp(-baseBlends[i]/2 - epsilon, baseBlends[i]/2, heightPercent - baseStartHeights[i]);
        float3 baseColour = baseColours[i] * baseColourStrength[i];
        float3 textureColour = triplanar(IN.worldPos, baseTextureScales[i], blendAxes, i) * (i - baseColourStrength[i]);

        o.Albedo = o.Albedo * (1 - drawStrength) + (baseColour + textureColour)* drawStrength;
    }
```

Code 14: surf function. The key function of Surface shader

As a result, all the terrain meshes in the generator are covered with colour shader and texture shader. The examples are shown in Figure 32.

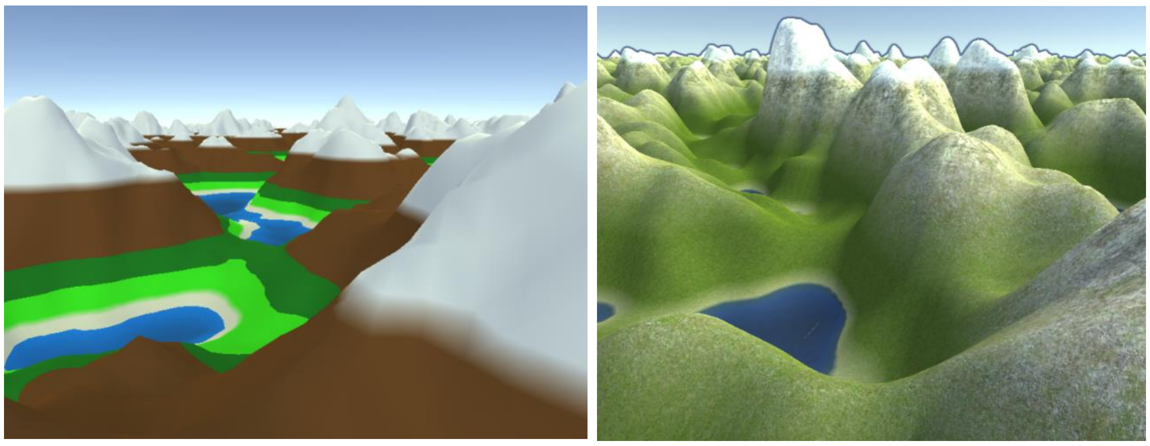


Figure 32: Left is the terrain only with Colour shader. Right is the terrain with Texture shader override on the Colour shader.

3.6 Summary

Overall, this chapter provides a detailed description of the entire implementation process in this project, covering the algorithm used to develop parts, and the methodologies for achieving each objective.

Comparing to the initial colour terrain in Figure 24 at early developing stage, the procedural terrain generation in this project now has the full functionality to automatically create more complex and natural endless topographies. More importantly, users are free to change any style of terrain they want and apply any textures for different terrain types.

In general, the final procedural terrain generation meets the desired requirements of this project. However, its rendering performance requires further testing. It is important to understand whether it can render infinite and random terrain smoothly at standard running frame rates or not. A highly efficient generator will carry more resources and space for future game development. Thus, the efficiency of this terrain generation will be used to decide if the aim would be achieved.

Chapter 4: Testing & Evaluation

4.1 Overview

This section will detailly test the outputs of procedural terrain generation in this dissertation project. During the test, all results will be compared primarily in its computational cost, such as framerate, memory, CPU, GPU and rendering time. Lastly, to conclude whether all the objectives have been achieved perfectly, the test results on the quality and performance of rendered terrain will also be evaluated graphically.

4.2 Testing tool

This section will introduce the machine and tool that been used throughout the testing.

Testing machine

The tests of all the results are ran on a PC with these specifications as follow.

CPU: AMD Ryzen 7 5800X
GPU: NVIDIA GeForce RTX 3070Ti
RAM: DDR4 32GB @3200Mhz
SSD: Samsung SSD 970 EVO Plus 1TB
HDD: WD 1TB

Unity Profiler

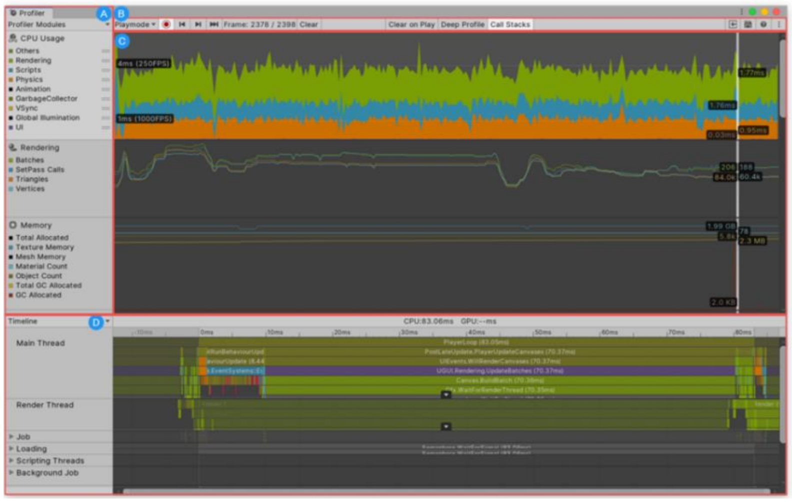


Figure 33: Unity Profiler [33]

With Figure 33 as reference, Unity engine provides a build-in Profiler to records different aspects of application's performance, and displays the data through several modules, ranging from CPU usage to GPU usage and from framerate to Memory. When running the project in the Editor, the program test will also be processed at same time in order to provide a more accurate overview on resource consumption.

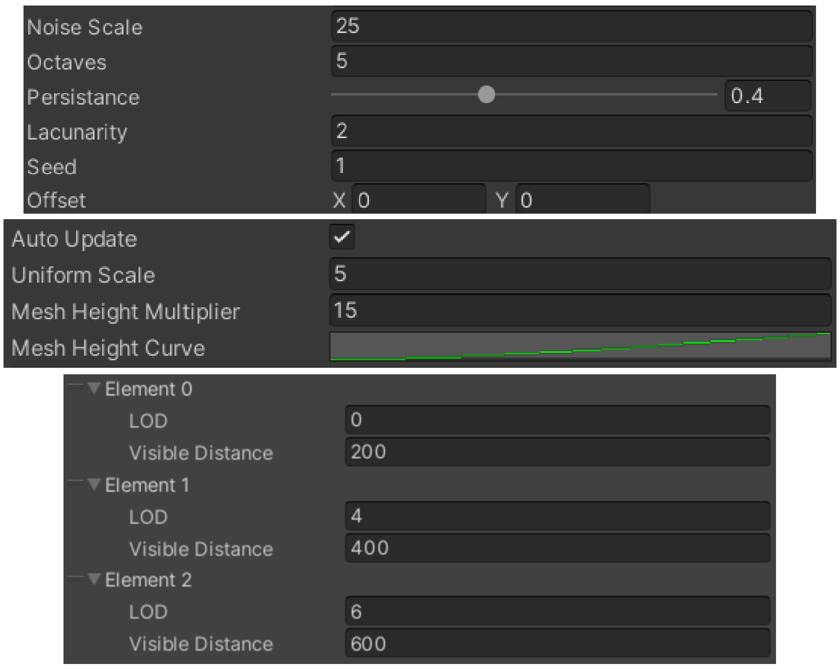
> *"The CPU Usage module provides the best overview of how much time your application spends on each frame. The other modules collect more specific data and can help you inspect more specific areas or to monitor the vitals of your application, such as memory consumption, rendering, or audio statistics." [33]*

Moreover, the deep profiling in Unity Profiler will be able to track every part of the project and arrange the performance consumption of each function from high to low. Although the deep profiling causes some extra computational costs, it will provide more comprehensive test results for this project

4.3 Testing Solution

All tests will mainly focus on the performance and quality of the terrain generator. This means that when endless terrain is generated at run time, its running framerate, total memory usage, and rendering time will be recorded as comparative data. To be specific, the main test idea is essentially similar to the control variates method. In other words, the basic operating framework of procedural terrain generation is the same, except for different test outputs.

Terrain setting
First, all terrain maps will be created with the same Perlin noise map settings, including their Octave as well as the Seed. Then, all the terrain meshes will be generated with same size and height value. Last but not least, different level of chunks will have the same level of detail and visible distance, and the player will be placed at the same position. The final setting of each output is shown in Figure 34



Noise Scale	25
Octaves	5
Persistance	0.4
Lacunarity	2
Seed	1
Offset	X 0 Y 0

Auto Update	✓
Uniform Scale	5
Mesh Height Multiplier	15
Mesh Height Curve	

Element 0	
LOD	0
Visible Distance	200
Element 1	
LOD	4
Visible Distance	400
Element 2	
LOD	6
Visible Distance	600

Figure 34: Top is terrain map setting. Middle is terrain mesh setting. Bottom is LOD and visible distance setting

Terrain outputs
With the same terrain setting, all the figures below show the testing objects of this project in order to ensure that the outputs at each stage of terrain generation are working properly before running the test.

Output Example 1

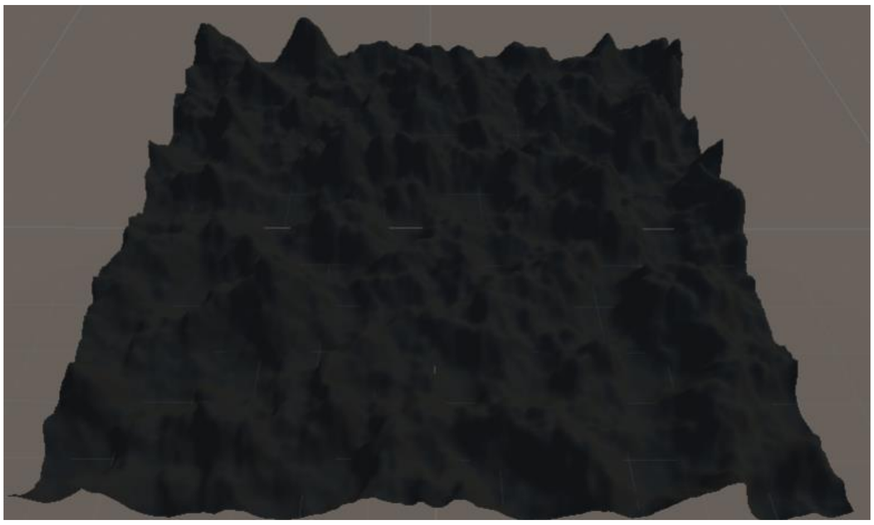


Figure 35: 3D terrain with no shader and texture

Output Example 2

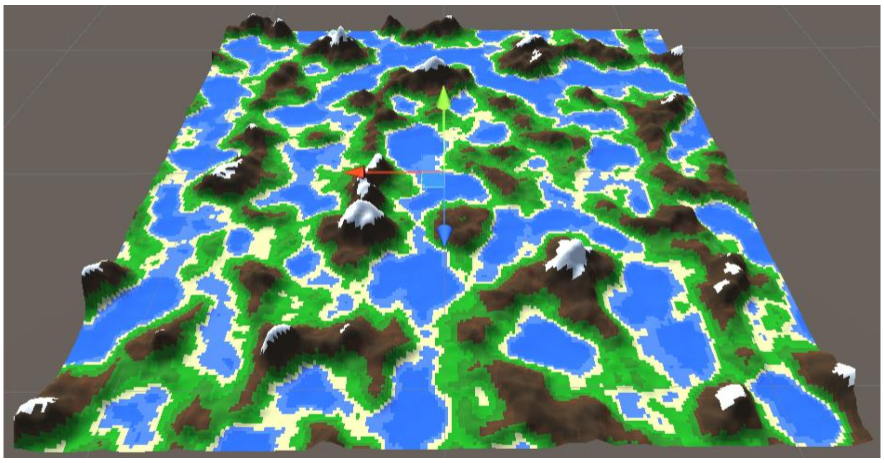


Figure 36: 3D terrain at early stage only with single colour pixel

Output Example 3

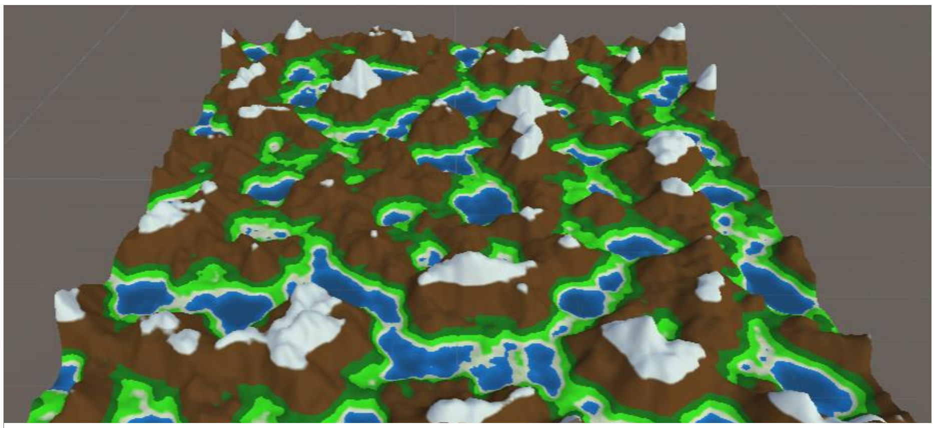


Figure 37: 3D terrain with colour shader

Output Example 4

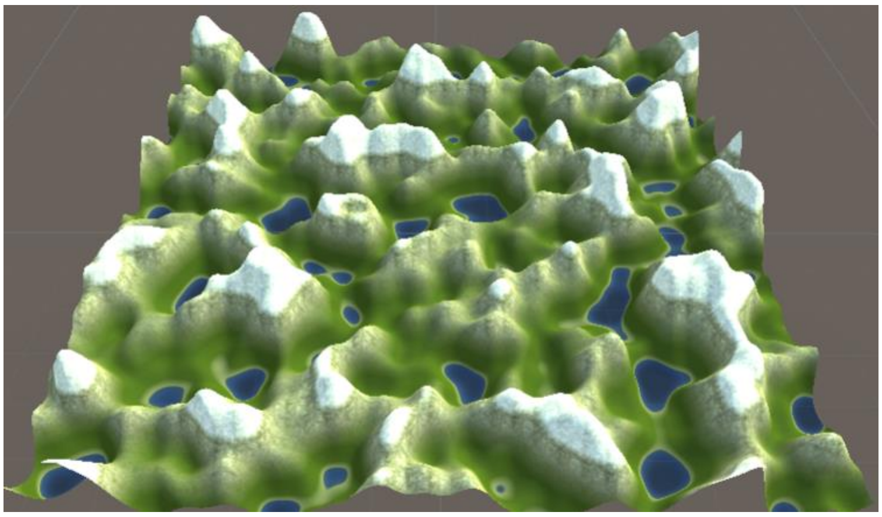


Figure 38: 3D terrain at final stage with colour and texture shader

4.4 Results Evaluation

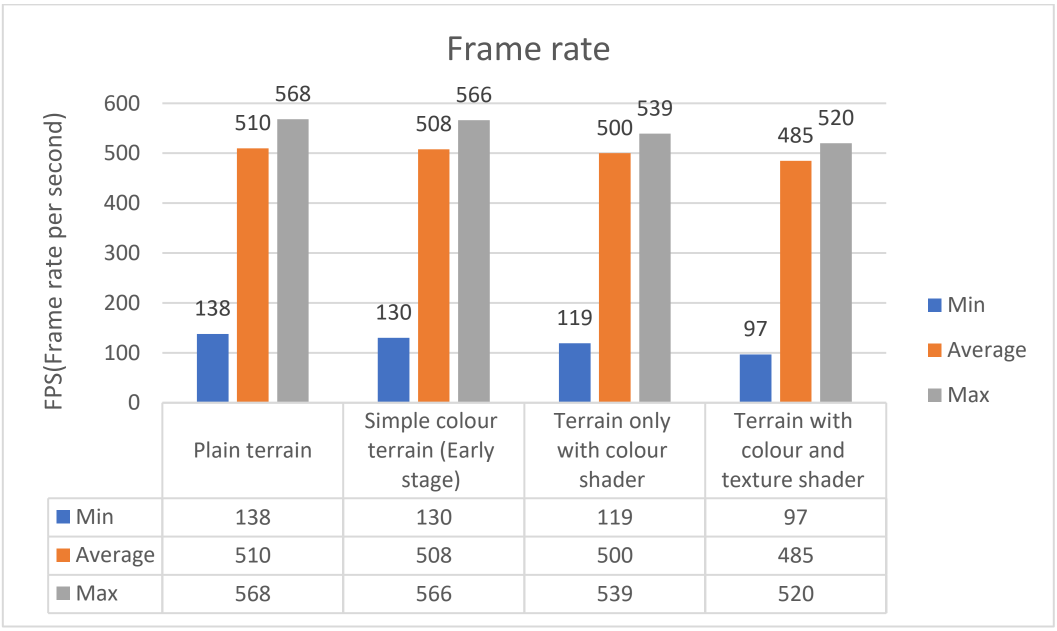
As stated earlier, the test object is to automatically generate different endless terrain outputs in one direction for one minute at a time.

Frame rate (FPS)

Frame rate is the first aspect of this project testing and is one of the most intuitive results of presenting the quality of terrain generation. Noted that frame rate in this test represents FPS (frame rate per second), which means the number of graphic frames that computer can render per second. Normally, in modern game development, the lowest frame rate for a playable game is 30FPS. However, with the development of technology, console and PCs' performance has reached a new level in recent years. Its processing power and memory consumption became cheaper and faster. This makes 60 FPS becomes the standard frame rate for evaluating an enjoyable gaming experience. When frame rate reaches to three digits, the game will provide the best graphic performance.

> *"For most people, 60 FPS is the best frame rate to play at. This isn't only because of the smoothness of the images displayed, but also because 60Hz monitors are the most readily available ones." [34]*

Therefore, in this project, the higher the running frame rate when generating endless terrain, the smoother the rendering environment will be. As a result, the frame rate comparison of different terrain outputs is shown in the Chart 1 below.



	Plain terrain	Simple colour terrain (Early stage)	Terrain only with colour shader	Terrain with colour and texture shader
Min	138	130	119	97
Average	510	508	500	485
Max	568	566	539	520

Chart 1: Frame rate comparison of different testing objects

In general, the plain terrain shows the highest frame rate in every category, including minimum frame rate, average frame rate, and maximum frame rate, while the terrain with colour and texture shader was the lowest. This is the expected result because plain terrain does not cover any texture and colour. Thus, it felt smoother than the terrain with shader when rendering pixels of each vertex in the meshes.

Moreover, it is interesting that every output shows the high performance following the order of plain terrain, Simple colour terrain, Terrain only with colour shader and Terrain with colour and texture shader, which reflects that the procedural terrain generation in this project can steadily generates random and complex terrain at standard frame rates.

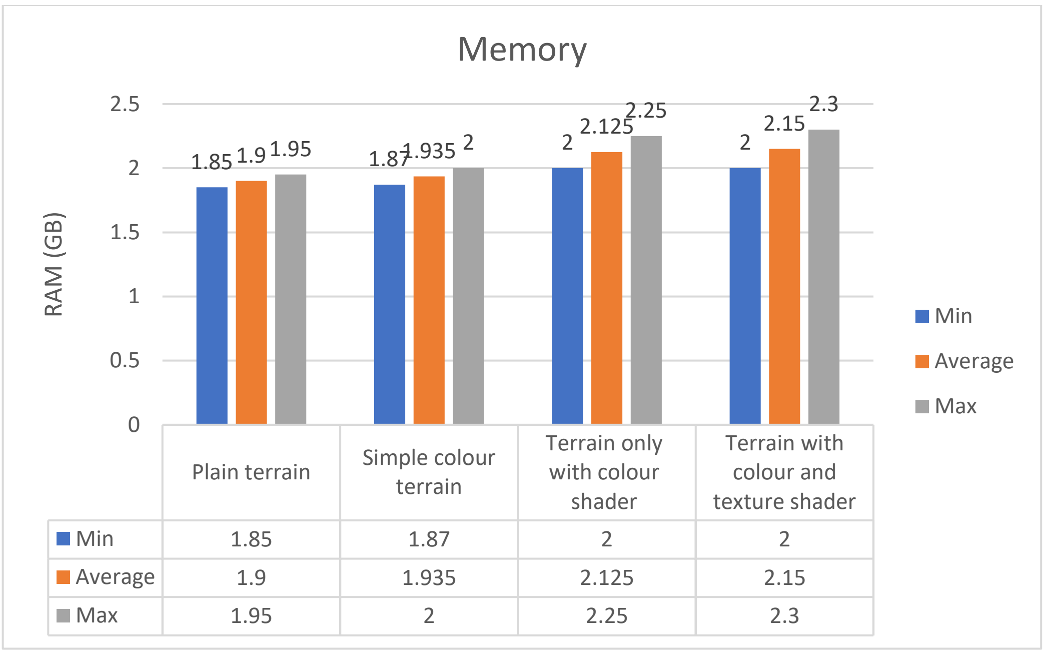
However, all the average value by every sector was higher than usual unlike the average value from mathematics. There are two possible reasons for explaining the significant drop in minimum frame rate. First of all, this project was tested in a deep profiling environment. As mentioned before in the test solution, deep profiling is resource-intensive and sometimes will cause the frame rate drop dramatically when tracking each script of the project. The second reason is that the project suddenly froze at some moment while creating new terrain. By monitoring the module details panel from Unity Profiler, the debugging information shows that Update function of EndlessTerrain method takes longer time to process than usual, which emphasizes the further optimization on generating endless terrain need to improve in the future.

In addition, even though of the pervious analysis, plain terrain still shows the highest performance with minimum value of 138 while others show the lowest of 97. In consequence, based on the comparation result, although the deviation between minimum frame rate and maximum frame rate is greater than normal, it is reasonable because of the increasing complexity of the terrain. Nevertheless, the stable average frame rate from each outputs represents the high quality of generating terrain in this project.

Memory (RAM)
The memory usage is the second part of this project testing and RAM here is referred as random access memory. In modern game, to achieve fast loading, RAM helps computers move information and quickly access game data from storage disk, such as SSD or HDD. If the computer does not have enough memory to access all of the game information that is functioning normally, it will cause the game to lose frame rate and performance. In more serious cases, the extreme lack of memory may even prevent the application from running completely.

“8GB is the baseline for gaming today, but 16GB is a good future-proofed option” [35]

Alternatively, in this project, the less memory the procedural terrain generation consumers to create random terrain, the more memory storage the computer will be saved to load the program quickly. Eventually, low memory cost represents the high performance of this project. As a result, the comparison of memory consumption of different terrain outputs is shown in the Chart 2 below.



	Plain terrain	Simple colour terrain	Terrain only with colour shader	Terrain with colour and texture shader
🟦 Min	1.85	1.87	2	2
🟧 Average	1.9	1.935	2.125	2.15
⬜ Max	1.95	2	2.25	2.3

Chart 2: Comparison of memory consumption of different testing objects

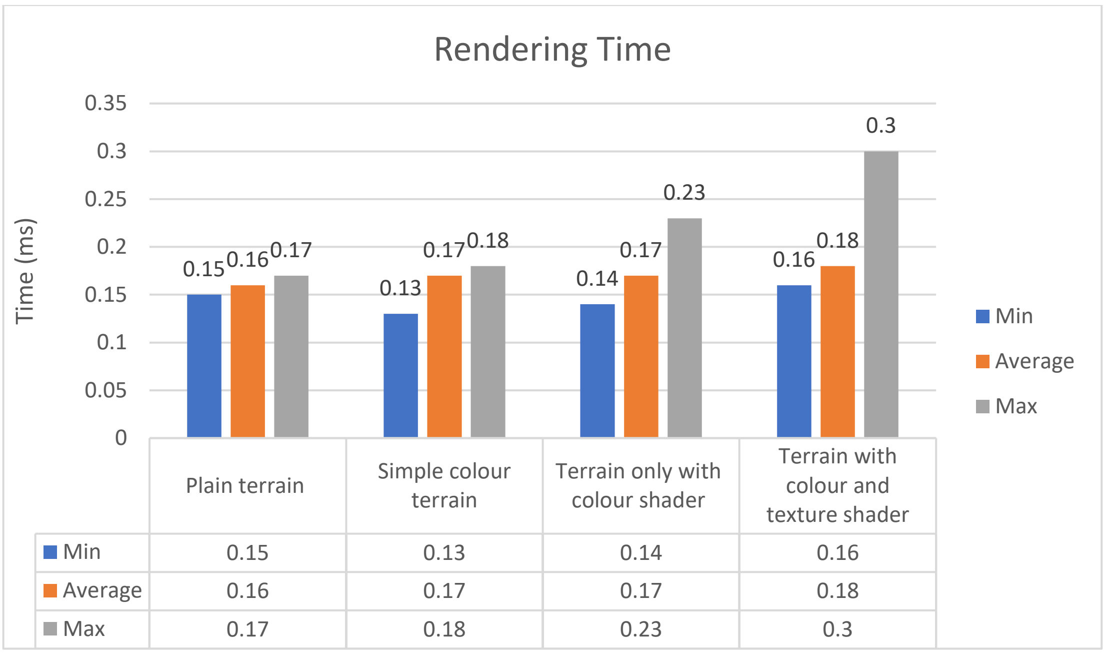
Overall, the memory cost shows the similar testing result as frame rate showing the highest of Plain terrain and the lowest of Terrain with colour and texture shader. This is the expected result because the plain terrain has lower complexity terrain than other outputs, which implies less memory will be used to transmit texture data.

Also, there is not much difference between each output, the average memory consumption is all around 2GB. This is due to the functionality of endless system in this project. As described earlier in Chapter 3, the terrain generation will not process the unloaded chunks and will also lower the game aspects when loaded chunks are far away from players. Even though the new terrain is regenerated, those chunks that disappeared in certain visible distance will be destroyed in order to free more memories and keeping the game running smoother.

However, in future developments, more complex texture may be added to this project, such as water shader or vegetation system. It could significantly increase the memory consumption and more future test will be done. Even so, in aspect of memory cost in different outputs, the average 2GB consumption is completely within control of procedural terrain generation in this project, which also represents it high performance of generating terrain.

Rendering time

The final piece of this project testing is its rendering time. Unlike other testing results, rendering time reveals how much time the procedural terrain generation spends on rendering each output. This will directly expose the efficiency and productivity of this project. More importantly, fast rendering represents a good optimization in the generator. Consequently, the comparison of rendering time of different terrain outputs is shown in the Chart 3 below.



	Plain terrain	Simple colour terrain	Terrain only with colour shader	Terrain with colour and texture shader
Min	0.15	0.13	0.14	0.16
Average	0.16	0.17	0.17	0.18
Max	0.17	0.18	0.23	0.3

Chart 3: Comparison of rendering time of different terrain outputs

In the aspect of Rendering, Terrain with colour and texture shader took the longest time of 0.3ms among four technologies due to its high complexity while as the plain terrain was the lowest of 0.15ms. It is an expected result since low texture and colour were assigned to each vertex in the meshes, which can also be reasonably used to explain the distinct between terrain only with colour shader and simple colour terrain.

However, unlike the maximum value, the average and the minimum appear not much difference. This is because the detail levels and visible distance in different part of terrain chunks are set with same value. Thus, same number of vertices and triangles are generated around similar time for different outputs. The minimal deviation in rendering time due to different mesh materials can be ignored. Generally, four different outputs have the similar rendering times around 0.17ms in average, which implies the highly efficient rendering capability of procedural terrain generation in this project.

4.5 Summary

Overall, the performance test of procedural terrain generation in this project was successful. By monitoring and analysing the frame rate, memory and rendering time of each output, it is clear to see that the quality of this terrain generation does not decrease with its increasing complexity. Although some of the test results shew the significant variations in specific data, these deviations were all reasonable and controllable. More importantly, the procedural terrain generation can quickly render random topography for different requirements in a low memory consumption situation with a stable frame rate.

However, during the test run, this project sometimes experienced a sharp drop in the number of frames as the terrain was generated, which produces some unnecessary pauses in the game. The cause of this problem can be speculated that all the endless terrain system was updated by every frame rather than depends on the visible distance of player. Therefore, future project optimization needs to be improved.

In addition, if fancier shader is added to future project for a higher level of graphics effect, the performance of this procedural terrain generation will be further tested.

Chapter 5: Conclusion

This section will reflect on the project as a whole, including a technical summary of the achievement of the objectives in Chapter 1 as well as a conclusion of meeting overall aim based on the methodology in Chapter 3 and testing evaluation in Chapter 4. Furthermore, the project will also be summarised and evaluated critically about what knowledge author has learned and what section can be improved in the future.

5.1 Satisfaction of the Aim and Objectives

The aim of this project is to implement a procedural and dynamic terrain generation using Unity engine that can help modern games populate vast surface areas. It will be able to render an “endless” scrolling surface in real-time with different types of terrains and textures. This aim is divided into several objectives, and the completeness of these objectives will verify that whether the development of this project satisfy the needs of the aim or not.

1. **Explore the usage of noise function in creating terrain and identify the best function that can be implemented in Unity to generate random heightmaps**
   This objective has been successfully achieved. By reviewing the algorithms of procedural generation techniques and figuring out the usage of different noise functions through Chapter 2, the Perlin noise meets the requirement of generating random terrain maps. Accordingly, with the implementation of Noise map and Colour map methods, an initial pseudorandom 2D terrain map with different types of topographies is well generated, which meets the requirements of its randomness.

2. **Research the data of mesh in Unity and create a terrain mesh by generating heightmaps**
   With the implementation in first objective, this object is also accomplished perfectly by the development of terrain mesh generator in Chapter 3. Under the help of the MeshGnerator Method, the 2D terrain map is fully filled with multiple vertices and triangles. User can adjust the shape and style of different 3D terrain mesh through its height values. Comparing to the terrain manually designed by artist, this project provides a more convenient and fast way to generate complex terrain.

3. **Understand the usage of Level of Detail (LOD) and implement the endless terrain system for terrain generation.**
   This objective also has fully been met. Chuck system is the key to create endless terrain. When player moves freely on the terrain map, the procedural terrain generation in this project can automatically generate new terrain around the player, which achieves the requirement of endless scrolling surface in real-time. Even to achieve better optimization, the implementation of level of detail makes the project run more smoothly.

4. **Implement custom terrain shader and UI for better visualization**

This objective was partially satisfied. To make the terrain as real as when designer creates it, the custom terrain shader, such as colour shader and texture shader, are well implemented in this project. Users are free to change any style of terrain they want and apply any textures for different terrain types. However, through the discussions with supervisor, he agreed that UI was not necessary to be implemented in this project since all the adjustments can be changes inside the Unity Editor.

5. **Summarise and evaluate the performance of terrain generation**
   The performance tests were conducted on the final version of procedural terrain generation in this project, including its frame rate, memory usage, and the overall rendering time for each terrain outputs. As the evaluation of the test results in Chapter 4 presents, the development of procedural terrain generation has achieved the overall aim of this project.

5.2 What went well

Generally, the author believes the development of procedural terrain generation went well. As mentioned earlier in this dissertation, the biggest technical challenge here is to automatically generate terrain in Unity and they look as good as when designers do it. Nevertheless, the exploration of different techniques and the approaches to the implementation of different methods, such as Perlin Noise or Chunk system, are the key to overcome this challenge. Moreover, with the performance testing at the end, author was sure that the procedural terrain generation in this project was best designed for creating vast surface area in modern game development.

5.3 What could have been done better

In the final test evaluation of Chapter 4, the endless terrain system was speculated to have some optimization problems. Hence, in order to reach a smoother rendering experience, the author think more improvement could be managed on system threading and help mitigate some thread creation overhead, especially the thread for processing endless system.

Moreover, computers with different graphic cards might have different results in evaluating the performance of procedural terrain generation. Even though in the final performance test, the terrain generation can successfully render a variety of complex terrain on test machine (NVIDIA GeForce RTX 3070TI) at a stable running frame rate of 60 and normal memory consumption, this does not mean it can run smoothly on other low-power graphics cards, such as 1050ti. Therefore, further test on different machine needs to be considered.

5.4 Personal development

The author has gained a better understanding of using different noise function to generate pseudorandom contents in game development. Through the development of meshes and shader in this project, author has become more proficient in creating and rendering complex 3D models in Unity. More importantly, the author not only felt confident in development higher-graphical games, but also interested in the knowledge behind computer graphics. It is exciting to create some realistic looking contents in virtual world.

5.5 Future work

The author believes that there are still plenty of space for future improvement. The first is the different selection on the noise function. Currently, the procedural terrain generation mainly create terrain in 3D. If the terrain is going to be generated at a higher dimension, such as 4D or 5D, Perlin noise will be less efficient. Thus, as stated previously, Simplex noise will be the beat choose for scaling to higher dimensions with much less computational cost.

Secondly, comparing to the graphic quality of some AAA game, this project can further improve the terrain quality and make the texture more realistic. Therefore, some of the more complicated procedural generation algorithms can be implemented. For example, the Erosion algorithm, which provides an extra layer of realism on the terrain. Or the vegetation system for creating realistic features and plants.

Last but not the least, there are plenty of fancy shader can also be applied to this project for a better-looking effect, such as water shader. However, as the complexity of the terrain grows, future performance test will be done to prevent game from overload.

References

[1] J. Clement, "COVID-19 impact on the gaming industry worldwide - statistics & facts," 4 Jun 2021. [Online]. Available: https://www.statista.com/topics/8016/covid-19-impact-on-the-gaming-industry-worldwide/#topicHeader__wrapper. [Accessed 2 May 2022].

[2] C. Compton, "How Minecraft Generates Massive Virtual Worlds from Scratch.," Rempton Game, 28 Feb 2021. [Online]. Available: https://remptongames.com/2021/02/28/how-minecraft-generates-massive-virtual-worlds-from-scratch/. [Accessed 2 May 2022].

[3] A. Kharpal, "No Man's Sky": Would you play a game that takes 584 billion years to explore?," CNBC, 11 Aug 2016. [Online]. Available: https://www.cnbc.com/2016/08/10/no-mans-sky-release-would-you-play-a-game-that-takes-584-billion-years-to-explore.html. [Accessed 2 May 2022].

[4] "Unity - Manual: Terrain," Unity Documentation, Mar 2021. [Online]. Available: https://docs.unity3d.com/Manual/script-Terrain.html. [Accessed 2 May 2022].

[5] S, "Procedural Generation – A Comprehensive Guide Put in Simple Words," Scaleyourapp, [Online]. Available: https://www.scaleyourapp.com/procedural-generation-a-comprehensive-guide-in-simple-words/. [Accessed 2 May 2022].

[6] N. A. Barriga, "A Short Introduction to Procedural Content Generation Algorithms for Videogames," International Journal on Artificial Intelligence Tools, vol. 28, no. 2, 2019. [Accessed 5 May 2022].

[7] J. Togelius, G. N. Yannakakis, K. O. Stanley and C. Browne, "Search-Based Procedural Content Generation," Applications of Evolutionary Computation. EvoApplications 2010. Lecture Notes in Computer Science, vol. 6024, 2010. [Accessed 5 May 2022].

[8] "Procedural Generation," Grendel Games, [Online]. Available: https://grendelgames.com/procedural-generation/. [Accessed 5 May 2022].

[9] K. Tokarev, "Procedural Technology in Ghost Recon: Wildlands," 80lv, 13 Apr 2017. [Online]. Available: https://80.lv/articles/procedural-technology-in-ghost-recon-wildlands/. [Accessed 5 May 2022].

[10] M.-F. G. Tsai, "Fractal Landscapes," MAT 335 Project, 2003. [Accessed 5 May 2022].

[11] T. Archer, "Procedurally Generating Terrain". [Accessed 5 May 2022].

[12] M. Fogelman, "Procedural Terrain Generation: Diamond-Square," 26 Dec 2015. [Online]. Available: http://jmecom.github.io/blog/2015/diamond-square/. [Accessed 5 May 2022].

[13] "Three Ways of Generating Terrain with Erosion Features," Github, 15 Jun 2021. [Online]. Available: https://github.com/dandrino/terrain-erosion-3-ways. [Accessed 6 May 2022].

[14] "Coherent noise," libnoise, 2005. [Online]. Available: http://libnoise.sourceforge.net/glossary/index.html#noisemodule. . [Accessed 6 May 2022].

[15] "Noise, being a pseudorandom artist," Catlike Coding, [Online]. Available: https://catlikecoding.com/unity/tutorials/noise/. . [Accessed 6 May 2022].

[16] "Noise and Turbulence," [Online]. Available: https://cs.nyu.edu/~perlin/doc/oscar.html. . [Accessed 6 May 2022].

[17] S. Gustavson, "Simplex noise demystified," 2005. [Accessed 6 May 2022].

[18] C.-J. Rosén, "Cell Noise and Processing," 2006. [Accessed 7 May 2022].

[19] "Unity - Manual: Graphics.," Unity Documentation, Mar 2021. [Online]. Available: https://docs.unity3d.com/Manual/Graphics.html. [Accessed 7 May 2022].

[20] "Mathf.PerlinNoise," Unity Documentation, Mar 2021. [Online]. Available: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html. [Accessed 7 May 2022].

[21] "Using the Mesh Class," Unity Documentation, Mar] 2021. [Online]. Available: https://docs.unity3d.com/Manual/UsingtheMeshClass.html. [Accessed 8 May 2022].

[22] "Level of Detail (LOD) for meshes," Unity Documentation, Mar 2021. [Online]. Available: https://docs.unity3d.com/Manual/LevelOfDetail.html. [Accessed 8 May 2022].

[23] W. W. Royce, "MANAGING THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS," pp. 1-9, 1970. [Accessed 10 May 2022].

[24] teamgantt, [Online]. Available: https://prod.teamgantt.com/gantt/schedule/?ids=3011199#&ids=3011199&user=&custom=&company=&hide_completed=false&date_filter=&color_filter=. [Accessed 10 May 2022].

[25] M. Dealessandri, "What is the best game engine: is Unity right for you?," gameindustry, 16 Jan 2020. [Online]. Available: https://www.gamesindustry.biz/articles/2020-01-16-what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you. [Accessed 11 May 2022].

[26] "Great Graphics," arnia software, 10 Aug 2020. [Online]. Available: https://www.arnia.com/what-makes-unity-so-popular-in-game-development/#:~:text=When%20it%20comes%20to%20graphics,creation%20of%20fantastic%20looking%20games.. [Accessed 11 May 2022].

[27] "Why You Should Be Using the Unity Game Engine," informIT, 4 Apr 2013. [Online]. Available: https://www.informit.com/articles/article.aspx?p=2031153. [Accessed 11 May 2022].

[28] "AnimationCurve," Unity Documentation, Mar 2021. [Online]. Available: https://docs.unity3d.com/ScriptReference/AnimationCurve.html. [Accessed 11 May 2022].

[29] "Chunk," Minecraft Wiki, [Online]. Available: https://minecraft.fandom.com/wiki/Chunk. [Accessed 12 May 2022].

[30] J. H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," vol. 19, no. 10, 1976. [Accessed 12 May 2022].

[31] W. H. Boer, "Fast Terrain Rendering Using Geometrical MipMapping," 2000. [Accessed 13 May 2022].

[32] "Writing Surface Shaders," Unity Documentation, Mar 2021. [Online]. Available: https://docs.unity3d.com/Manual/SL-SurfaceShaders.html. [Accessed 13 May 2022].

[33] "Profiler overview," Unity Documentation, Mar 2021. [Online]. Available: https://docs.unity3d.com/Manual/Profiler.html. [Accessed 13 May 2022].

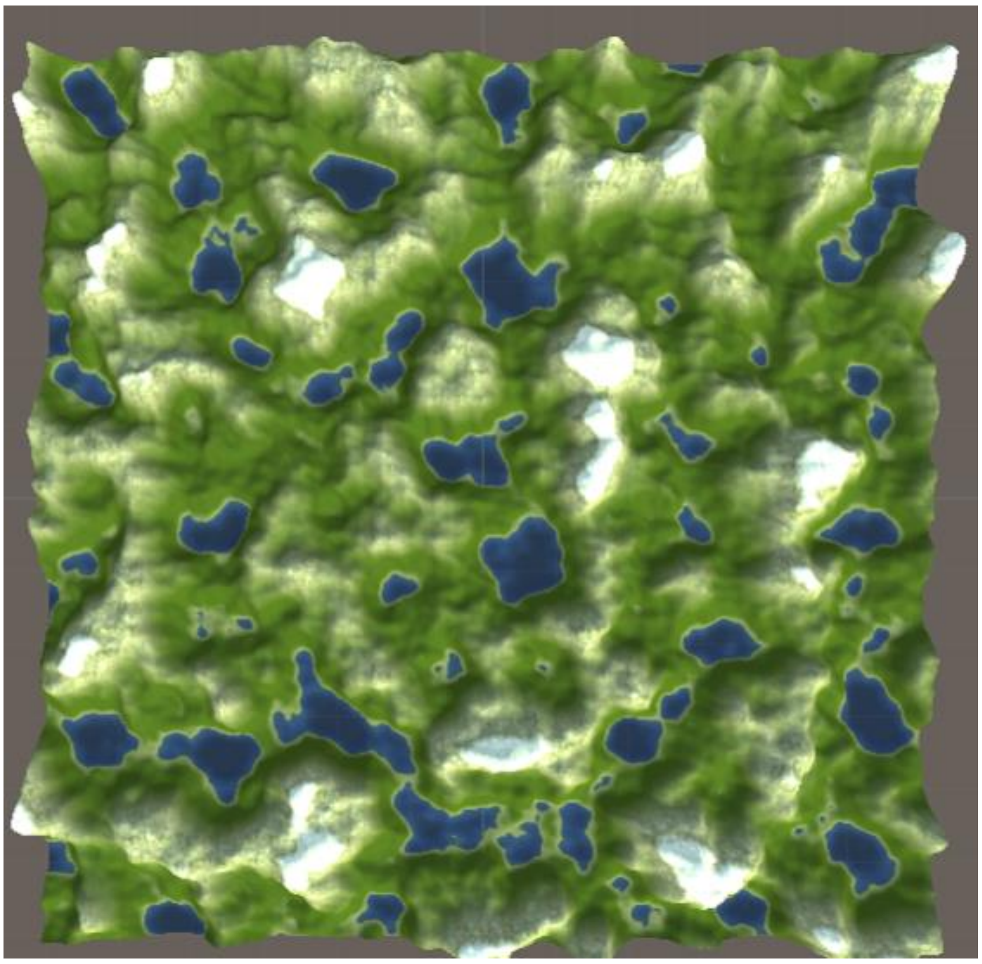
[34] B. Gapo, "What Is A Good FPS For Gaming?," GPU Mag, 25 Oct 2021. [Online]. Available: https://www.gpumag.com/good-fps-for-gaming/#:~:text=The%20old%20standard%20of%2030,the%20standard%20is%2024%20FPS.. [Accessed 14 May 2022].

[35] B. Stegner, "What Does RAM Do for Gaming and How Much RAM Do I Need?," MUO, 11 Jul 2019. [Online]. Available: https://www.makeuseof.com/tag/ram-for-gamers-what-do-the-specs-mean-and-how-do-they-alter-performance/. [Accessed 14 May 2022].

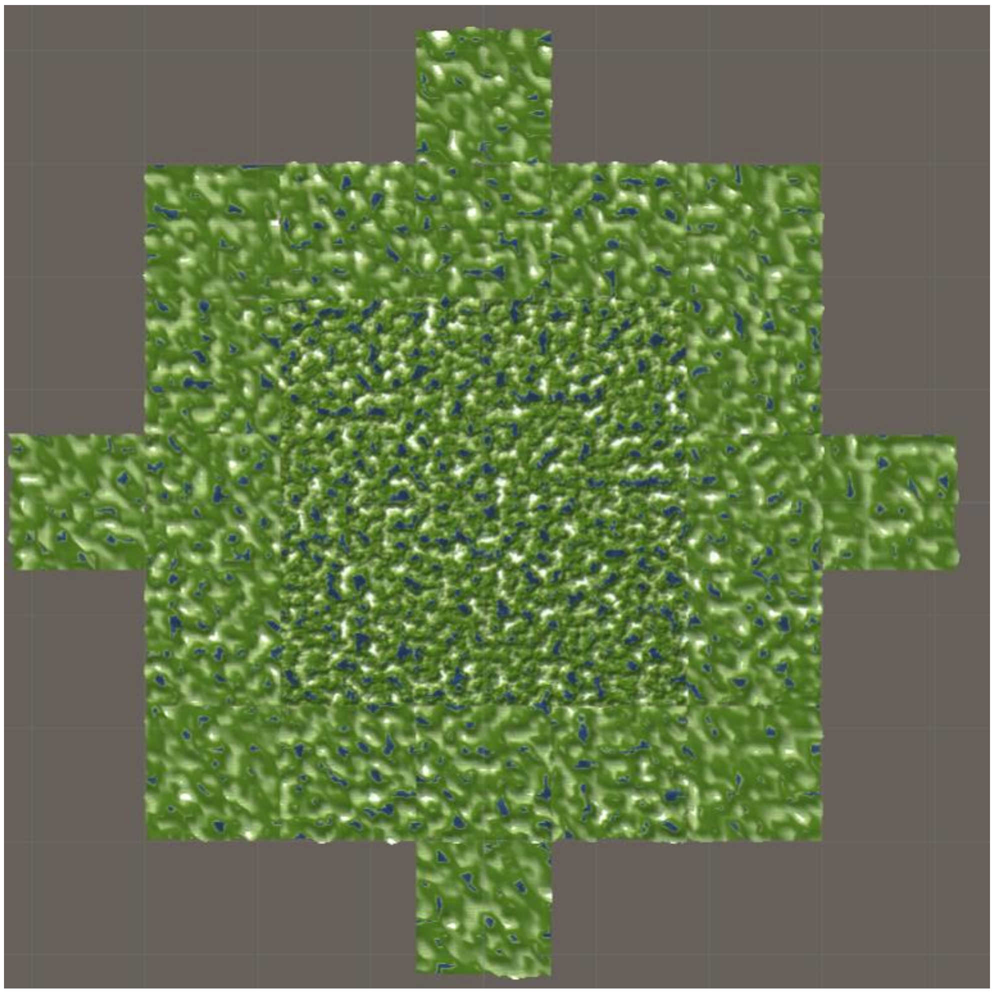
Appendices

Procedural Terrain Generation screenshots

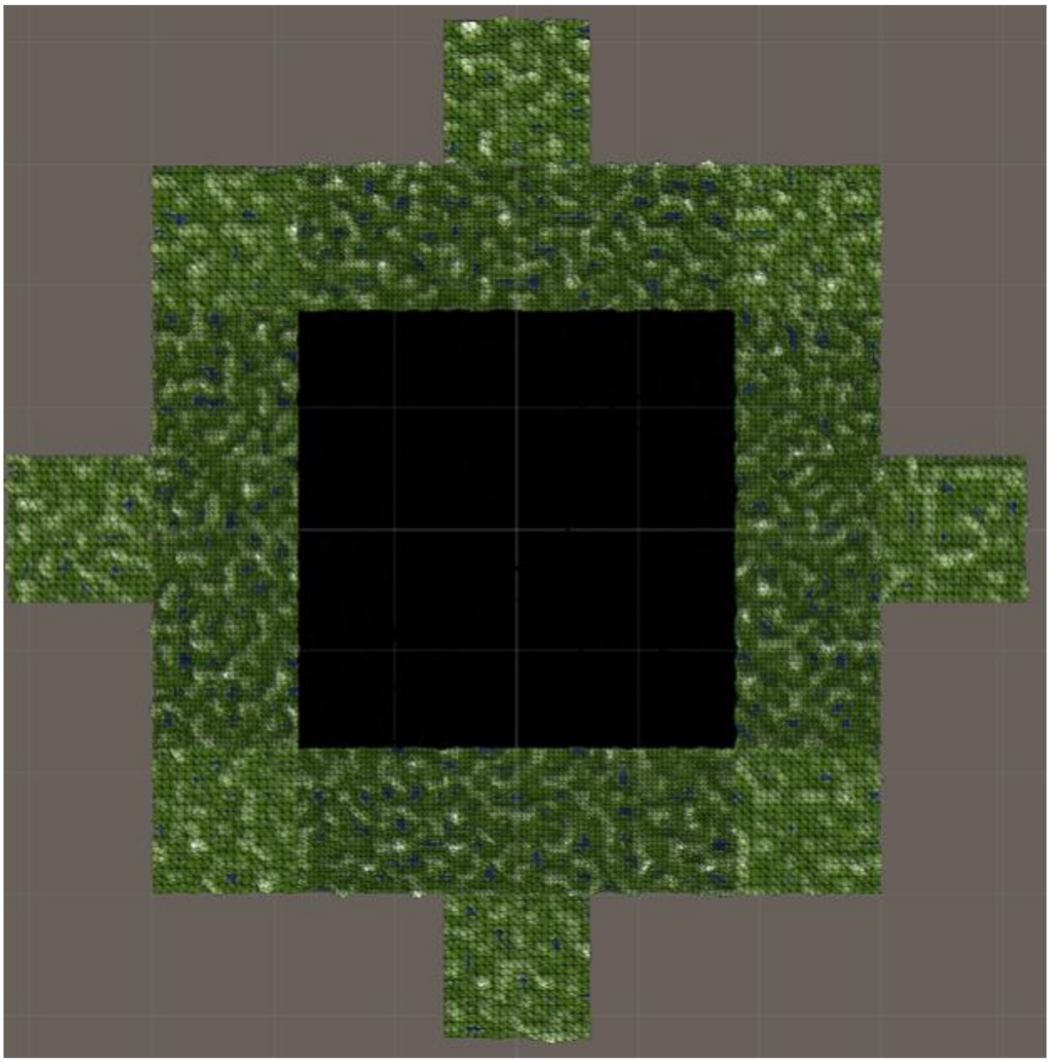
Single terrain mesh



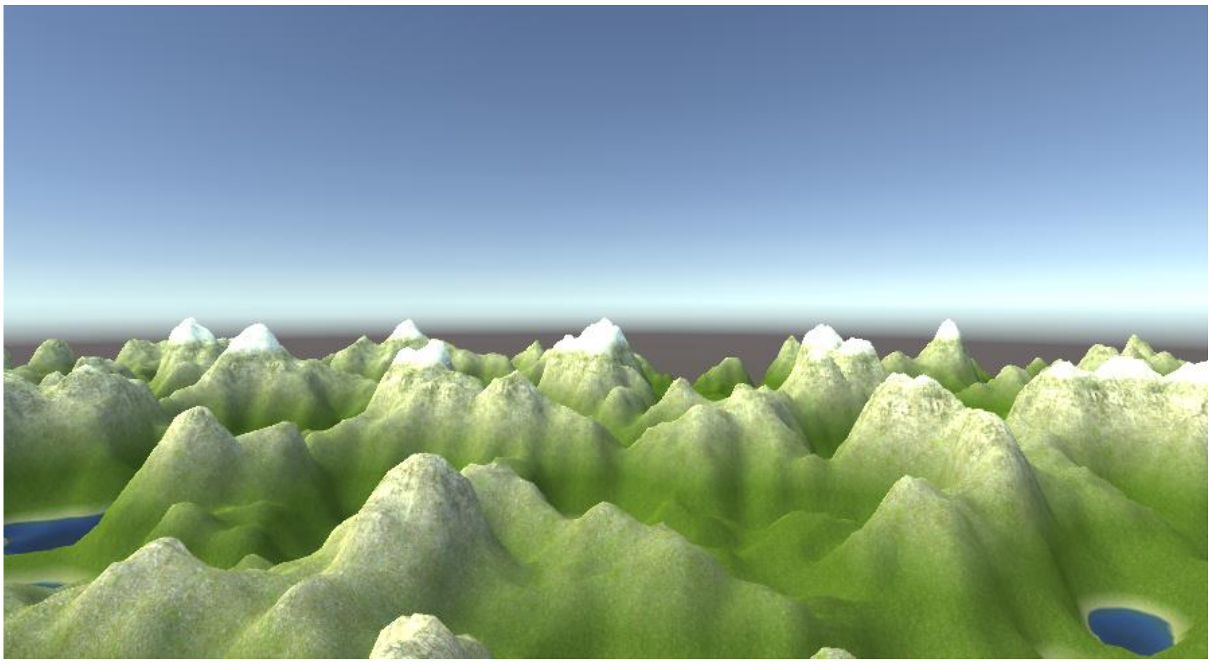
Endless terrain mesh



Endless terrain mesh with wireframe

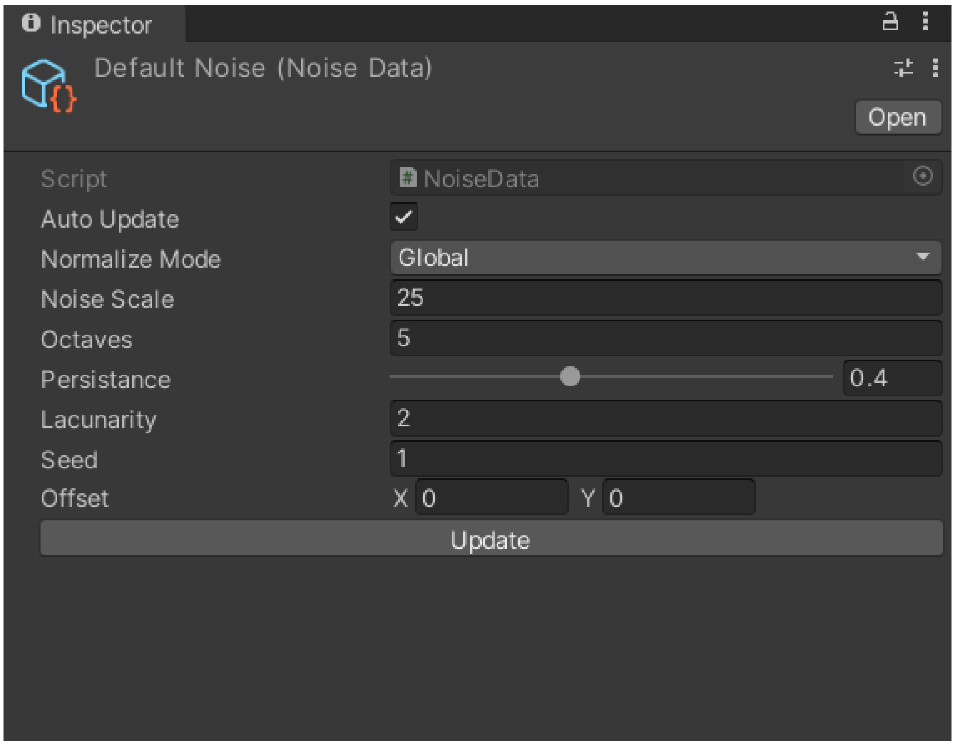


Endless terrain at run time



Terrain Editor

Noise editor


Inspector

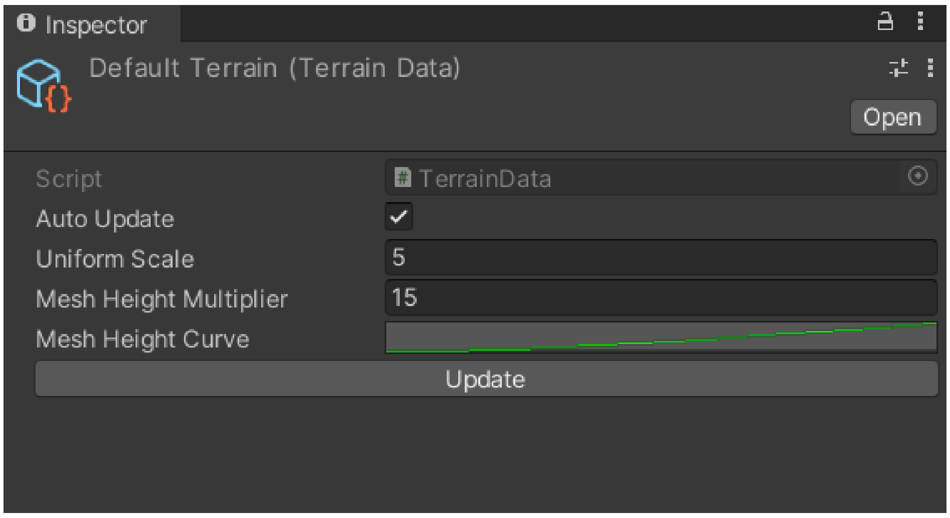
Default Noise (Noise Data)

Open

Script: NoiseData
Auto Update: ✓
Normalize Mode: Global
Noise Scale: 25
Octaves: 5
Persistance: 0.4
Lacunarity: 2
Seed: 1
Offset: X 0 Y 0

Update

Mesh editor


Inspector

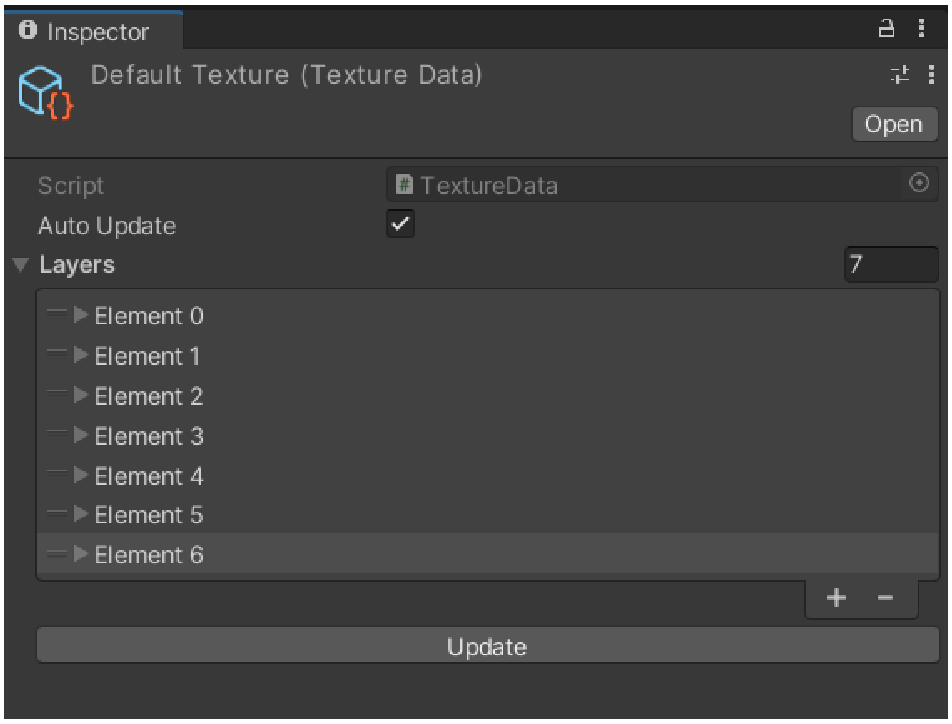
Default Terrain (Terrain Data)

Open

Script: TerrainData
Auto Update: ✓
Uniform Scale: 5
Mesh Height Multiplier: 15
Mesh Height Curve

Update

Texture editor



Inspector

Default Texture (Texture Data)

Open

Script: TextureData

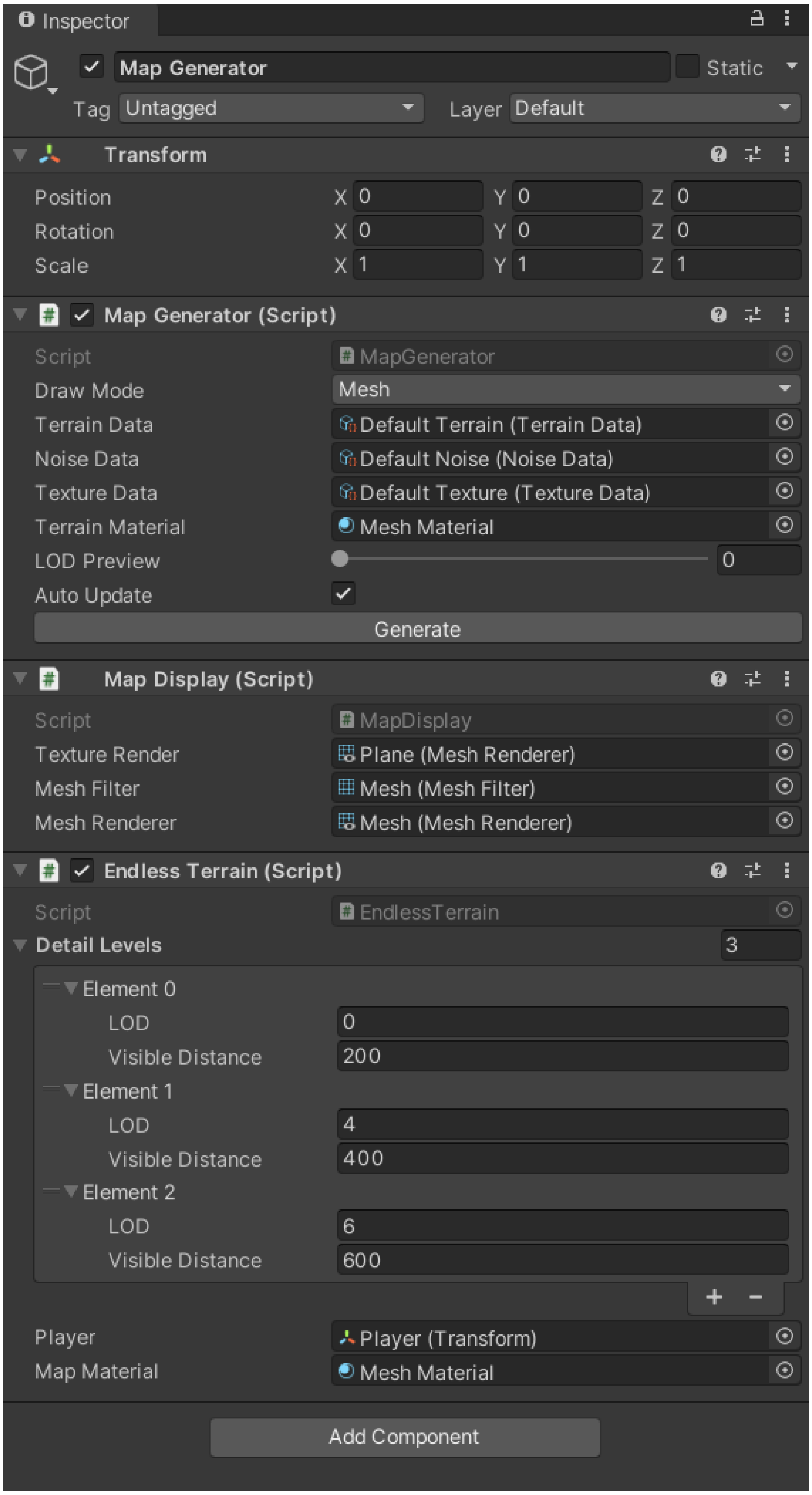
Auto Update: ✓

Layers: 7

Element 0
Element 1
Element 2
Element 3
Element 4
Element 5
Element 6

Update

Map Generator


Inspector

Map Generator
Static
Tag Untagged
Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Map Generator (Script)

Script	MapGenerator
Draw Mode	Mesh
Terrain Data	Default Terrain (Terrain Data)
Noise Data	Default Noise (Noise Data)
Texture Data	Default Texture (Texture Data)
Terrain Material	Mesh Material
LOD Preview	0
Auto Update	✓

Generate

Map Display (Script)

Script	MapDisplay
Texture Render	Plane (Mesh Renderer)
Mesh Filter	Mesh (Mesh Filter)
Mesh Renderer	Mesh (Mesh Renderer)

Endless Terrain (Script)

Script	EndlessTerrain
Detail Levels	3
Element 0	
LOD	0
Visible Distance	200
Element 1	
LOD	4
Visible Distance	400
Element 2	
LOD	6
Visible Distance	600
Player	Player (Transform)
Map Material	Mesh Material

Add Component