PROCEDURAL GENERATION OF AN ALIEN PLANET

BY HAITHAM AL SAIDI (180274773)
BSC COMPUTER SCIENCE WITH GAME ENGINEERING
SUPERVISED BY DR GRAHAM MORGAN

7/05/2021
10,424 WORDS

ABSTRACT

This dissertation looks at techniques for procedurally generating terrain on an alien planet. The adopted method involves using Unity and Perlin noise to create endlessly generating terrain with different topographical features.

The motivation for this project stems from my belief that it is important for indie developers and smaller games companies to have accessible technologies available to them that can reduce their required budgets of time and investment. Procedural generation is one such feature.

The output of this project is a highly customizable terrain generator capable of creating varying biomes and planet types by manipulating Perlin noise, mesh properties and texture layers.

DECLARATION

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

ACKNOWLEDGEMENTS

I would like to thank my project supervisor Dr Graham Morgan as well as Dr Gary Ushaw and Dr Richard Davison for their advice and support while making this project. I would also like to thank my father for helping me plan my schedule and organize myself properly. Finally, I would like to thank the Brackeys and Sebastian Lague YouTube channels for providing high quality educational videos on game development with Unity that helped me create my project.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

This section is an introduction to the dissertation that will provide context and detail the problem, as well as lay out the aims and objectives.

1.1 CONTEXT

Procedural generation is an algorithmic method of generating content using a mixture of human generated assets and computer pseudo-randomness. One of its most prevalent applications is generating the levels or worlds, amongst other aspects, of videogames.

Its prevalence is due to it being an accessible technology that enables game developers to generate large amounts of re-playable content with lots of variety for lower financial and time costs. This has enabled indie studios to generate game worlds on scales that would not have been possible on their budgets. A great example of this is Minecraft by Mojang, which started as an unemployed developer's passion project. It now harbours one of the biggest gaming communities on the planet and has been bought by Microsoft for US$2.5 billion in 2014. [1][2]

I believe it is vital that smaller developers, who are largely the source of the gaming industries new ideas, have tools like procedural generation available to them in order to compete with larger game companies with higher budgets. [3]

1.2 THE PROBLEM

Procedural generation has significant drawbacks a developer would need to consider.

Due to its random nature, it leaves the developer with less control over quality and choice. For example, it would be difficult to create a game with scripted events in a randomly generated world, as there is no way to guarantee the order in which players would come across them or if they come across them at all. This means a developer is limited in the types of gameplay they can offer with a procedurally generated game.

Furthermore, procedurally generated games tend to be very taxing on hardware. This is because, instead of simply loading premade assets, the game is constantly generating new content, which can have a significant performance impact. This would likely explain why procedurally generated games tend to be limited in their graphics implementation.

1.3 AIMS & OBJECTIVES

1.3.1 AIM

To create a video game with Unity that utilizes procedural generation with Perlin noise to achieve pseudo-random endless terrain generation.

1.3.2 OBJECTIVES

1. To create a chunk of terrain using a triangle mesh.
2. To use Perlin noise to create variation in topography, including biomes and typical terrain features such as lakes, oceans, islands, plains, hills, deserts, tundra, and mountains.

3. To implement infinite, repeated and pseudo-randomly generated chunks of terrain.
4. To ensure an acceptable level of performance by utilizing techniques such as multithreading and variable levels of detail. The measurable criteria are as follows:
    a. An average FPS of 144.
    b. A 1% low FPS of no less than 45.
    c. A 0.1% low FPS of no less than 5.
    d. FPS results are averaged over 5 tests, each involving 5 minutes of constantly moving in one direction and generating chunks.
5. To improve visual detail by implementing colours and texture shaders as well as flat shaded lighting. Two examples of what I would deem outstanding visual detail for indie procedurally generated games that utilize flat shading are 'Polytopia' and 'For the King'.

Figure 1: Polytopia, a turn-based strategy game featuring procedurally generated worlds and flat shaded lighting

Figure 2: For the King, a turn-based strategy RPG game with flat shaded lighting

1.4 OUTCOME

I have created a game that consists of a procedurally generated alien planet, tackling some of the prevalent issues of procedural generation mentioned above, while demonstrating its power and effectiveness as a tool for creating a vast and varying game world with limited resources and essentially no budget.

The game includes variations in landscape, different types of terrain, LOD switching, infinite chunk generation, flat shading, water, and textures. Performance is acceptable by the standards set in objective 4. Visual detail is acceptable by the standards set in objective 5.

Overall, the project was successful, all objectives were met, and the aim was fulfilled.

1.5 DISSERTATION STRUCTURE

Introduction

An introduction to the dissertation providing context and detailing the problem, as well as laying out the aims and objectives.

- Context
- The Problem
- Aims & Objectives
- Outcome
- Dissertation Structure

Background & Research

Research into the project area, including a review of existing academic works and a summary of my findings.

- Procedurally Generated Content
- Terrain Generation
- Mesh Generation
- Flat Shading
- Summary

Implementation

A detailed account of how the project was planned, designed and developed.

- Methodology
- Requirements
- Development
- Summary

Testing, Results and Evaluation

The testing methodology and results, followed by an evaluation against the criteria laid down in the objective.

- Evaluation technique
- Testing & Analysis
- Evaluation

Conclusion

Final notes on personal development and continuation of the project

- Personal development
- Continuing the project

2 BACKGROUND & RESEARCH

This section contains my research into the project area, including a review of existing academic works and a summary of my findings.

2.1 PROCEDURALLY GENERATED CONTENT

The first mainstream use of procedural generation in video games were in roguelike games inspired by the tabletop game Dungeons & Dragons. Rogue, the genre's namesake, is a dungeon crawling game created in 1980 by Michael Toy and Glenn Wichman. Rogue's procedural generation system would create dungeons by generating ASCII on a tile-based system to define rooms, hallways, monsters, and treasure. [4]

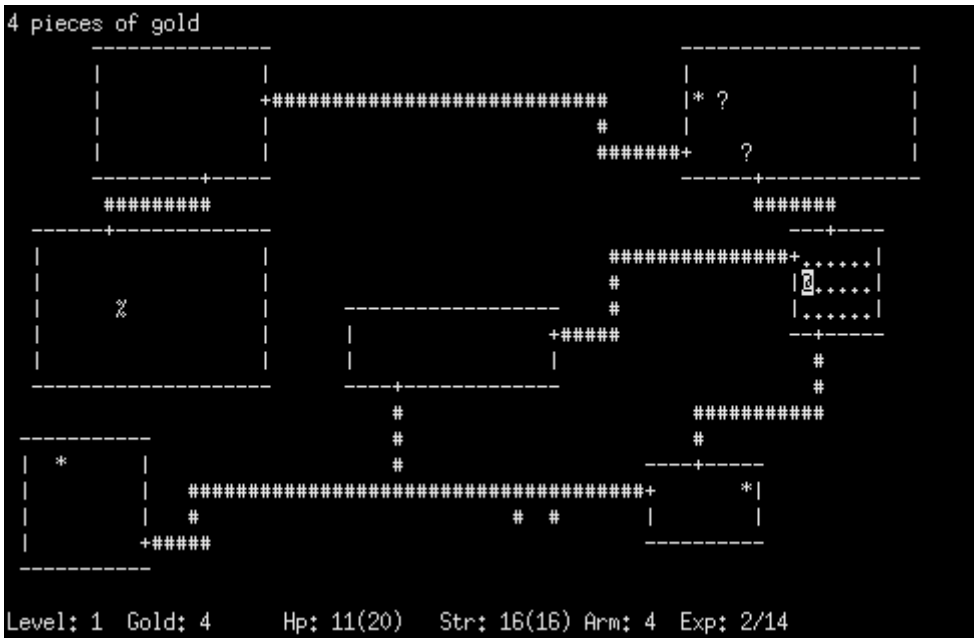


Figure 3: A procedurally-generated dungeon in the 1980 video game Rogue

"Dungeons are intrinsically tied to the history of PCG, showcasing its potential in video games. Players of these types of adventure games and RPGs were introduced early on to the notion of procedural levels and already recognize their value. Rogue (Troy and Wichman, 1980), The Elder Scrolls II: Daggerfall (Bethesda Softworks, 1996), and Diablo (Blizzard Entertainment, 1998) are some of the better known early examples of this relationship between PCG and dungeons."

Procedural generation allowed for the development of complex gameplay while having to deal with the limitations of early-era games consoles, as procedural generating levels or textures can significantly reduce the size of the game's executable.

A good example of this is .kkreiger, a first-person shooter created by demogroup .theprodukkt in 2004. .kkreiger has an extraordinary executable file size of 97 kilobytes.

This was achieved by storing textures via their creation history instead of per-pixel, meaning only the history data and the generator code was needed in the executable. Meshes were created using a box-modelling technique, where basic solids like boxes and cylinders were altered and morphed to

take on the desired shape. Additionally, the audio for the game was produced by a synthesizer that produced music in real time as it was fed a continuous stream of MIDI data.

According to the .kkreiger developers, the game would take around 300MB of space had these innovations not taken place. [5]

Figure 4: .kkreiger, a first person shooter fit into a 96KB executable

While modern computer systems do not provide as harsh technical restrictions to game developers, procedural generation is still heavily used in the games and film industries to reduce development time and project costs, among other reasons.

Massive is a high-end computer animation and artificial intelligence software package used for generating crowd-related visual effects for film and television. It was originally developed for use in Peter Jackson's The Lord of the Rings and was used to generate war scenes containing hundreds of thousands of soldiers automatically. [6] It has since been developed and utilized in films and tv series such as World War Z, Game of Thrones, Hugo, and John Carter.

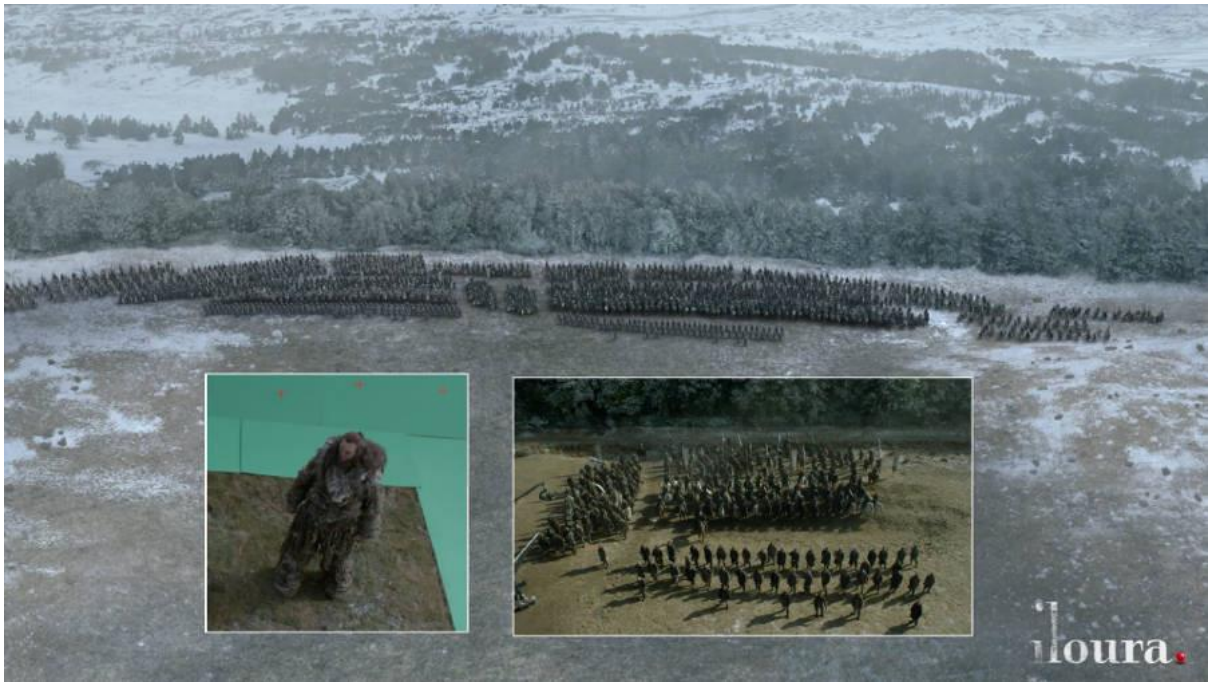


Figure 5: Massive being used to generate crowds in Game of Thrones

In video games, procedural generation can be broken down into two categories [7]:

- Offline
- Online

Offline procedural generation means that content is produced offline, often as a design tool for textures. This form of procedural generation is not relevant to this dissertation, as generating terrain on the fly is a form of online procedural generation.

Online procedural generation means it has been integrated into the game to be used at runtime. The generation process is executed at load time or continuously while the game loads. Common applications of online procedural generation include terrain, vegetation, and certain textures such as lava using a Voronoi tessellation [8]. This is the type I will be implementing into my game.

Mini Metro is a good example of online procedural generation in games. Mini Metro is a puzzle strategy game developed by Dinosaur Polo Club and was released in 2014. Mini Metro's levels are based on real cities and use procedural generation to generate the appearance of stations and passengers. [17] The game also uses a procedural audio system that generates sounds based on the players interactions with the transit network. The sounds change and evolve based on the size and shape of the player's subway system. [18]

Figure 6: A procedurally generated subway system in Mini Metro

2.2 TERRAIN GENERATION

The classic example of infinite procedurally generated terrain is Minecraft by Mojang. Minecraft has a near infinite game world, completely generated through an on-the-fly Perlin noise based algorithm.

“Mojang's game relies on procedural generation, which automatically creates environments and objects that are at once random, but guided by rules that maintain a consistent logic. Mountains are always rocky and sprinkled with snow, for example, while the low lands are typically full of grass and trees.

Minecraft is specifically using Perlin noise calculations, like the kind you'd use to create a rough-looking texture for a 3D model. It starts out on a very broad level, painting a basic topographical map, and adds "noise" through finer terrain details like lakes, shrubbery and animals. Importantly, it has just enough freedom to create unexpected delights, like the elaborate rock structure you see above -- as in the real world, there's an incentive to discover what's just around the bend.” [9]

Figure 7: Procedural terrain generation in Minecraft

Marcus Persson, Minecraft's original creator, describes his process pertaining to using Perlin noise for terrain generation:

*"In the very earliest version of Minecraft, I used a 2D Perlin noise heightmap to set the shape of the world. Or, rather, I used quite a few of them. One for overall elevation, one for terrain roughness, and one for local detail. For each column of blocks, the height was (elevation + (roughness*detail))*64+64. Both elevation and roughness were smooth, large scale noises, and detail was a more intricate one. This method had the great advantage of being very fast as there's just 16*16*(noiseNum) samples per chunk to generate, but the disadvantage of being rather dull. Specifically, there's no way for this method to generate any overhangs.*

So I switched the system over into a similar system based off 3D Perlin noise. Instead of sampling the "ground height", I treated the noise value as the "density", where anything lower than 0 would be air, and anything higher than or equal to 0 would be ground. To make sure the bottom layer is solid and the top isn't, I just add the height (offset by the water level) to the sampled result.

Unfortunately, I immediately ran into both performance issues and playability issues. Performance issues because of the huge amount of sampling needed to be done, and playability issues because there were no flat areas or smooth hills. The solution to both problems turned out to be just sampling at a lower resolution (scaled 8x along the horizontals, 4x along the vertical) and doing a linear interpolation. Suddenly, the game had flat areas, smooth hills, and also most single floating blocks were gone." [10]

Perlin noise was the first known implementation of gradient noise. It is generated by hashing coordinates to correspond to stochastic values, and then interpolating between them [12]. A Perlin noise function can be used to generate a 2D map of values ranging from 0 to 1, which can then be used to generate a heightmap:

“Both noise and most aspects of terrains can fruitfully be represented as twodimensional matrices of real numbers. The width and height of the matrix map to the x and y dimensions of a rectangular surface. In the case of noise, this is called an intensity map, and the values of cells correspond directly to the brightness of the associated pixels. In the case of terrains, the value of each cell corresponds to the height of the terrain (over some baseline) at that point. This is called a heightmap. If the resolution with which the terrain is rendered is greater than the resolution of the heightmap, intermediate points on the ground can simply be interpolated between points that do have specified height values.“ [13]

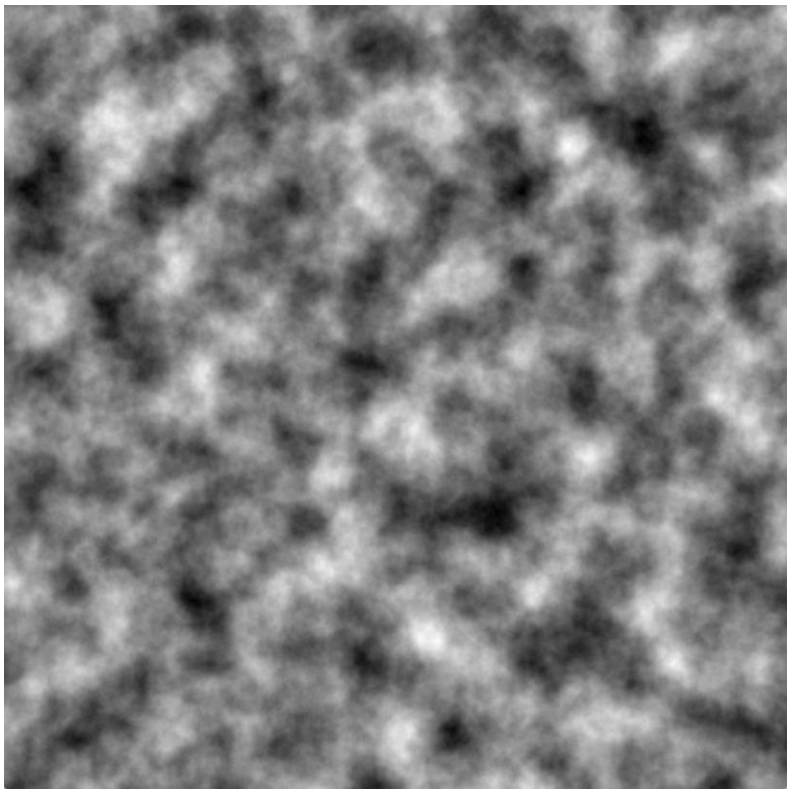


Figure 8: Perlin noise

While procedural generation is random in its nature, it must still be controllable such that the developer can adapt their algorithm to their specific use case:

“A very important element in any procedural method is the control it provides over the output and its properties. We refer to control as the set of options that a designer (or programmer) has in order to purposefully steer the level generation process, as well as the amount of effort that steering takes. Control also determines to which extent editing those options and parameters causes sensible output changes, i.e., the intuitive responsiveness of a generator. Proper control assures that a generator creates consistent results (e.g., playable levels), while maintaining both the set of desired properties and variability.” [11]

“While a procedural terrain generator should create unique maps, which means using enough randomness to avoid duplicating identifiable map features, at the same time this randomness should not lead to a loss of internal structure. Structure in terrain means that the map makes sense and is internally consistent, as opposed to looking as if a random set of features were placed on the map.” [14]

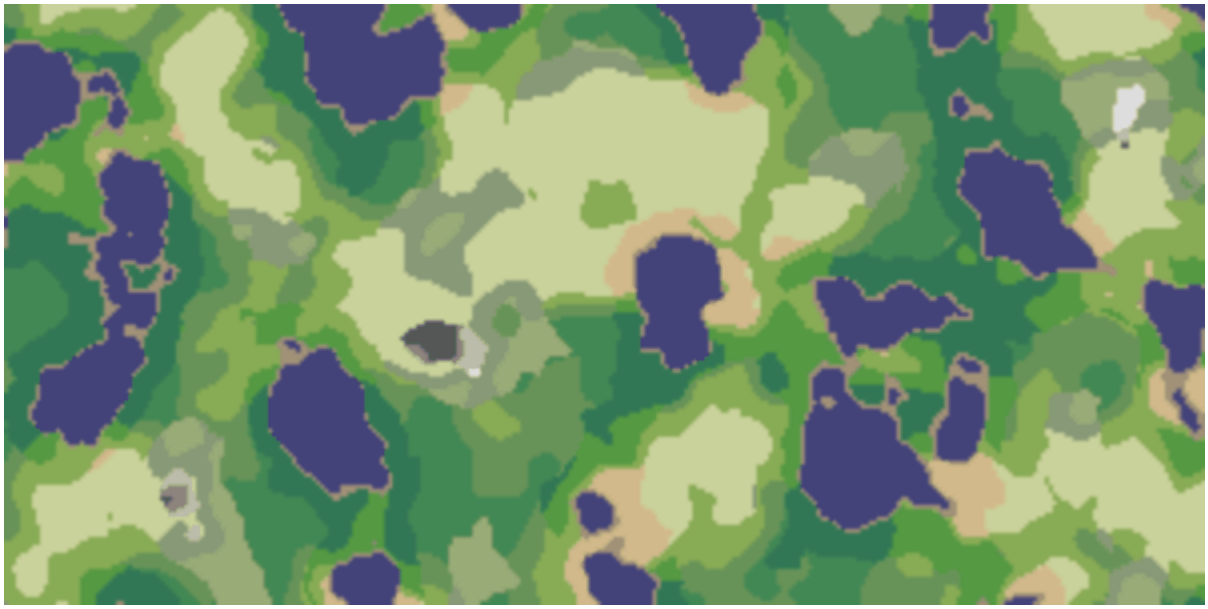


Figure 9: A terrain map generated by Perlin noise

Perlin noise can be expanded upon by introducing the concept of 1/f noise or turbulence. This introduces 4 new values: octaves, persistence, lacunarity and a height scale factor. Octaves allow noise to be stretched and added into itself to create fractal noise. Lacunarity is a number that determines how much detail is added or removed at each octave, while persistence is a number that determines how much each octave contributes to the overall shape by adjusting amplitude. [16]

"Noise at a single frequency is called an octave. The amplitude is multiplied by the persistence (usually 0.5) from one octave to the next. The frequency is multiplied by the lacunarity (usually 2.0) from one octave to the next. For the purposes of terrain generation, y = h(x,z) is multiplied by a scale value and used as the height of the terrain at horizontal point (x,z) for each point (x,z) in the 2D Cartesian plane." [15]

The Perlin noise map can be used as a height map where a mesh can be defined by placing vertices at each point corresponding to the x and z pixels of the noise map, where the y value of each vertex is determined by the Perlin value of that pixel.

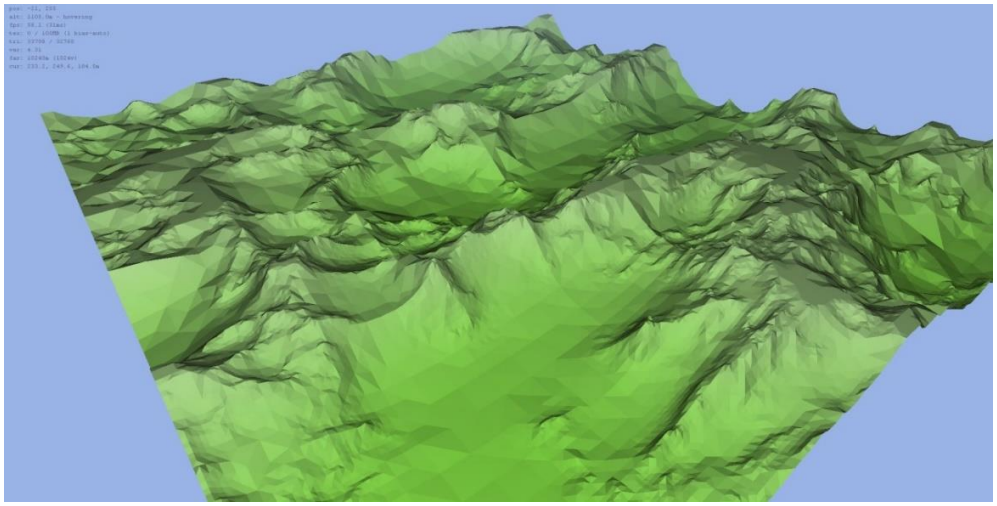


Figure 10: A terrain mesh procedurally generated with Perlin noise

Creating islands with Perlin noise is possible using falloff maps. A falloff map has a range of values ranging from 0 in the centre to 1 around the edges, which can be subtracted from the heightmap to create isolated landmasses in the centre. [19]

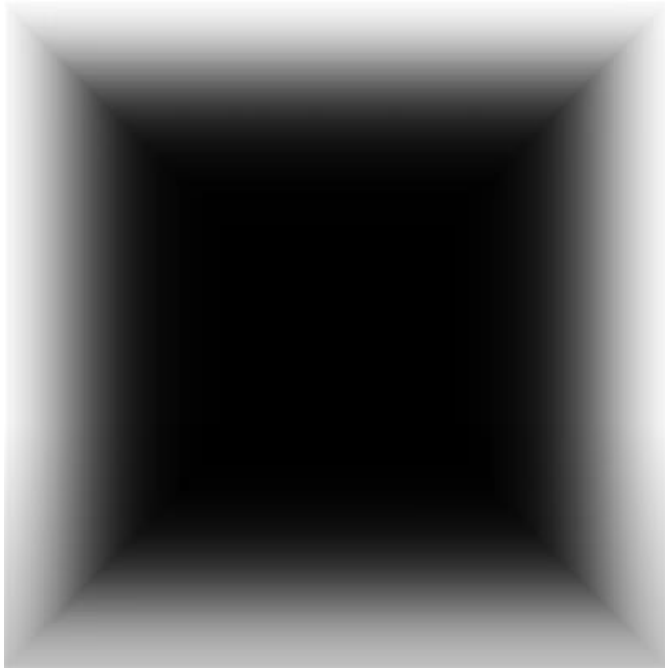


Figure 11: A square falloff map with values ranging from 0 in the centre to 1 around the edges

2.3 MESH GENERATION

In Unity, a mesh requires two components:

- Mesh renderer
- Mesh filter

The mesh filter contains the mesh itself, while the mesh renderer allows us to render the object and apply a texture [23] [24].

"Meshes contain vertices and multiple triangle arrays. The triangle arrays are indices into the vertex arrays; three indices for each triangle. For every vertex there can be a normal, texture coordinates, a color and a tangent. These are optional, and can be removed at will. All vertex information is stored in separate arrays of the same size, so if your mesh has 10 vertices, you would also have 10-size arrays for normals and other attributes.

The mesh face data, i.e. the triangles it is made of, is simply three vertex indices for each triangle. For example, if the mesh has 10 triangles, then the triangles array should be 30 numbers, with each number indicating which vertex to use. The first three elements in the triangles array are the indices for the vertices that make up that triangle; the second three elements make up another triangle and so on." [22]

It is important to consider backface culling. Backface culling is a process that eliminates the back sides of triangles. In other words, if a triangle is not facing in the right direction, it won't be drawn. This is done for performance reasons, as there is no point in wasting system resources to draw the inside of the object if it won't ever be seen by the player. Unity determines the direction in which a triangle is facing by the order in which the triangle's vertices are provided. In unity, triangles are drawn clockwise. Counter-clockwise drawn triangles will not be rendered unless looked at from the rear side (or inside) of the mesh. [25]

The number of vertices in a mesh is: $vertices = width * height$.

The number of triangles in a mesh is determined by the number of squares in the mesh, multiplied by 2 as there were two triangles per square, multiplied by 3 as there were three vertices per triangle. The final formula is: $triangles = 6(width - 1)(height - 1)$.

2.4 FLAT SHADING

Flat shading is a lighting technique where the lighting is evaluated based on the polygon's surface normal under the assumption that the polygons are flat. The computed colour is used for the whole polygon, making the corners look sharp. This is a relatively simple and inexpensive (with regards to system resources) technique to employ, while providing a unique cartoon like aesthetic (See figures 1, 2 and 12).

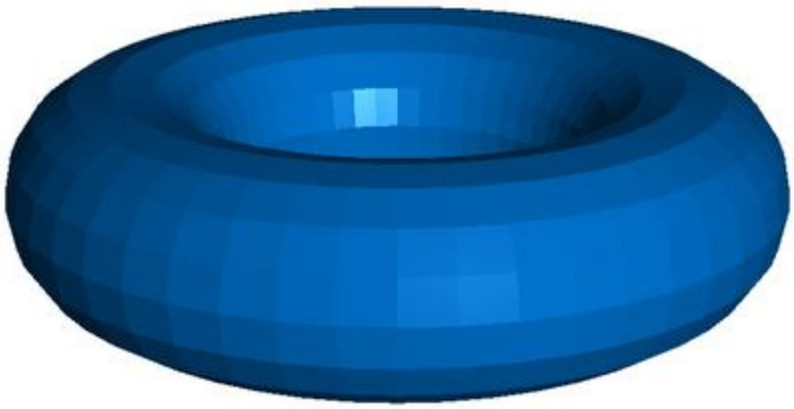


Figure 12: A flat shaded toroid

The significant limitation with flat shading is that the polygon must be flat, which may not always be true for polygons with more than three edges. [20] Furthermore, all the normals of the polygon must be pointing in the same direction. [21] This would mean that each vertex of each polygon must be unique to that polygon, i.e. no two polygons share a vertex.

2.5 SUMMARY

From this research, we can conclude that random terrain generation noise function must:

- Be versatile. Bland terrain generation can cause repetitive and empty feeling worlds of no substance. The noise function should be effective as a tool for implement variation in terrain.
- Be fast. Procedural generation techniques are relatively expensive, and such performance is a large factor to investigate. The noise function used must be efficient and not very performance impacting.
- Be controllable. In order to morph the terrain according to the designer's vision, the noise function must be fully controllable.

The findings of my research provide multiple sources that indicate Perlin noise is an effective and versatile took for procedurally generating terrain. Procedural generation is fast and performant, while being simple to use and implement. With the introduction of octaves and other variables, Perlin noise can be morphed into different shapes and sizes, making it an effective tool for implementing topographic variation. Certain erosion techniques can be applied to further customize the terrain generated.

These factors make Perlin noise a sufficient tool for completing the objectives set for the project, as it would allow for objectives 2 (to use Perlin noise to create variation in topography, including biomes and typical terrain features), 3 (to implement infinite, repeated and pseudo-randomly generated chunks of terrain) and 4 (to ensure an acceptable level of performance by utilizing techniques such as multithreading and variable levels of detail) to be achievable.

Unity provides a quality array of libraries for implementing the solution of this project. Unity has a built-in Perlin noise function that can return 2D Perlin noise efficiently. Furthermore, Unity has an intuitive system for creating custom meshes programmatically. This has two advantages:

- Creating custom meshes with a Perlin noise based height map is possible and easy to implement
- Implementing the necessary vertex adjustments for flat shading is possible and easy to implement

As a result, using Unity would make objectives 1 (to create a chunk of terrain using a triangle mesh) and 5 (to improve visual detail by implementing colours and texture shaders as well as flat shaded lighting) achievable objectives.

3 IMPLEMENTATION

This section will provide a detailed account of how the project was planned, designed, and developed.

3.1 METHODOLOGY

In order to organize myself and set personal deadlines for my workflow, I created a Trello board and a Gantt chart. I used the Gantt chart as a timing tool and the Trello board as a way to get organized and set specific tasks and sub-tasks for myself to do each week.

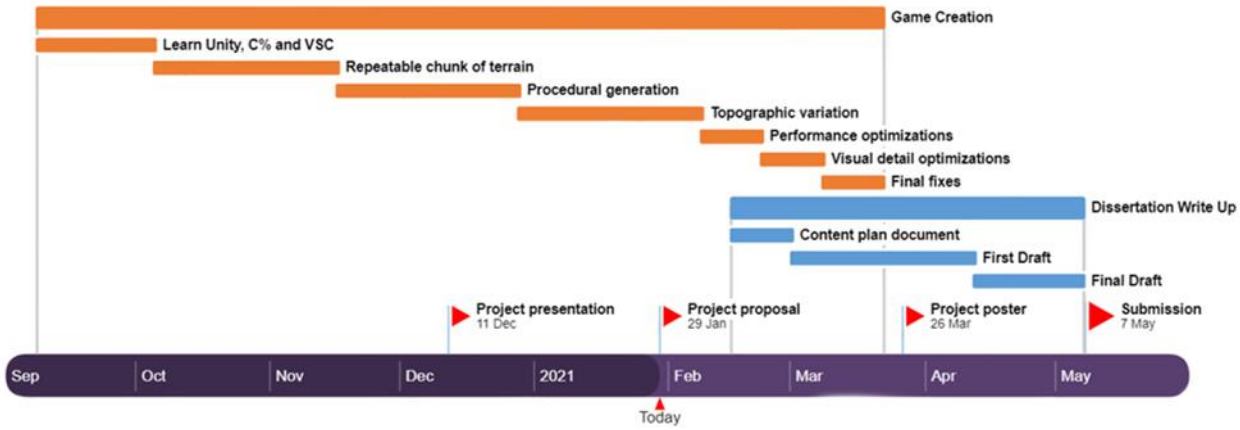


Figure 13: Project Gantt chart

In general, the project development process was smooth. While I aimed to finish it in March, it was completed in late February. This left me more time to plan and write-up the dissertation.

There were quite a few hiccups on the project, however. I struggled with implementing threading, as it was the first time I had ever tried to do so. I also struggled with the first draft of the dissertation write-up, as I had never written a document this long. However, a solution to both problems were found with the help of my supervisor group and the project was continued as planned.

Overall, I am happy with how I planned my time and would only make minor changes in hindsight pertaining to the time allocated for certain tasks in the project, as well as elicit more advice from my project supervisors on the dissertation write-up.

3.2 REQUIREMENTS

3.2.1 FUNCTIONAL REQUIREMENTS

1. World Generation
    a. The player will be able to travel an infinite distance in any direction.
    b. The player will never be in a chunk that is the same as one they have come across before.
    c. The player can experience a different biome per world.
    d. Seamless chunk transitions.
2. Code

    a. Code must be commented and well laid out.
3. Optimizations
    a. The performance of the game must be of an adequate level according to the criteria set in objective 4 of this project.
    b. The game must have some form of colour to represent what kind of terrain the player is in.

3.2.2 NON-FUNCTIONAL REQUIREMENTS

1. Graphics
    a. Flat shaded lighting.
    b. Textures as opposed to colours that can more accurately represent what kind of terrain the player is in.
2. Biome specific features
    a. Trees, bushes, flowers, and other shrubbery.
    b. Rocks, stones, and other geological features.
    c. Other non-terrain related features.
    d. Water implementations such as oceans, rivers, and lakes.
3. Biomes
    a. Multiple biomes on the same planet.
    b. Smooth transitions between biomes.
4. Code
    a. Code must be factored such that it is legible to outside developers.

3.2.3 MOSCOW PRIORITIZATION

MoSCoW prioritization is a workflow prioritization framework for time-boxed projects. It will allow me to prioritize certain tasks and features over others, ensuring a smoother workflow. Green items are a must, yellow items should be implemented, and red items could be implemented, each having descending priority.

Table of Requirements	
Requirement	**Description**
Unity	The game requires a game engine to be built upon, and Unity provides many tools and libraries necessary to develop this project
Visual Studio	Visual studio has significant integration into the Unity game engine and will be the best choice for an IDE for this project
C#	Unity scripts are written using the C# programming language, and thus the scripts necessary for the project must be written in C#
Noise generator	Noise is required to generate terrain in a pseudo-random manner
Mesh generator	A mesh must be created based off the noise to represent the terrain of the alien planet
Endless terrain generation	A terrain generation script must exist to create chunks of terrain to simulate the surface of a planet
Topographic variation	The topography of the planet should change as you move in a direction
Flat shaded lighting	Flat shaded lighting should enhance the visual aesthetic of the game
Colours or Textures	Colours or Textures should help to differentiate between different biomes and terrain features
Cg	The Cg language is one of the languages supported by Unity for writing shader programs, which will be needed for applying texture or colour shaders to the mesh
Threading	Threading would significantly increase the efficiency and performance of the game
LOD optimizations	The level of detail should be reduced for chunks farther away so as not to waste system resources
Seamless LOD transitions	Seamless transitions between levels of detail could help the world look smoother
Seamless biome transitions	Seamless transitions between different biomes could help the world look smoother
Water	Water would increase the realism of certain terrains that contain it
Falloff map	Falloff maps would help when generating islands
Collisions	Collisions would allow a player character to roam around the game
Biome specific features	Trees, shrubbery, rocks, and other non-terrain related features could enhance the players ability to differentiate between biomes

3.3 DEVELOPMENT

3.3.1 GENERATING THE HEIGHT MAP

In order to generate the height map for the mesh, I needed to generate some height values. For this, I created a script that uses Unity's built in math library to generate Perlin noise and returned a 2D float array of values ranging from 0 to 1 that would represent the height of the terrain at that coordinate.

The initial version of this script took in the desired width and height as well as the scale of the noise to generate a height map.

```
public static float[,] GenerateHeightMap(int width, int height, int scale)
{
    float[,] heightMap = new float[width, height];

    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            heightMap[x, y] = Mathf.PerlinNoise((x / scale), (y / scale));
        }
    }

    return heightMap;
}
```

Figure 14: Noise generator script v1

After this, I created a plane in the Unity scene to display the heightmap by linearly interpolating between white and black based on the value at that point in the heightmap and drawing it onto the plane. See figure 14 for an output with a scale of 20. A scale of 20 will be used throughout this section so that different version of the noise generator script can be compared consistently.

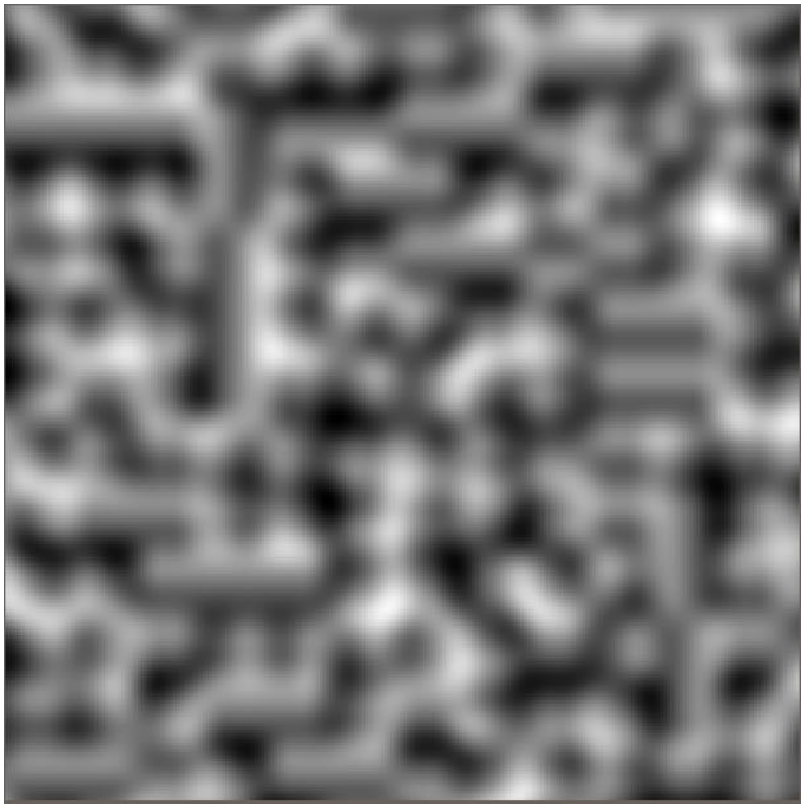


Figure 15: Noise generator script v1 output

I then proceeded to implement octaves, persistence and lacunarity modifiers to the Perlin noise according to my research above. This was done to increase my control over the noise and increase its realism with regards to terrain generation.

In order to do this, I added in amplitude and frequency variables as described by Parberry [15]. As amplitude multiplies the height value at a point, I had to inversely interpolate the values back into the range of 0 to 1 as some valued may exceed this range.

Furthermore, the script would take in a seed that allow the generation of completely different worlds per seed. This works by offsetting the value used when sampling the Perlin noise by a large value determined by a pseudo-random number generator. This is the same approach Minecraft uses to generate new random worlds each time.

This version of the script allowed for very different looking height maps to be generated, which would help later when implementing topographic variation. See figure 15 for an output with a scale of 20, 6 octaves, a persistence of 0.5 and a lacunarity value of 2. These were the values suggested by Parberry as a starting point [15].

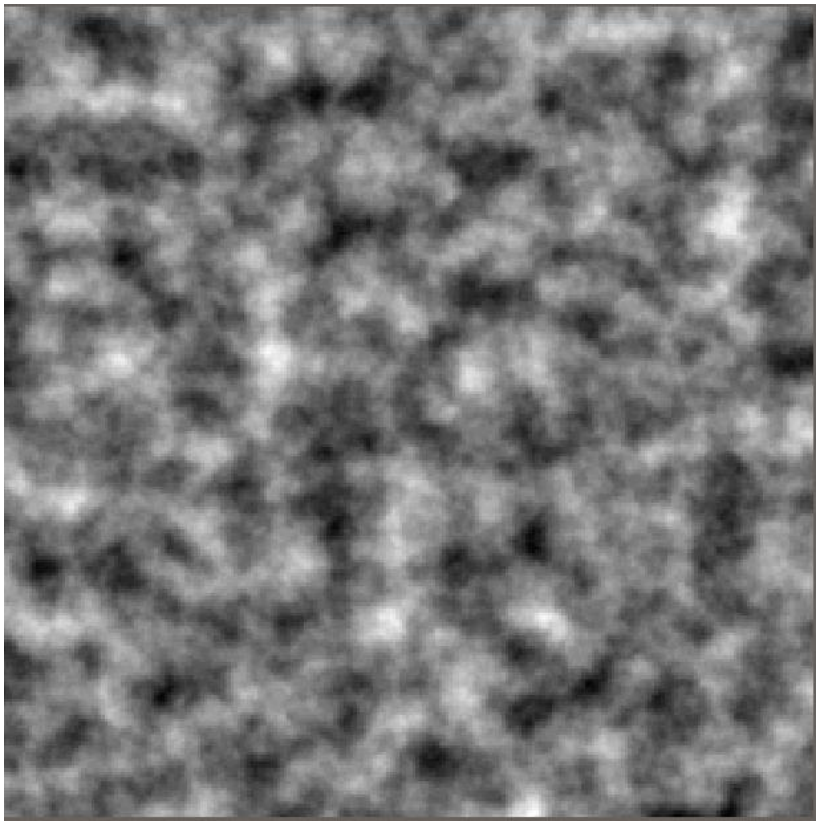


Figure 16: Noise generator script v2 output

The final addition to the height map was the option to add a falloff map if I wanted to generate islands. For this, I created another script that would generate a falloff map and would apply it to the height map if the option were selected.

The first version of the script worked by looking at which axis's absolute value is larger (x or y) and setting the height at that coordinate to that axis's absolute value for each coordinate.

```
public static float[,] GenerateFalloffMap(int size)
{
    float[,] falloffMap = new float[size, size];

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            float x = i / (float) size * 2 - 1;
            float y = j / (float) size * 2 - 1;

            float value = Mathf.Max(Mathf.Abs(x), Mathf.Abs(y));
            falloffMap[i, j] = value;
        }
    }
    return falloffMap;
}
```

Figure 17: Falloff map generator v1

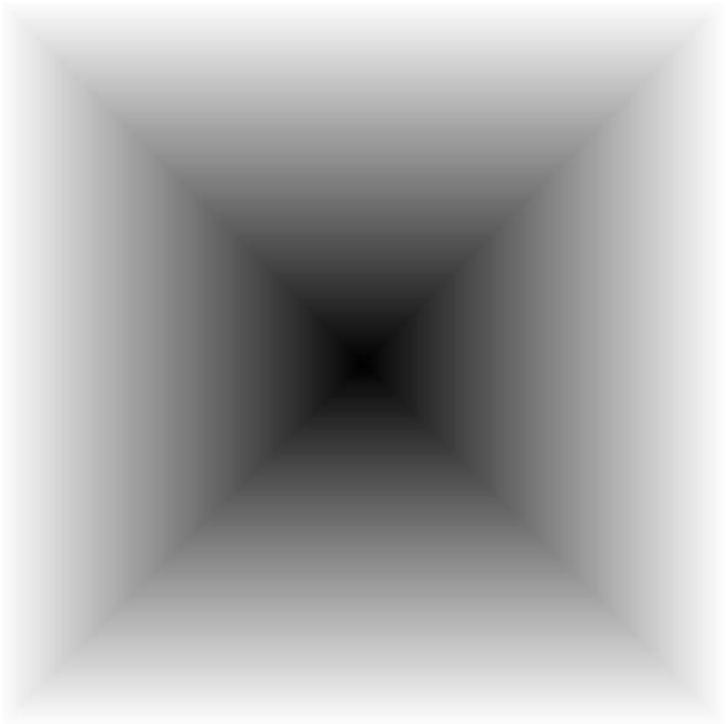


Figure 18: Falloff map generator v1 output

However, this implementation does not allow for the scalability of the size or smoothness of the islands. In order to allow for more adjustability, I added two more variables: falloffModifierA and falloffModifierB. These values would then be used to evaluate the value in the falloff map at that point according to the function $f(x) = \frac{x^a}{x^a+(b-bx)^a}$ [19].

This means that increasing falloffModifierA would decrease the gradient slope of the falloff map and increasing falloffModifierB would increase the size of the falloff map. Therefore, falloffModifierA would adjust the smoothness of the transition between ocean and land, while falloffModifierB would adjust the size of the island.

```
public static class Noise
{
    0 references
    public static float[,] GenerateFalloffMap(int size, float falloffModifierA, float falloffModifierB) {
        float[,] map = new float[size,size];

        for(int i = 0; i < size; i++) {
            for(int j = 0; j < size; j++) {
                float x = i / (float)size * 2 - 1;
                float y = j / (float)size * 2 - 1;

                float value = Mathf.Max(Mathf.Abs(x), Mathf.Abs(y));

                float a = falloffModifierA;
                float b = falloffModifierB;

                map[i, j] = Mathf.Pow(value, a) / (Mathf.Pow(value, a) + Mathf.Pow(b - b * value, a));
            }
        }
        return map;
    }
}
```

Figure 19: Falloff map generator v2

The falloff map would then be applied to the heightmap by subtracting the falloff value from the heightmap value at each coordinate.

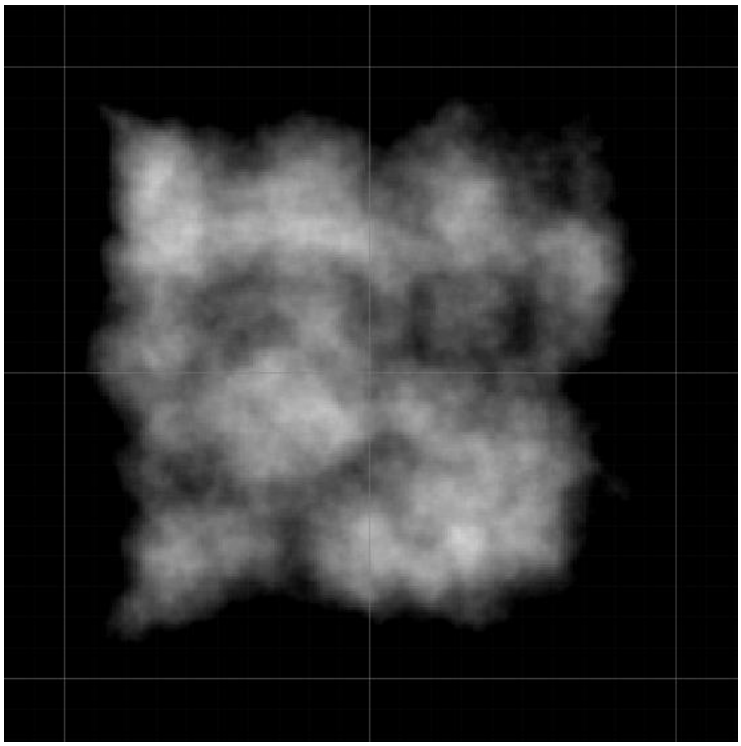


Figure 20: Height map with falloff

At this point I was satisfied with the quality of the noise generated and decided to continue on to generating the mesh.

3.3.2 GENERATING THE MESH

Now that the height map was generated, it was possible to start generating the mesh. I prioritized making the mesh as controllable and customizable as possible, as my research indicates it would make creating topographic variation easier later on.

I created a script to generate meshes that would take in a heightmap as well as its size. The script generates triangles one square at a time by creating two triangles per loop in a clockwise manner. See figure 20 for a visualization of this.

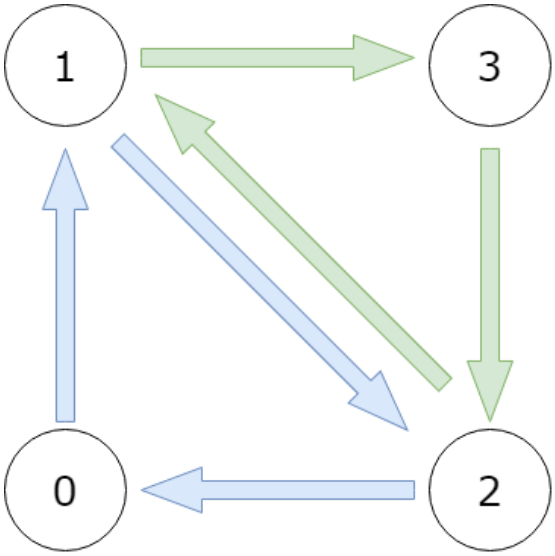


Figure 21: Creating triangles clockwise

If creating a 1x1 mesh like in figure 20, the vertices array should look like:

$$vertices = \{(0,0), (0,1), (1,0), (1,1)\}$$

And the triangles array would look like:

$$triangles = \{0, 1, 2, 2, 1, 3\}$$

The size of these arrays match what they should be based on my research. Note that the arrays used for my mesh have 3 dimensions, with the y coordinate corresponding to its respective value in the height map.

Finally, in addition to generating the vertex and triangle arrays, I generated a UV array for use later when applying textures. The UV array is an array of 2D vectors with each vector's coordinates being $\frac{x}{width}$ and $\frac{y}{height}$.

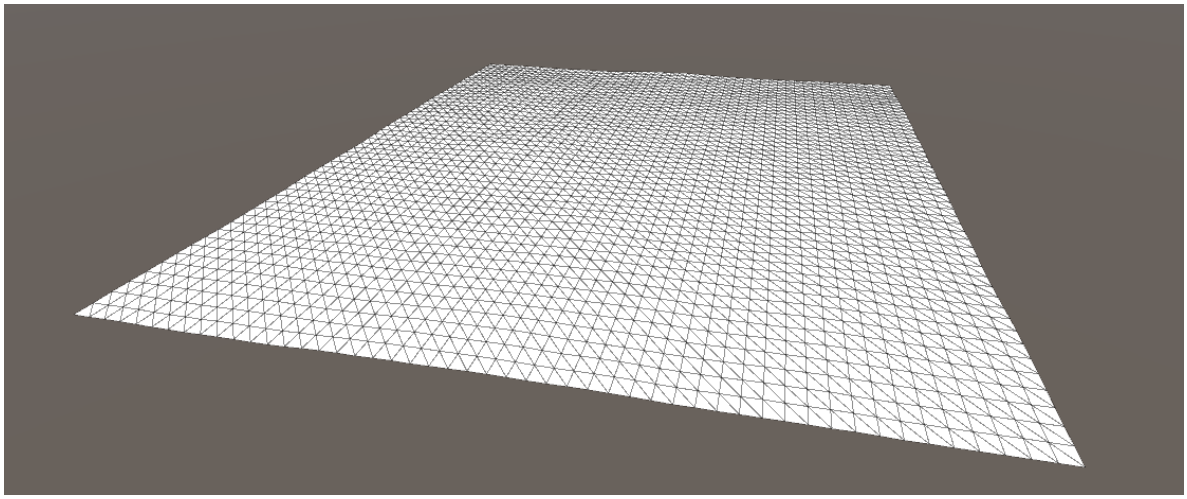


Figure 22: Mesh generator v1 output

As can be seen in figure 21, the mesh is still quite flat. In order to solve this issue, I added a height multiplier to the noise script to further exaggerate the difference in height between coordinates. This would simply multiply the y value of each vertex when generating the height map. See figure 22 for a mesh generated with a height multiplier of 20.

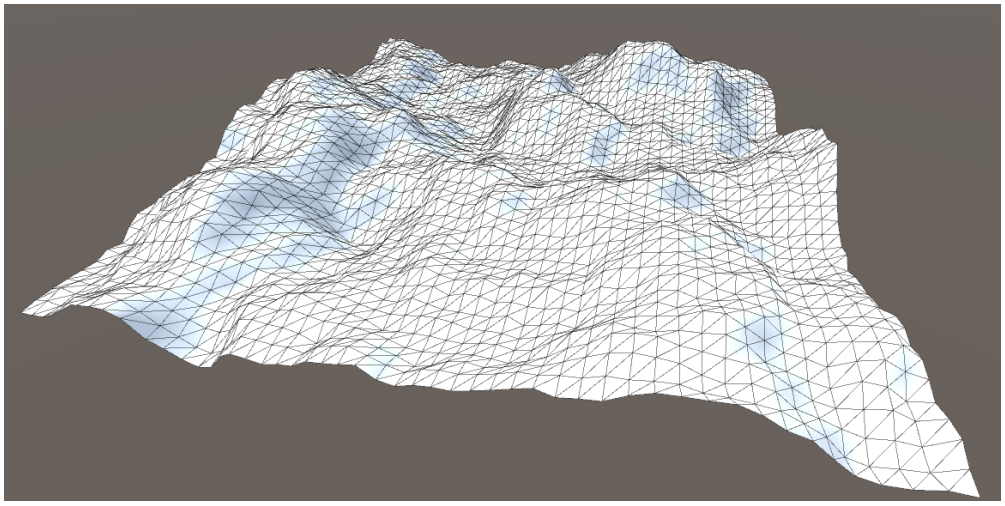


Figure 23: Mesh generator v2 output

The limitation of having a static height multiplier is that there is no control over the gradient of change of the heights of each vertex. My solution to this problem was to implement a customizable height curve to the mesh generator script.

It is important to note that this multiplier directly multiplies the mesh, whereas the height multiplier implemented previously multiplies the noise. I implemented it this way so that there is more control of the topography with the height curve, while the noise multiplier can act as a constant scale factor. See figures 23 and 24 for examples where I use the height curve and height modifier to create peaky or flatter terrain, respectively.

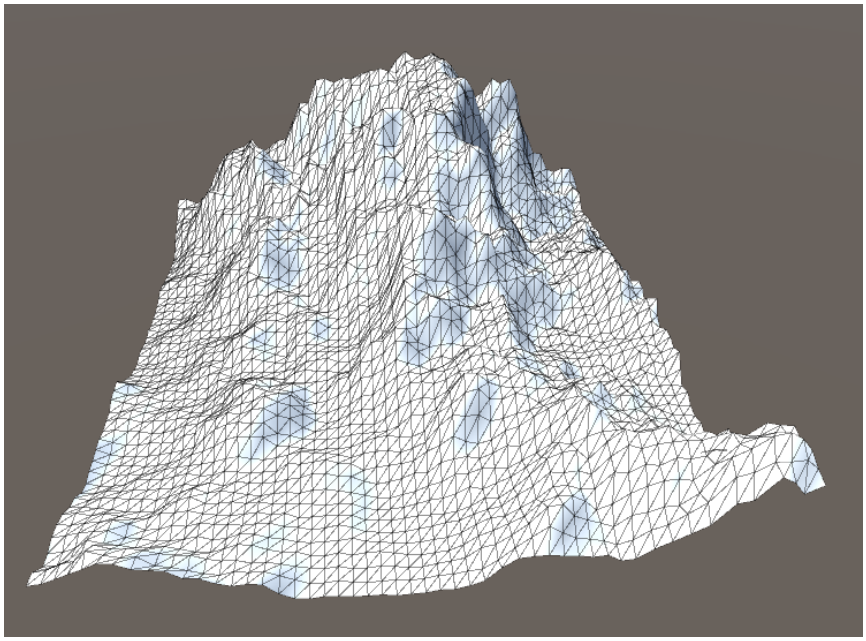


Figure 24: Peaky mountain terrain

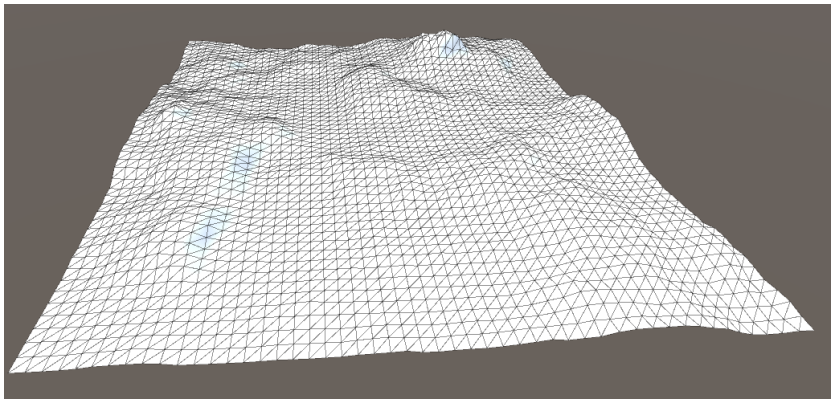


Figure 25: Flatter plains terrain

The final addition to my meshes was adding a collider. This was simple, as all it involved was creating a collider based on the mesh and adding it to the same game object that contains the mesh renderer and mesh filter.

At this point, I was happy with the customizability of the mesh and decided to move on.

3.3.3 GENERATING ENDLESS TERRAIN

After creating the mesh, I then made it so that the meshes would generate endlessly when moving in a direction. Note that throughout this section, I refer to each terrain mesh as a chunk.

The first thing I did was define a max view distance in units. This would make the terrain generator only generate new chunks if those new chunks would be in the view distance of the player. That implies that the number of chunks visible in the players view distance would be:

$$visibleChunks = \frac{viewDistance}{chunkSize}$$

The next step would be to set up a map coordinate system. The player will spawn in the centre of the first chunk, which would be at the coordinate (0,0). The chunk directly east of this would have the coordinate of (chunkSize,0). For simplicity, I used the chunk size as the unit of measurement, making this chunk at position (1,0).

The system works such that if a player has a view distance of 512 and the size of each chunk is 128, the player will see 4 chunks in each direction if they were standing on the intersection of 4 chunks. If the player were standing in the centre of a chunk, they would only see 4.5 chunks in each direction (including half of the chunk they are standing on). This is because the terrain generator checks the distance between the player's position and the nearest edge of the chunk, and if that distance is less than the view distance, it will generate that chunk.

Finally, for each chunk within the players view distance, a mesh would be generated. If the player walks out of range of a chunk, that chunk is set as not active and is not rendered. If the player walks back into range, it is set back to active. Each generated chunk is kept in a dictionary. Each game loop, for each map coordinate within range of the player, the chunk is generated or updated accordingly.

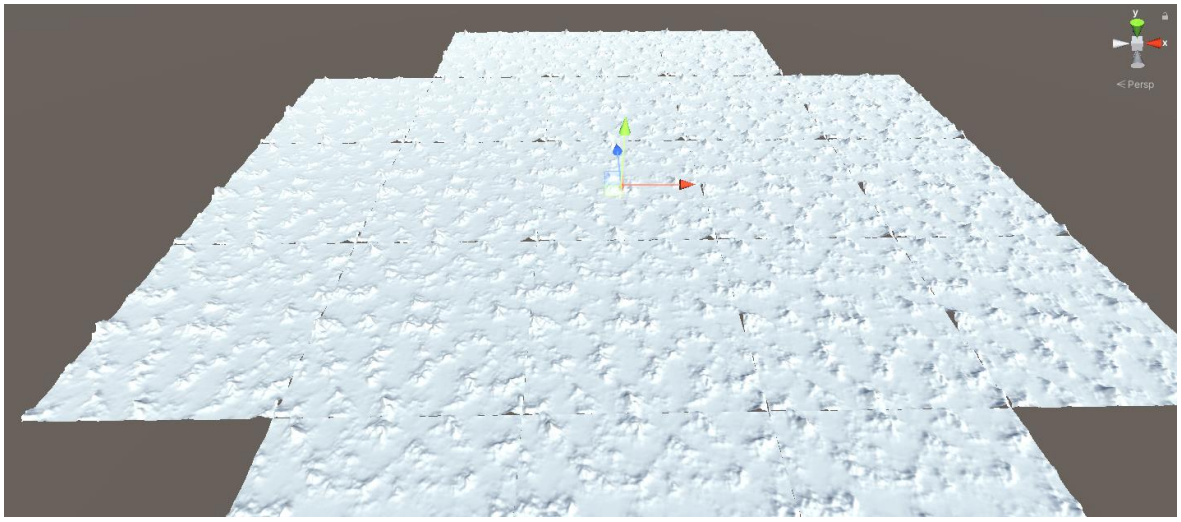


Figure 26: Endless terrain script v1

There are a few things to note at this point:

- Things that work
    - Chunks are rendered only in the players view distance
    - Meshes generate, unloads and re-loads appropriately
- Things that need fixing
    - Each chunk has the exact same mesh
    - The seams between chunks are not smooth

The first thing I decided to fix was each chunk having the same mesh. This was fixed by modifying the noise generator such that it adds an offset to the sampling point of the Perlin noise function depending on its map coordinate. For example, if chunk (0,0) sampled starting at point (0,0) in the Perlin noise function, chunk (1,0) would sample starting at point (chunkSize,0) in the Perlin noise function. The chunk size and map coordinate are now passed into the noise generator to accomplish this.

The final fix was ensuring smooth transitions between chunks by removing the seams. This issue exists because of the way we inversely interpolated the height values of the noise in the noise generator. The range for the interpolation is the maximum and minimum values for any point in that mesh. However, those maximum and minimum values will be different from mesh to mesh.

The solution to this problem was to calculate the maximum possible height value that the noise generator could output, then inversely interpolate all meshes using the maximum possible height. The only issue with this solution is that the heights generated by the noise generator almost never reach that value, which would flatten the mesh. I solved this by dividing the maximum height value by 1.7. This number was chosen after randomly trying numbers and stopping at what felt acceptable to me.

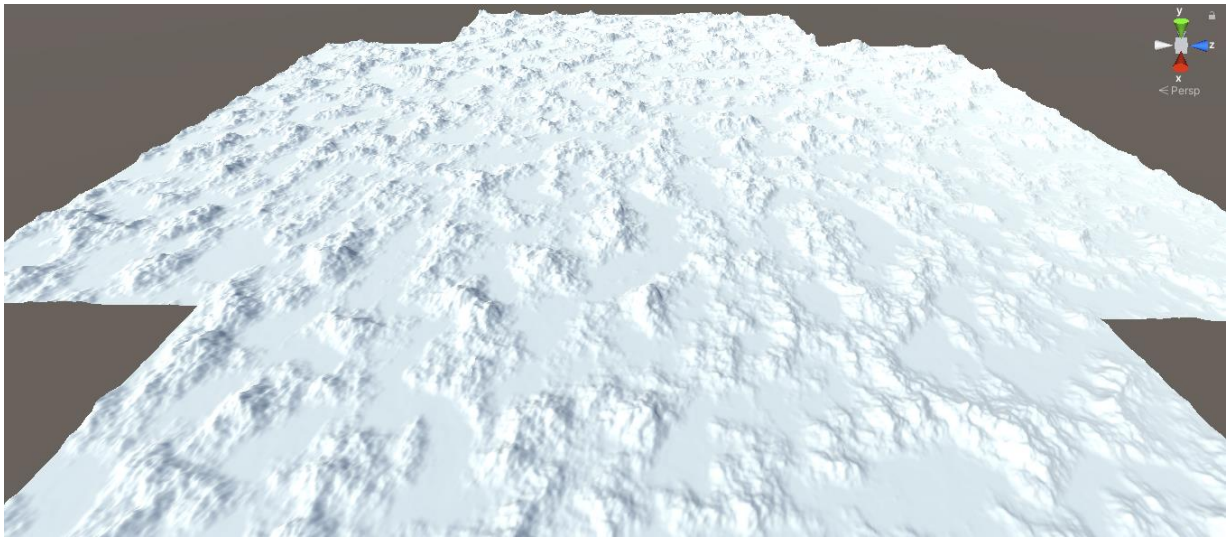


Figure 27: Endless terrain script v2

Now that the endless terrain generator worked, I decided to spruce up the aesthetic of the game.

3.3.4 OPTIMIZING AESTHETIC

At this point, the game looked bland as everything was just white and bumpy. Because of this, I decided to implement the following features:

- A texture shader. Terrain needs textures in order to tell the player what kind of biome they are in.
- Flat shaded lighting. As described in my research section, it is a low-cost lighting system with a cartoon-like appearance.
- Water. Some biomes feature water as a feature, such as oceans or plains biomes with lakes.

In order to add textures to the game, I decided to write a custom shader. The reason I created my own shader was so that I could change the texture used depending on the height of the terrain at that point. I was also inspired by some tutorial videos I watched on YouTube. [26] [27]

I began by creating a serializable class that would hold all the data the texture shader would need for a layer. It is serializable so that I could create and edit layers from the Unity inspector.

```
    [System.Serializable]
    1 reference
    public class Layer
    {
        public Texture2D texture;
        public float textureScale;
        [Range(0, 1)]
        public float startHeight;

        public Color tint;
        [Range(0,1)]
        public float tintStrength;
        [Range(0, 1)]
        public float blendStrength;

    }
}
```

Figure 28: Layer class for texture shader

The struct contains the texture to be used and some settings for it, optionally a colour for tint as well as the starting height that the texture/colour would be drawn from (starting at 0 and going up).

Next, I created a class for managing textures that could send this information to a shader, as well as providing the shader with the minimum and maximum heights from the height map. These heights were used to blend textures from different layers together by inversely interpolating them to get a smooth gradient.

The shader then drew textures on the mesh depending on the steepness of the mesh at that square. This was done to avoid the texture looking stretched. A good way to visualise this is by imagining placing a sheet of paper on a cube as opposed to placing a smaller piece of paper on each face of the cube depending on which direction its facing.

Finally, colours acted as an albedo on top of the texture. This was implemented to provide different tints to textures. This would allow me to have dark green and light green grass while using the same texture.

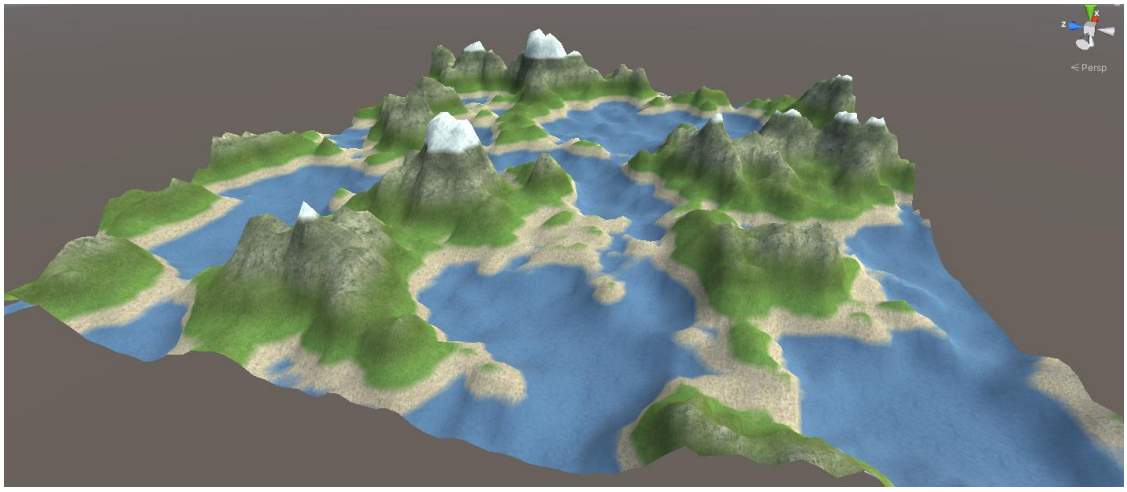


Figure 29: Terrain with texture shader

The next feature to add was flat shaded lighting. In order to do this, I had to adjust the mesh generation such that all the normals for each vertex in a triangle are facing the same direction, as discussed in the research section.

This is not possible if a vertex is shared by multiple triangles, as the normal of that vertex would be an average of all the triangles. Therefore, each triangle must have vertices that only it uses.

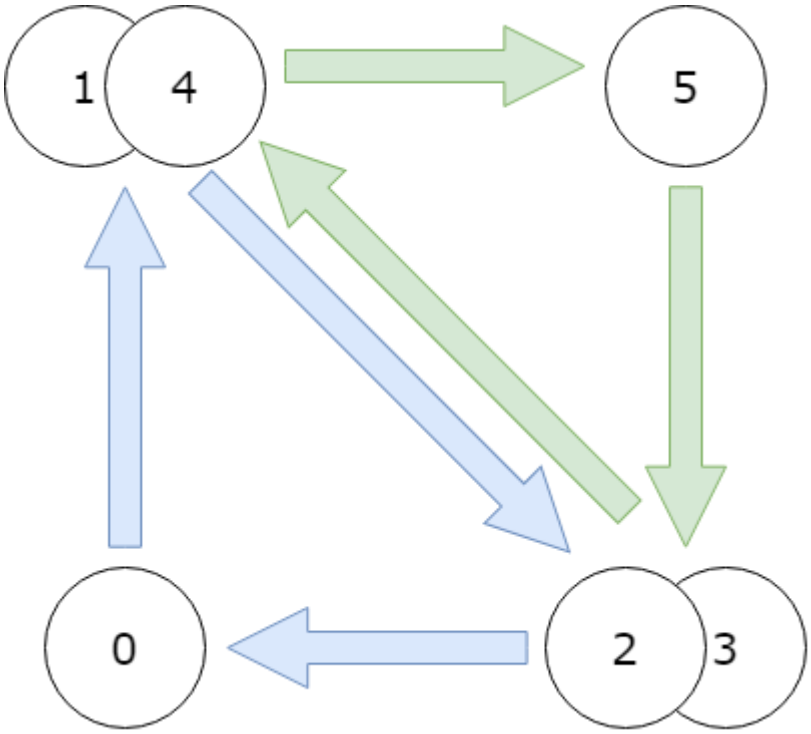


Figure 30: Creating triangles clockwise with adjusted normals for flat shading

If creating a 1x1 mesh like in figure 20, the vertices array should look like:

$$vertices = \{(0,0), (0,1), (1,0), (0,1), (1,0), (1,1)\}$$

And the triangles array would look like:

$$triangles = \{0,1,2,3,4,5\}$$

Note that the size of the triangle array stays the same, while the length of the vertex and UV arrays become the same length as the triangle array.

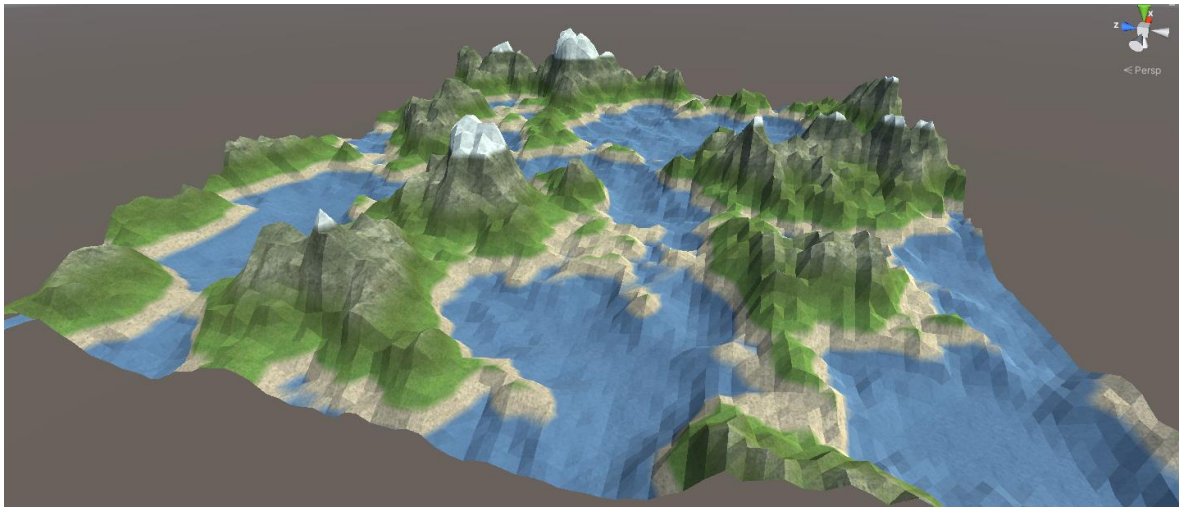


Figure 31: Terrain with flat shading

The final aesthetic change was the addition of water. I simply added in a water prefab from the Unity standard assets pack and set its radius to be the view distance of the player. I added a variable that allowed for the worlds water level to be set and made the water prefab move with the player's X and Z coordinates.

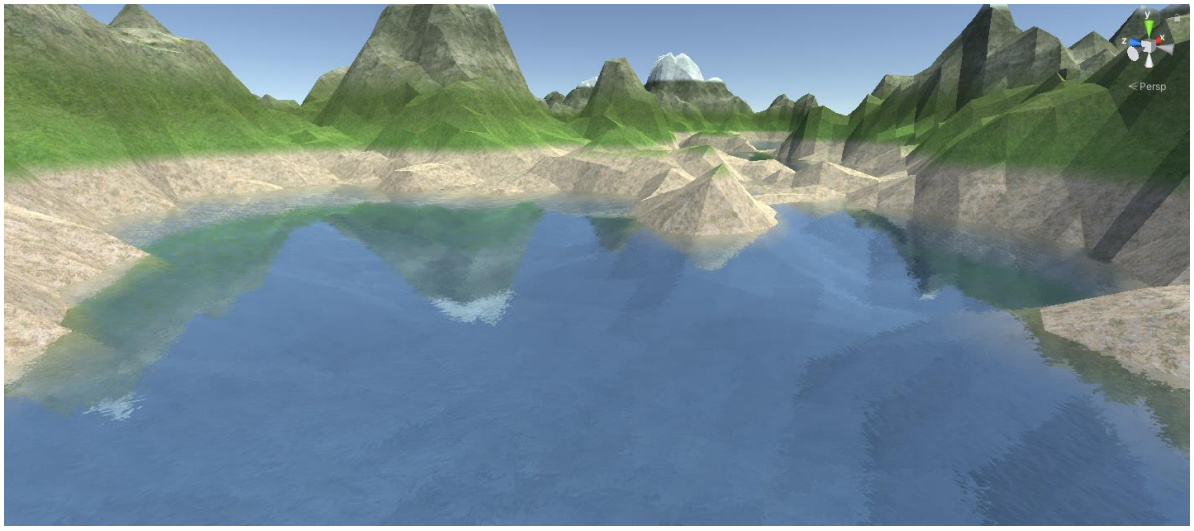


Figure 32: Terrain with water

I was satisfied with the look of the terrain at this point but was starting to notice a drop in performance while running the game. Therefore, I decided to move on to optimizing performance.

3.3.5 OPTIMIZING PERFORMANCE

There were several performance optimizations I had in mind:

- Level of detail (LOD) scaling: The level of detail for chunks farther out from the player did not need to be rendered in the same quality as chunks nearby, as the player would not be able to see the difference in detail from so far out.

- Threading: At this point, everything was performed on a single thread. There should be threaded functionality to perform tasks simultaneously and in order using threads and queues.
- Mesh collider optimizations: I current experience large lag spikes when generating new chunks. According to the Unity profiler, this is coming from the script generating the colliders.
- Chunk update optimizations: The current algorithms for updating chunks are quite expensive.

I decided to begin with adding different levels of detail. Meshes generated farther from the player don't need to be as high resolution as they are too far away to see clearly, thus they require less triangles. In order to reduce the triangles in the mesh while keeping it the same size, we can skip a vertex when drawing the triangles.

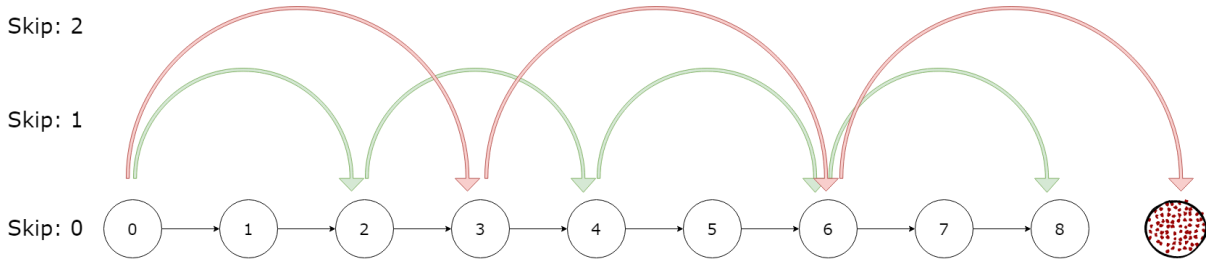


Figure 33: Skipping vertices for LOD adjustments

As you can see in figure 32, the width of this mesh is eight. Two is a factor of eight, and therefore skipping one vertex works. However, three is not a factor of eight, so skipping two vertices does not work. From this, we can deduce that a suitable chunk size would be one that has at least all numbers up to four as a factor.

Another constraint on the mesh size is Unity's 65,536 vertex limit per mesh, leaving 256x256 as our maximum mesh size without flat shading. As we are using flat shading, we must consider that each triangle uses unique vertices. That means for each square we have 6 vertices. This leaves us with a maximum mesh size of about 96x96. That leaves us with three supported mesh sizes that fit all criteria: 48, 72 and 96.

Armed with this knowledge, I created a variable called the skip increment inside the mesh generator that would skip that number of vertices when creating the mesh.

Note that the collider for the chunk is based off the mesh and will therefore have the same resolution as the mesh.

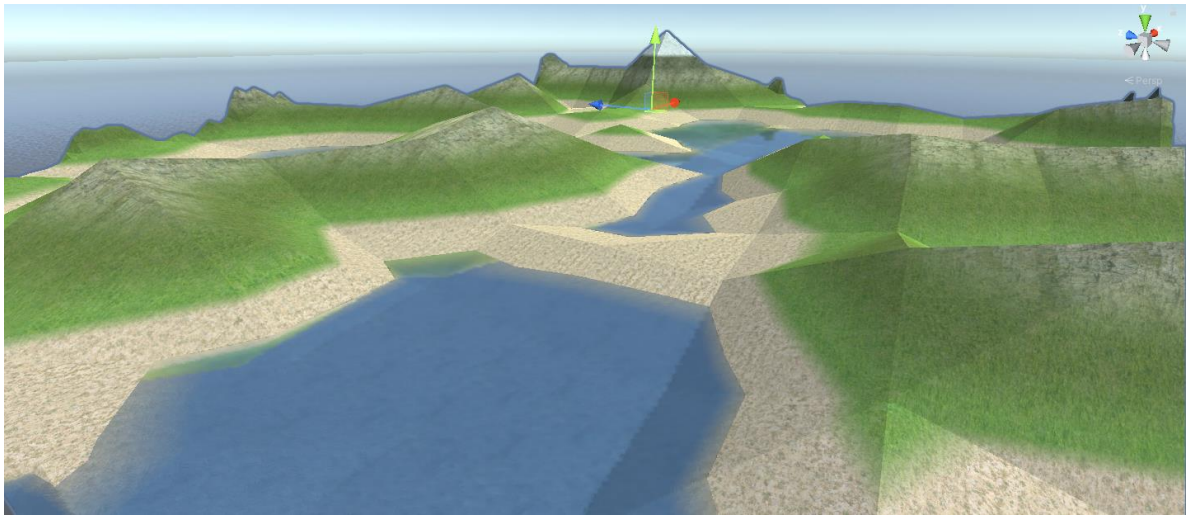


Figure 34: The same mesh as in figure 30 but with an LOD skip increment of 4

The next step would be to implement this in the endless terrain generator. To do this, I set certain view distances that the level of detail would drop depending on the players distance to them.

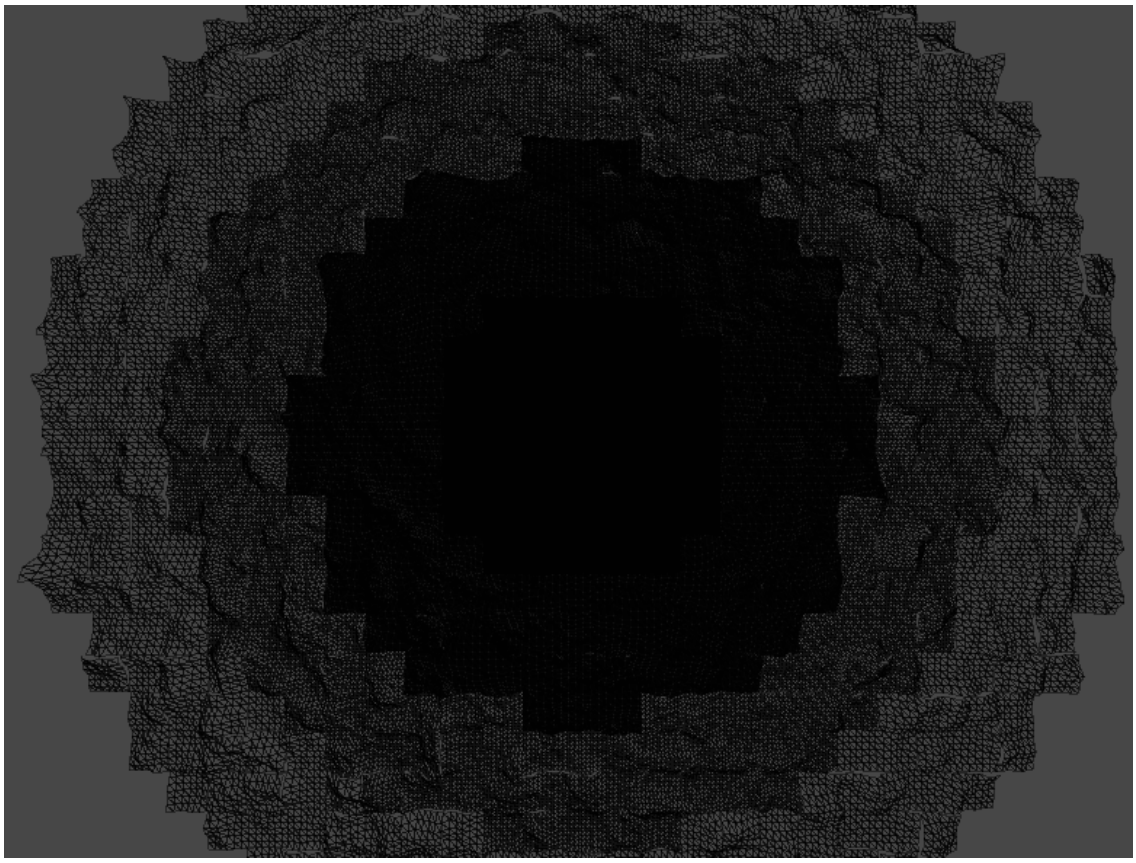


Figure 35: A top-down wireframe view of the terrain chunks with LOD switching

The final issue with this LOD switching implementation would be the seams that are introduced between each level of detail. I attempted finding a solution to this on my own, but in the end, I used a solution by Sebastian Lague. [28] This solution involved creating higher resolution border around lower resolution chunks. The idea is that if all chunks have these high resolution borders, then all chunks will be able to line up without any gaps. This solution is not perfect however, as on the low

detail chunks, the high detail border would create some jarring artefacts. However, as this was much too far away to notice, I deemed it better than having large gaps that you could see even from far away.

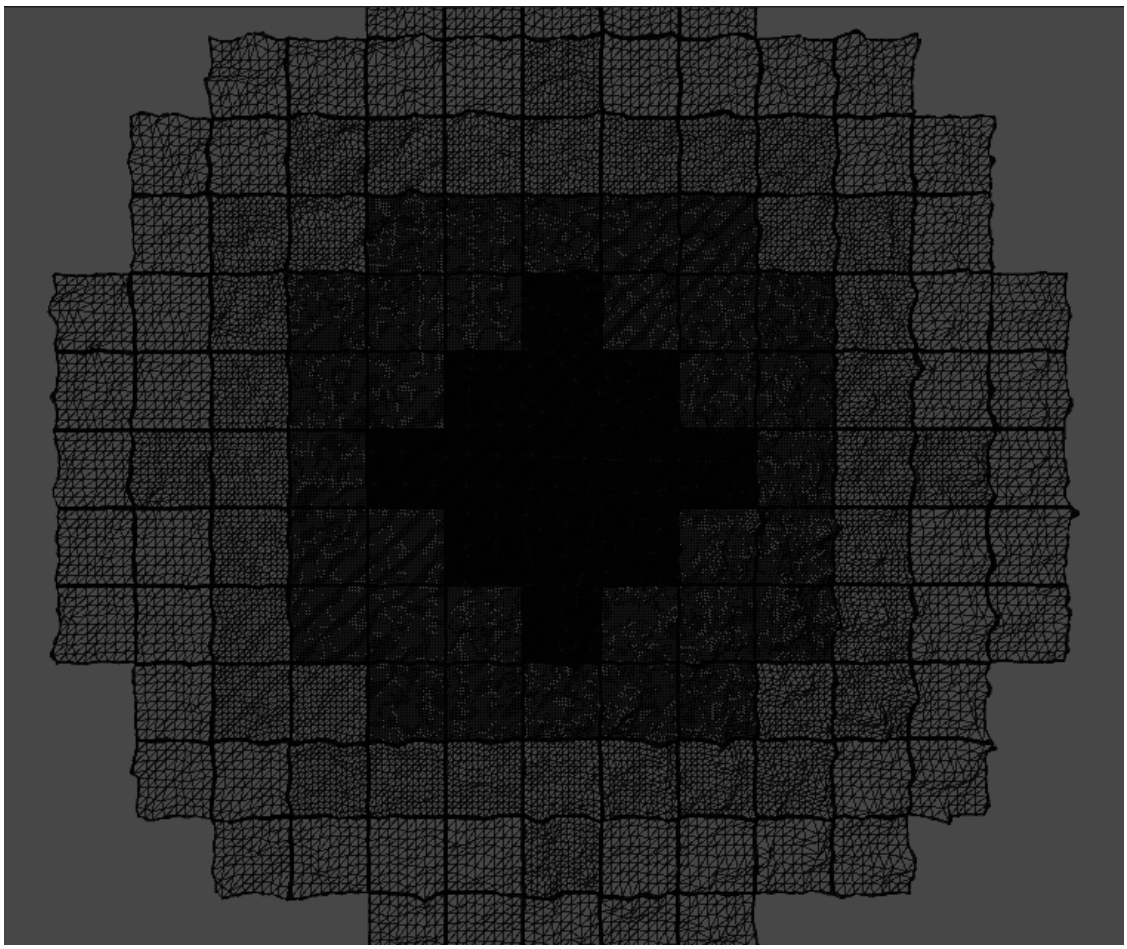


Figure 36: A top-down wireframe view of the terrain chunks with LOD switching and Sebastian Lague's LOD seam solution

The next optimization would be adding threading to the terrain generator. I created two threads; one to handle the generation of the height map, and one to handle the generation of the mesh data. I also created a queue for each thread. Each time a new mesh is to be generated, it will request a height map, and is added to the queue. The height map generator is locked such that only one height map is generated at a time in the order of the elements in the queue. A similar implementation exists for the mesh thread such that every time a new chunk is to be generated, it will request a mesh, and is added to the queue. The mesh generator is also locked in the same manner. An analysis of how this helped with performance of the game is in the testing, results & evaluation section of this dissertation.

The next optimization was to do with the generation of the collider. The first thing I did was reduce the range from the player at which colliders are generated. Previously, colliders would generate for any visible chunk. I implemented a second view distance for collider generation that is a quarter of the normal view distance. This means the script now only generates colliders for chunks that the player is near to.

The second optimization for the generation of colliders was to generate the collider at the level of detail below the true level of detail of that chunk. This would mean less vertices and triangles for the PhysX engine to bake while creating the collider. An analysis of how this helped with performance of the game is in the testing, results & evaluation section of this dissertation.

The final changes made were to do with how the chunks update themselves as the player moves. The first optimization in this regard was to only update the chunks after the player moves a certain amount, as opposed to every game tick. To implement this, I added a variable called the distance update threshold that is equal to a quarter of the chunk size. If a player moves as far as the distance update threshold, then the game would loop through all chunks and update them.

The second optimization was to check if a chunk needed updating before updating it. If a chunk is not visible this check cycle, and nor was it visible last check cycle, it will not be updated. This means chunks that are very far away are never updated, and only chunks in and around the players view distance are updated.

At this point, the game had all must-have and should-have features implemented, and I felt it was time to create some generator pre-sets for different biomes.

3.3.6 TOPOGRAPHIC VARIATION

I split the map settings into three categories:

- Mesh settings: Settings pertaining to the mesh such as the game object scale and chunk size.
- Height map settings: Settings pertaining to the noise generation and height map including seed, octaves, persistence, lacunarity, noise scale, height curve, height multiplier and water level.
- Texture data: Contains texture layers and the settings for each texture layer, including the starting height, texture, texture scale, tint colour, tint strength and blend strength.

Below are some images demonstrating different biomes with vast topographical variation, as well as the settings used to achieve them.

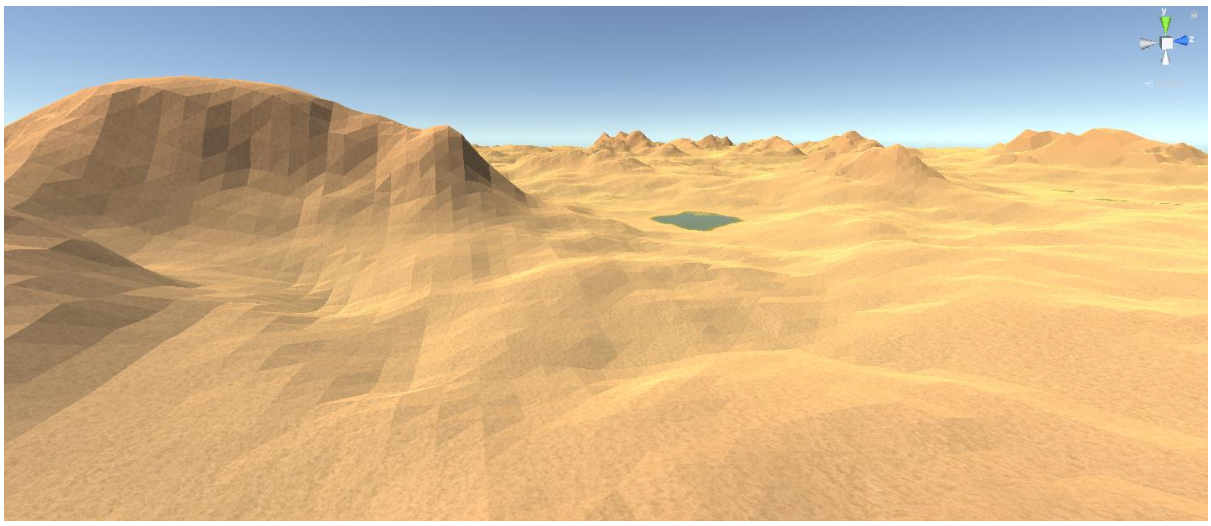


Figure 37: Desert oasis biome terrain

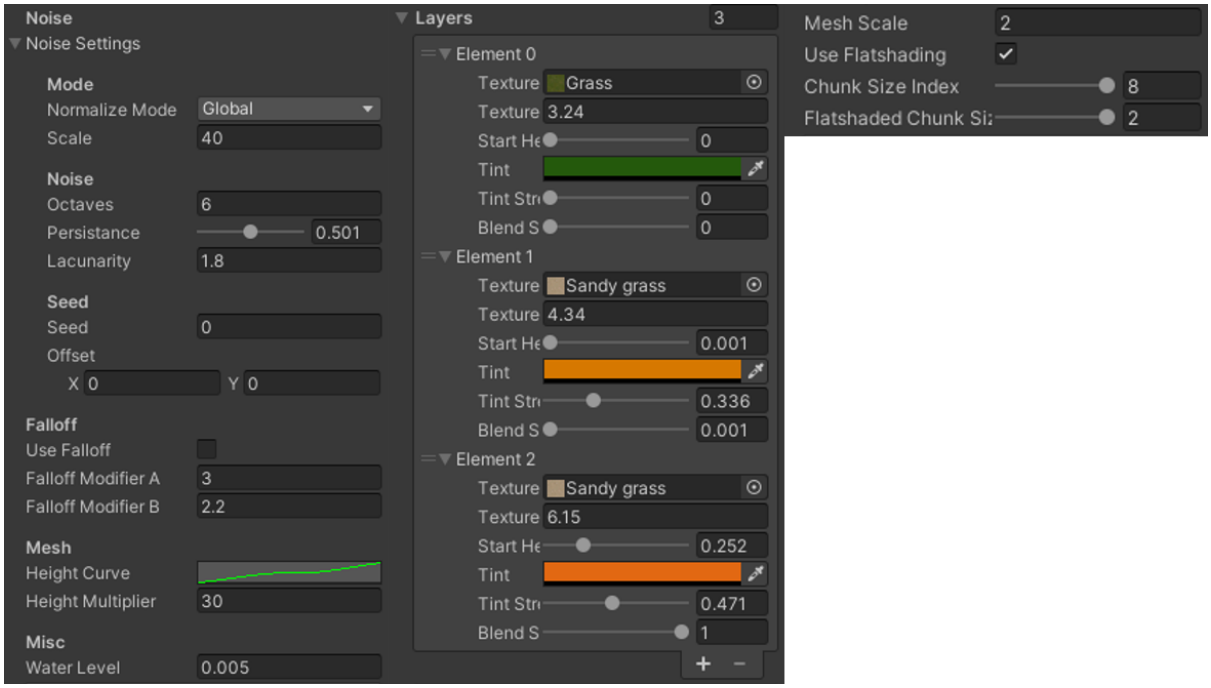


Figure 38: Desert oasis biome settings

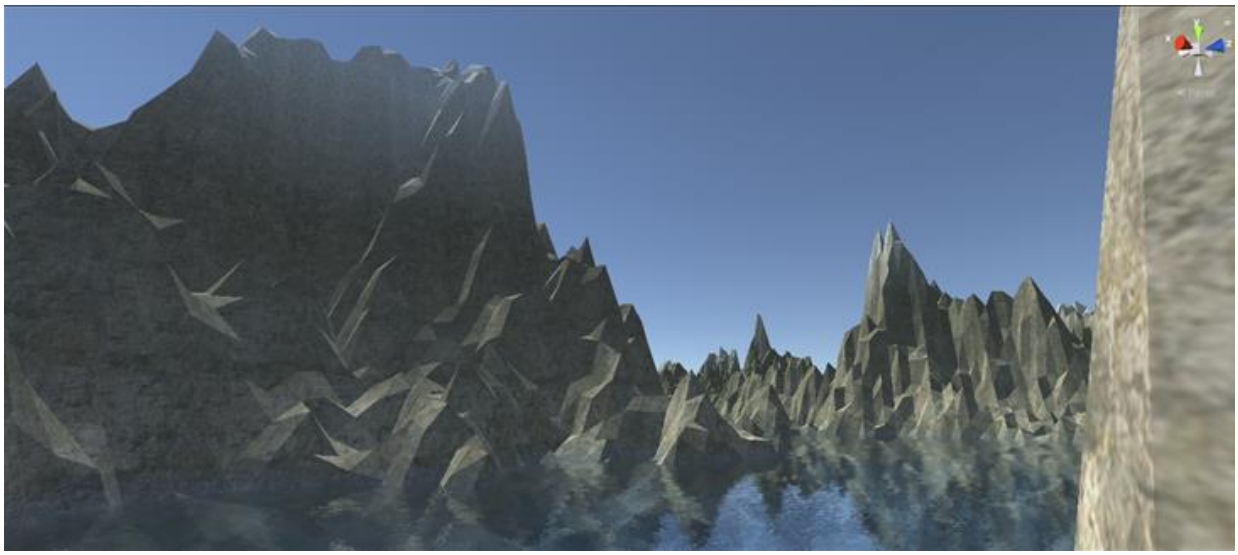


Figure 39: Jagged mountain rivers biome terrain

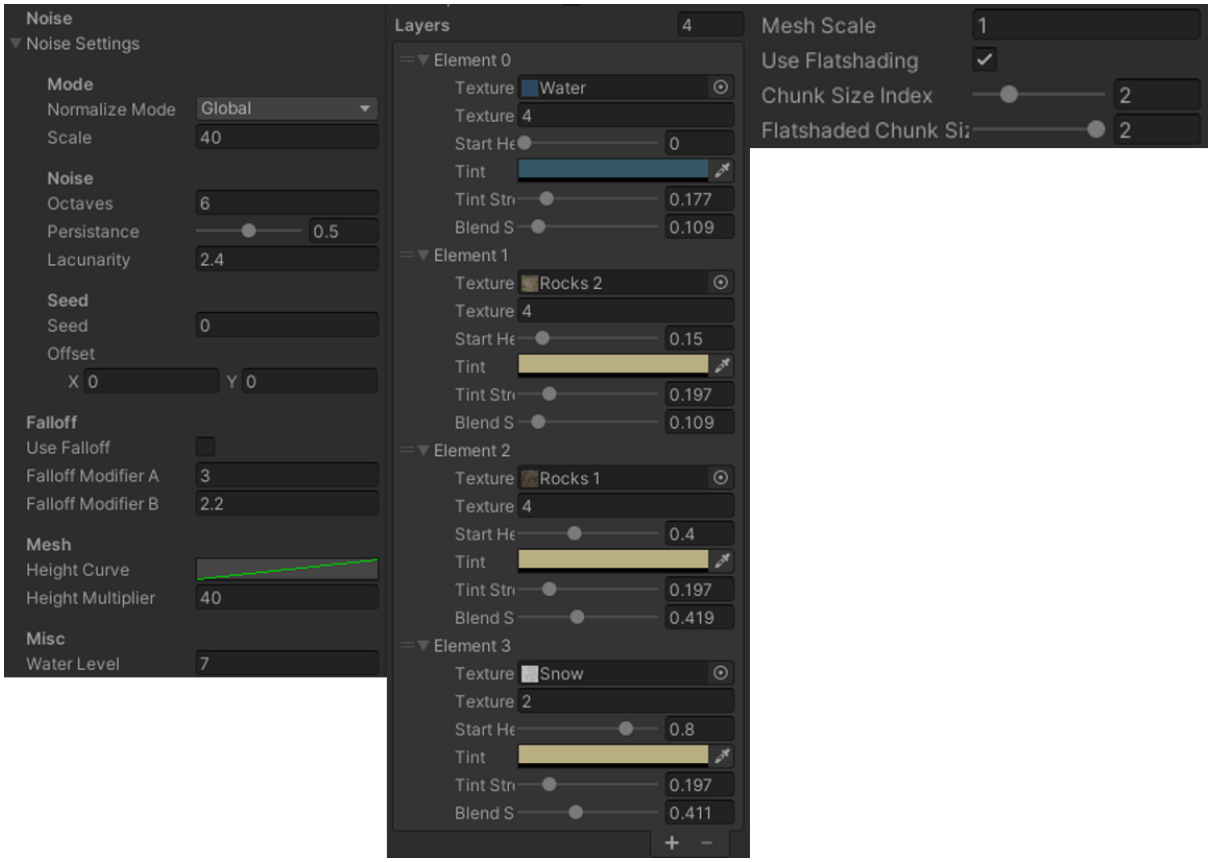


Figure 40: Jagged mountain rivers biome settings

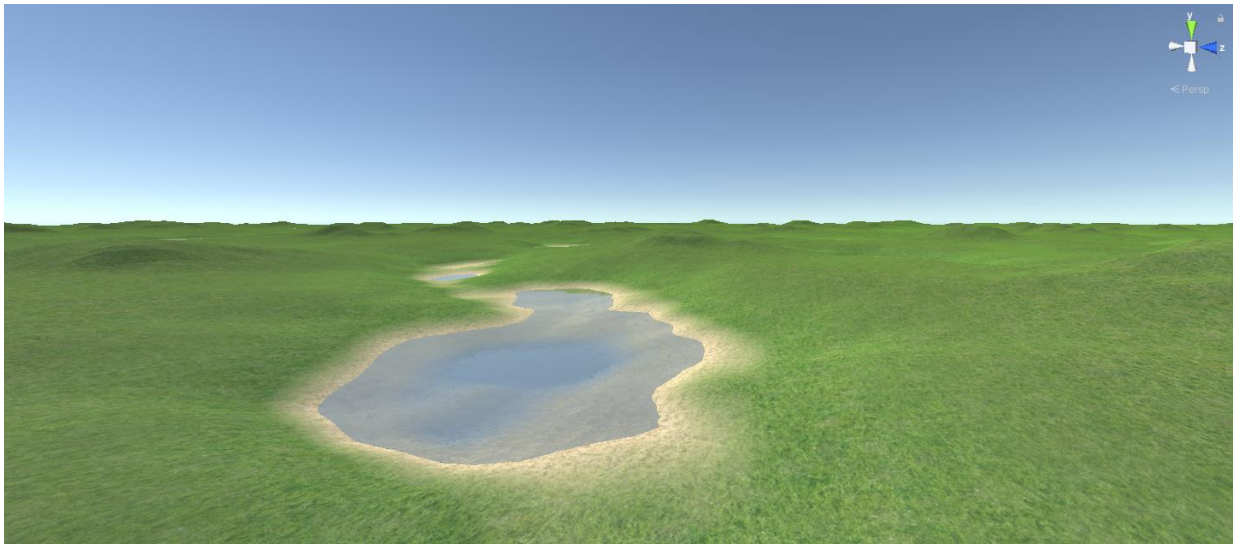


Figure 41: Plains lake biome terrain

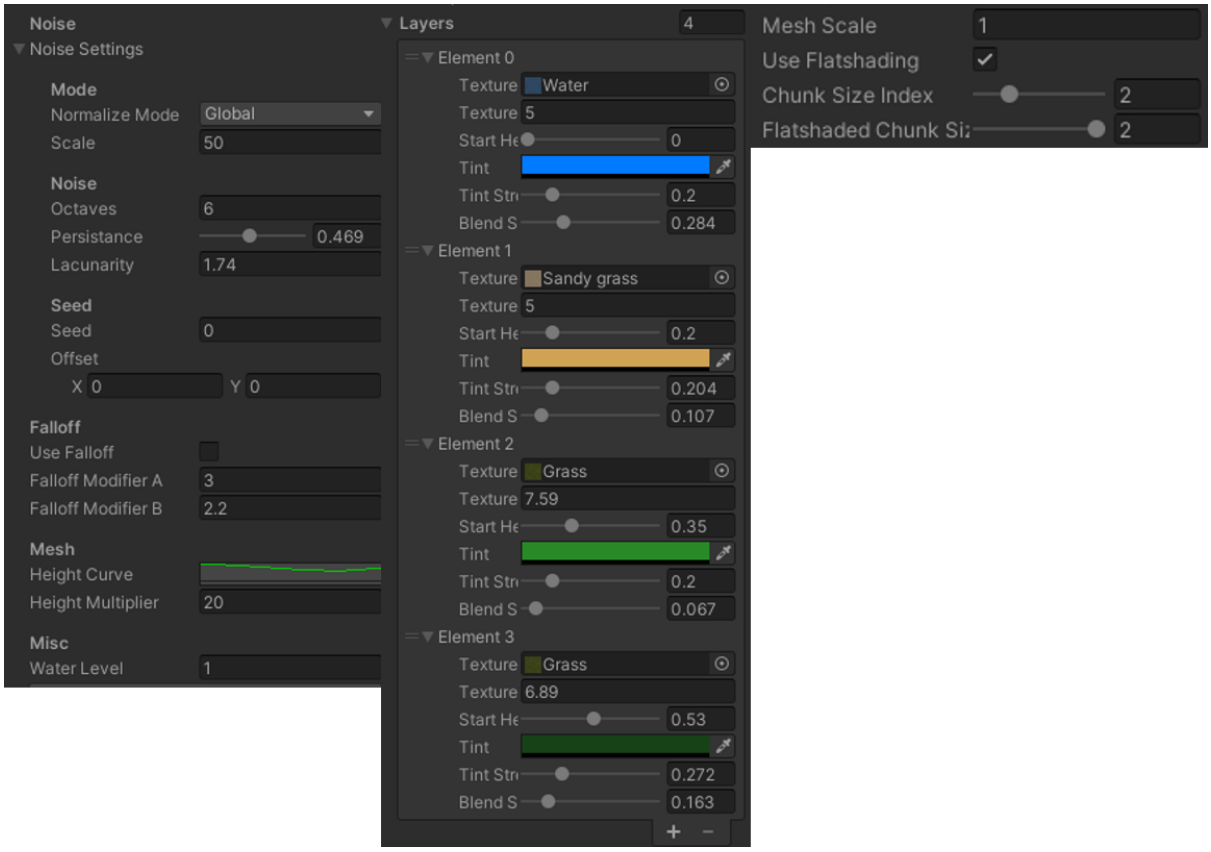


Figure 42: Plains lake biome settings

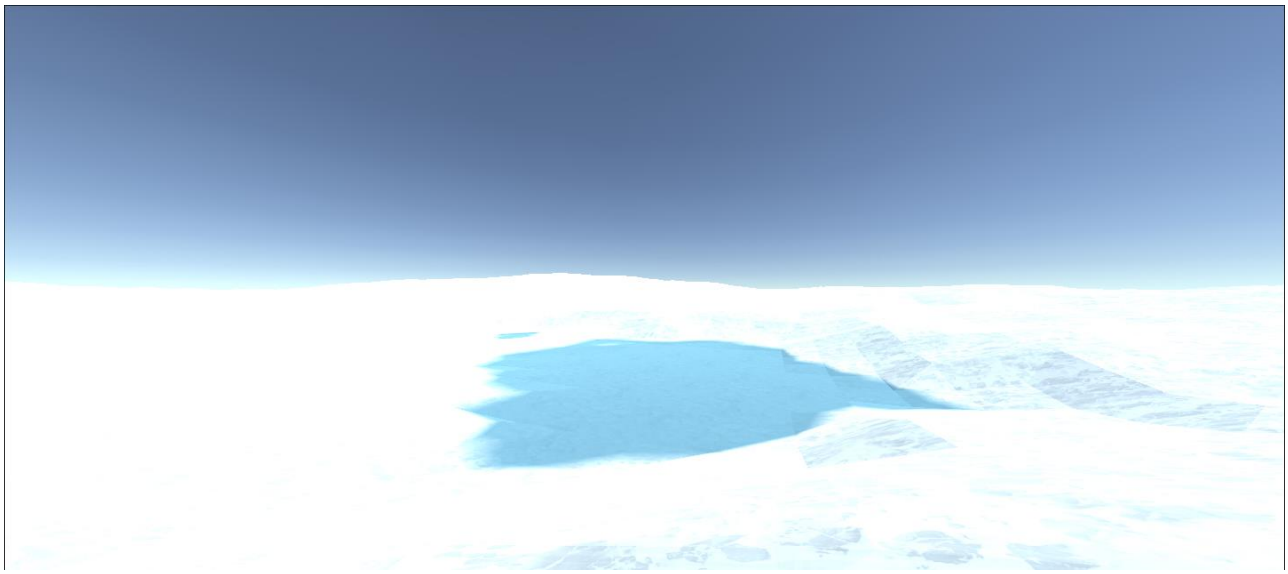


Figure 43: Tundra frozen lake biome terrain

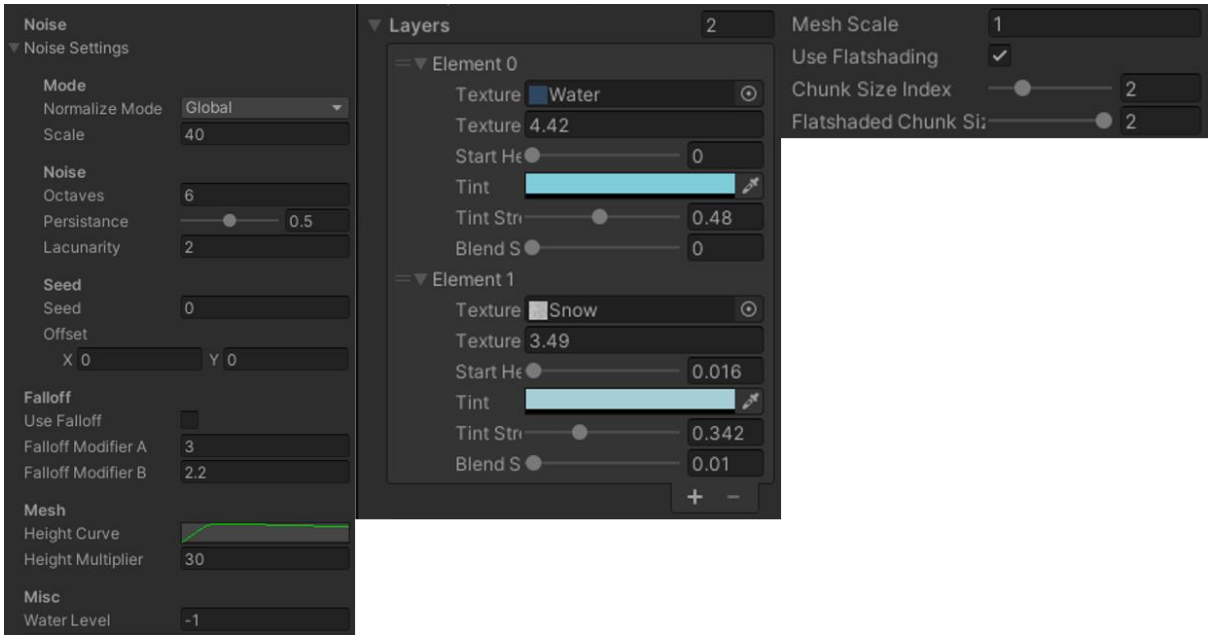


Figure 44: Tundra frozen lake biome settings

3.4 SUMMARY

The implementation of the project proceeded in a relatively smooth manner with only a few hiccups. When there was a problem, an effective solution was almost always found eventually.

All functional requirements, as well as all must-have and should-have items in the MoSCoW prioritization were met. Most non-functional requirements and could-have items in the MoSCoW prioritization were met.

This project will be evaluated against the aims and objectives set at the start of this project in the testing, results & evaluation section of the dissertation.

4 TESTING, RESULTS & EVALUATION

This section lays out the testing methodology and results of the project, followed by an evaluation against the criteria laid down in the aims and objectives.

4.1 EVALUATION TECHNIQUE

Testing will be conducted using Unity's built in deep profiling tool in combination with a custom Unity script I created to track average FPS, 1% low FPS and 0.1% low FPS. The code for this script can be found in the appendix.

In the project proposal, I had a specific set of hardware that I would be testing on that was around 7 years old. However, I have since upgraded my PC to one that can be considered a modern mid-range gaming PC.

Operating system	Windows 10
Form	Desktop PC
Processor	AMD Ryzen 5 5600X
RAM	Corsair Vengeance 16 GB DDR4 CAS-16 @ 3200MHz
SSD	Sabrent Rocket 1TB NVMe PCIe M.2 @ 3450 MB/s read & 3000MB/s write
Graphics card	NVIDIA GTX 1660 Super 6GB GDDR6 @ 1530 - 1785 (Boost) MHz

All of the tests below were conducted using this system with the aforementioned specifications. The tests were conducted as follows:

Average FPS, 1% FPS low and 0.1% FPS low measured across 5 tests involving moving in a straight line were tested against the following criteria:

- Collider creation at different LOD's
- Different chunk sizes
- With/without flat shaded lighting
- With/without LOD scaling
- Different view distances

All tests were conducted at a 16:9 aspect ratio on a 1080p monitor in Fullscreen mode.

4.2 TESTING & ANALYSIS

4.2.1 COLLIDER CREATION AT DIFFERENT LOD'S

Controls:

- View distance of 768
- LOD scaling at 256, 384 and 512 to LOD skip increments of 1, 2 and 3 respectively
- Chunk size of 96
- Flat shading on
- Jagged mountain rivers biome

LOD Increment	Average FPS	1% FPS Low	0.1% FPS Low
0	182.9	83.5	8.9
1	183.4	112.9	10.0
2	209.3	128.1	11.2
3	208.5	132.1	15.0
4	211.1	135.2	18.3

See appendix B.i for graphical representation

Analysis:

As the resolution of the collider decreases, we can see that all FPS readings improve. The most significant improvement observed was the 0.1% FPS lows. The Unity profiler reading suggests that this is due to the fact that the collider is taking less time to be created and applied to the object.

As all metrics point to the same conclusion, there is not much to analyse. Lower collider resolutions result in higher performance. However, we must consider the effect this had on the gameplay. Lower resolution colliders can cause inaccurate collision points on the mesh, harming the realism involved with walking across terrain. I believe collision LOD increment values of 0 and 1 are the only ones that are acceptable, as all other values result in the player seemingly floating just above, or slightly sunk into the ground, increasing in severity as the resolution decreases.

The optimal collision resolution appears to be the maximum resolution, as the difference between the two viable collision resolutions is minute in terms of performance but offers a more significant difference in terms of realistic gameplay.

4.2.2 DIFFERENT CHUNK SIZES

Controls:

- View distance of 768
- LOD scaling at 256, 384 and 512 to LOD skip increments of 1, 2 and 3 respectively
- Flat shading on
- Collider created at LOD skip increment 0
- Jagged mountain rivers biome

Chunk Size	Average FPS	1% FPS Low	0.1% FPS Low
48	217.1	111.2	8.9
72	218.6	112.9	10.0
96	216.5	120.1	11.2

See appendix B.ii for graphical representation

Analysis:

While the different chunk sizes had a negligible impact on the average FPS, it had a noticeable impact on the 1% FPS lows and a significant impact on the 0.1% FPS lows.

The Unity profiler suggests that this is due to chunk loading. Higher chunk sizes require more time for the processing of triangles, vertices and collider meshes, resulting in longer peak timings. This suggestion correlates with the data and explains why average FPS is not as adversely affected. The reason behind this is that for most frames during gameplay, the game is not generating or updating chunks, and so this does not leave a significant impact. However, when it does generate chunks, it takes longer, and therefore produces lower minimum FPS readings in the 1% and especially the 0.1% range.

According to the data, smaller chunk sizes have a higher average FPS, but larger chunk sizes have less FPS spikes. This makes sense as the mesh processing and updating procedures are spread more evenly over multiple frames but happen more often.

All three chunk sizes meet the criteria set in the objectives for minimum performance, and as such additional tests are needed to conclude which chunk size is optimal.

4.2.3 WITH/WITHOUT FLAT SHADED LIGHTING

Controls:

- View distance of 768
- LOD scaling at 256, 384 and 512 to LOD skip increments of 1, 2 and 3 respectively
- Collider created at LOD skip increment 0
- Chunk sizes of 48, 72 and 96
- Jagged mountain rivers biome

Flat Shading On

Chunk size	Average FPS	1% FPS Low	0.1% FPS Low
48	217.1	111.2	8.9
72	218.6	112.9	10.0
96	216.5	120.1	11.2

See appendix B.iii for graphical representation

Flat Shading Off

Chunk size	Average FPS	1% FPS Low	0.1% FPS Low
48	278.9	151.6	12.4
72	283.0	152.7	13.8
96	291.0	154.1	13.8

See appendix B.iv for graphical representation

Analysis:

Flat shading impacts the performance of the game across all FPS ranges. We can also gather that chunk sizes have a lesser impact on the games performance when flat shading is off.

The unity profiler suggests that this is due to the increased number of vertices per mesh that need to be processed when creating the mesh, as well as the increased number of normals to check when calculating the lighting.

Flat shading is an integral part of the aesthetic of the game, and I believe that this outweighs the performance impact it has on the game. This is because all performance criteria set in the objectives are met, even at the worst case scenario with flat shading on. Therefore, it would not be a worthwhile trade-off to remove flat shading for the purpose of performance gains.

4.2.4 WITH/WITHOUT LOD SCALING AND DIFFERENT VIEW DISTANCES

Controls:

- View distances of 384, 512 and 768
- Collider created at LOD skip increment 0
- LOD scaling, if enabled, set to 25%, 50% and 75%
- Chunk size of 96
- Flat shading off
- Jagged mountain rivers biome

LOD Scaling On

View Distance	Average FPS	1% FPS Low	0.1% FPS Low
384	320.3	139.8	16.8
512	299.7	126.4	13.3
768	237.5	121.9	12.0

See appendix B.v for graphical representation

LOD Scaling Off

View Distance	Average FPS	1% FPS Low	0.1% FPS Low
384	238.6	133.3	10.2
512	120.7	89.4	8.7
768	106.3	50.3	6.5

See appendix B.vi for graphical representation

Analysis:

Turning LOD scaling off negatively impacted all FPS reading significantly and has the most significant performance impact of any of the other performance enhancements implemented.

The performance impact is more dramatic as the view distance increases. At a view distance of 768 with LOD scaling off, the performance is less than half of the performance at the same view distance with LOD scaling on, whereas on a view distance of 384, the performance impact is roughly 50%.

The Unity profiler suggests that this is due to the significantly larger number of vertices and normals to be processed by the engine per frame.

While LOD scaling can cause unusual visual artefacts if the player purposefully looks for them in the distance, this is a very small sacrifice in terms of gameplay when compared to the performance impact that is observed. Therefore, LOD scaling should be on at all times.

4.3 EVALUATION AGAINST OBJECTIVES

4.3.1 TO CREATE A CHUNK OF TERRAIN USING A TRIANGLE MESH

Each chunk has a mesh, and each mesh is created at a set size using a triangle mesh. The mesh filter, mesh collider and mesh renderer all function adequately and therefore this objective was met.

4.3.2 TO USE PERLIN NOISE TO CREATE VARIATION IN TOPOGRAPHY

The noise generator uses Perlin noise to create the height map, which is used to create a mesh. The transitions between chunks are smooth due to an offset provided to the noise generator. Variation in the topography is possible as demonstrated in the topographic variation sub-section of the implementation section. This is made possible by providing the following possible adjustable variables:

- Mesh settings: Settings pertaining to the mesh such as the game object scale and chunk size.

- Height map settings: Settings pertaining to the noise generation and height map including seed, octaves, persistence, lacunarity, noise scale, height curve, height multiplier and water level.
- Texture data: Contains texture layers and the settings for each texture layer, including the starting height, texture, texture scale, tint colour, tint strength and blend strength.

As such, diverse variation in topography is possible and easy to implement. Therefore, this objective was met.

4.3.3 TO IMPLEMENT INFINITE PSEUDO-RANDOMLY GENERATED CHUNKS OF TERRAIN

During the tests for the different performance criteria in the testing sub-section of the testing, results & evaluation section, the player moves for 5 minutes in a random direction in a straight line. During this, the terrain generator is sampling Perlin noise using different offsets for each chunk depending on the current location of the player and the map coordinate of the chunk. No repeated chunks are possible with this implementation. Furthermore, changing the seed generates a completely different set of terrain, sampling from a much farther point in the Perlin noise function. Therefore, this objective has been met.

4.3.4 TO ENSURE AN ACCEPTABLE LEVEL OF PERFORMANCE

The SMART sub-objectives set were:

a. An average FPS of 60.
b. A 1% low FPS of no less than 45.
c. A 0.1% low FPS of no less than 5.
d. FPS results are averaged over 5 tests, each involving 5 minutes of constantly moving in one direction and generating chunks.

As can be seen in the analysis section, all sub-objectives were exceeded in all but extreme test cases. The settings used for optimal visual detail all meet these criteria, and as such this objective has been met.

4.3.5 TO IMPROVE VISUAL DETAIL

The main purpose of this objective was to provide variation between different biome types with textures and water. Different biomes have:

- Different terrain
- Different textures and tints
- Unique sky colours

The second criteria for this objective were to provide a cartoon-like graphical experience by implementing flat shading. As can be seen in the screenshots throughout the topographic variation sub-section of the development section, the game incorporates flat shading into its graphical representation. Therefore, this objective has been met.

5 CONCLUSIONS

This section contains my final notes on personal development and the continuation of the project.

5.1 PERSONAL DEVELOPMENT

Throughout this project, I have learned new skills and developed old ones:

- Research skills. I have learned how to efficiently research information pertaining to a project. Google Scholar is an immensely powerful tool and can provide insight in detail that YouTube and Wikipedia cannot, while making citing sources easy.
- Project management. Managing this project has taught me a good deal about time management, resource allocation as well as the importance of meeting set deadlines. This project was the first in which I truly gave it my all, and I feel I have accomplished something great by doing it.
- Unity development. Unity is a powerful tool for game development. I found it intuitive and simple to use once I understood its inner workings. The massive adaptation of this engine makes it easy to find information online for, especially for smaller indie developers such as myself.
- Problem solving. My process for problem solving has changed dramatically for the better. The mentality employed and required for game development is much harsher than I previously realised, as some problems can leave you stuck and frustrated for weeks. However, with persistence, almost all problems can be solved, and sometimes with unique solutions.

I have gained a lot from this project and thoroughly enjoyed doing it. This project has made game development one of my main hobbies, and hopefully this enthusiasm and interest will translate well into the professional game development industry.

5.2 CONTINUING THE PROJECT

The version of the game that you can see in this dissertation is version 0.5.11. I plan to continue with this project and develop it into a survival RPG with bosses, dungeons and other features. In order to do this, I still need to work on the terrain generation in a few ways, as well as some core survival RPG features.

5.2.1 EROSION

Erosion algorithms can add increased realism to procedural noise generation. Some examples of erosion algorithms include:

Thermal erosion

"Thermal erosion models gravity eroding cliffs that are too steep. If the angle is too sharp, soil will fall to a lower area. Thermal erosion is fairly simple to model. First, define the difference T, which is the maximum difference allowed before gravity takes over. For all pixels:

1. *Get the difference in height between this pixel and the neighbouring pixel*

2. *If the difference is greater than T, remove some amount of soil from the taller pixel and deposit it into the lower pixel." [29]*

Hydraulic erosion

"Hydraulic Erosion models rainwater picking up soil, washing down into basins, then evaporating and depositing soil.[5] Hydraulic erosion provides good quality results, but is very slow. Hydraulic Erosion also requires large amounts of memory. While the other erosion algorithms work directly with the source image, Hydraulic Erosion requires a water table and a sediment table, meaning it may use up to 3 times the memory of Thermal Erosion." [29]

I believe adopting these erosion techniques would increase the realism of my terrain, and it is what I will be working on next.

5.2.2 UPDATE TO URP

The Unity Universal Render Pipeline is a scriptable render pipeline that is more versatile than the Built-In Render Pipeline. Some benefits and features of the URP I wish to take advantage of including are:

- Performance. The URP, despite being graphically more stunning, is much faster than the Built-In Render Pipeline.
- Simplicity. The URP is simpler to use, and likely won't require me to spend as much time learning about.
- Active development. The URP is still in active development, meaning there will be newer features coming out over time.
- Shader graphs. The URP's main selling point is its versatile shader graphs. This feature allows you to build shaders visually. Instead of writing code, I could create and connect nodes in a graph framework, providing instant feedback that reflects my changes. This would solve the current texture problem I am having when attempting to load multiple terrain types in the same world.

5.2.3 OTHER FEATURES

- Same-world biomes. While the purpose of this dissertation was to generate different topographic variations in landscape on a procedurally generated alien planet, a complete game would have biome transitions.
- Non-terrain environmental features. This would include trees, rocks and other foliage and scenery elements that would improve the realism of the game.
- More advanced water system. The current water system implemented does not support water being at different levels, such as rivers flowing downhill or waterfalls.
- NPCs. Animals, monsters, and bosses are a must-have in survival RPG games, where they can be found in the overworld or possibly in dungeons underground.
- Inventory, crafting and loot system. This would provide some gameplay incentive to the player as they would collect new materials throughout their journey.

TABLE OF REFERENCES

[1] ewg.k12.ri.us. (2014). How Minecraft Started. [online] Available at: https://sites.google.com/a/ewg.k12.ri.us/arc-minecraft/home/how-minecraft-started [Accessed 1 January 2021].

[2] independent.co.uk (2014). Built to last: the Minecraft model. [online] Available at: https://www.independent.co.uk/news/business/analysis-and-features/built-last-minecraft-model-9788669.html [Accessed 1 January 2021].

[3] denofgeek.com (2014). Why indie videogaming is so important. [online] Available at: https://www.denofgeek.com/games/why-indie-videogaming-is-so-important [Accessed 1 January 2021].

[4] Craddock, D., 2015. *Dungeon hacks*. Press Start Press.

[5] Youtube.com. 2021. kkrieger: Making an Impossible FPS. [online] Available at: <https://www.youtube.com/watch?v=bD1wWY1YD-M> [Accessed 4 Jan 2021].

[6] Massivesoftware.com. 2021. Massive Software :: About Us. [online] Available at: <http://www.massivesoftware.com/about.html> [Accessed 4 Jan 2021].

[7] Smith, G., 2014, April. Understanding procedural content generation: a design-centric analysis of the role of PCG in games. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 917-926).

[8] Lovellette, E. and Hexmoor, H., 2016, October. Abstract Argumentations Using Voronoi Diagrams. In 2016 International Conference on Collaboration Technologies and Systems (CTS) (pp. 472-477). IEEE.

[9] Fingas, J., 2021. Here's how 'Minecraft' creates its gigantic worlds. [online] Engadget.com. Available at: <https://www.engadget.com/2015-03-04-how-minecraft-worlds-are-made.html> [Accessed 16 Jan 2021].

[10] Persson, M., 2011. Terrain generation, Part 1. [Blog] The Word of Notch, Available at: <https://n0tch-blog-blog.tumblr.com/post/4231184692/terrain-generation-part-1> [Accessed 16 Jan 2021].

[11] Van Der Linden, R., Lopes, R. and Bidarra, R., 2013. Procedural generation of dungeons. IEEE Transactions on Computational Intelligence and AI in Games, 6(1), pp.78-89.

[12] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley, October 1994. Texturing and Modelling: A Procedural Approach. Academic Press. ISBN 0-12-228760-6

[13] Shaker, N., Togelius, J. and Nelson, M.J., 2016. Fractals, noise and agents with applications to landscapes. In Procedural Content Generation in Games (pp. 57-72). Springer, Cham.

[14] Doran, J. and Parberry, I., 2010. Controlled procedural terrain generation using software agents. IEEE Transactions on Computational Intelligence and AI in Games, 2(2), pp.111-119.

[15] Parberry, I., 2014. Designer worlds: Procedural generation of infinite terrain from real-world elevation data. Journal of Computer Graphics Techniques, 3(1).

[16] Scher, Y., 2021. Playing with Perlin Noise: Generating Realistic Archipelagos. [online] Medium. Available at: <https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401> [Accessed 18 Jan 2021].

[17] Curry, P., 2021. Postmortem: Dinosaur Polo Club's Mini Metro. [online] Gamasutra.com. Available at: <https://www.gamasutra.com/view/news/273284/Postmortem_Dinosaur_Polo_Clubs_Mini_Metro.php> [Accessed 19 Jan May 2021].

[18] Gould, R., 2021. The Programmed Music of “Mini Metro” – Interview with Rich Vreeland (Disasterpeace). [online] Designingsound.org. Available at: <https://designingsound.org/2016/02/18/the-programmed-music-of-mini-metro-interview-with-rich-vreeland-disasterpeace/> [Accessed 19 Jan 2021].

[19] Lague, S., 2021. Procedural Landmass Generation (E11: falloff map). [online] Youtube.com. Available at: <https://www.youtube.com/watch?v=COmtTyLCd6I> [Accessed 22 Jan 2021].

[20] Hextant Studios. 2021. Rendering Flat-Shaded / Low-Poly Style Models in Unity. [online] Available at: <https://hextantstudios.com/unity-flat-low-poly-shader/> [Accessed 22 Jan May 2021].

[21] Lague, S., 2021. Procedural Landmass Generation (E14: flatshading). [online] Youtube.com. Available at: <https://www.youtube.com/watch?v=V1vL9yRA_eM> [Accessed 24 Jan 2021].

[22] Docs.unity3d.com. 2021. Unity - Scripting API: Mesh. [online] Available at: <https://docs.unity3d.com/ScriptReference/Mesh.html> [Accessed 24 January 2021].

[23] Docs.unity3d.com. 2021. Unity - Scripting API: MeshRenderer. [online] Available at: <https://docs.unity3d.com/ScriptReference/MeshRenderer.html> [Accessed 24 January 2021].

[24] Docs.unity3d.com. 2021. Unity - Scripting API: MeshFilter. [online] Available at: <https://docs.unity3d.com/ScriptReference/MeshFilter.html> [Accessed 24 January 2021].

[25] Youtube.com. 2021. MESH GENERATION in Unity - Basics. [online] Available at: <https://www.youtube.com/watch?v=eJEpeUH1EMg> [Accessed 24 January 2021].

[26] Youtube.com. 2021. Writing Shaders In Unity - Basic Shader - Beginner Tutorial. [online] Available at: <https://www.youtube.com/watch?v=bR8DHcj6Htg> [Accessed 12 February 2021].

[27] Lague, S., 2021. [online] Youtube.com. Available at: <https://www.youtube.com/watch?v=XdahmaohYvI> [Accessed 25 February 2021].

[28] Lague, S., 2021. Procedural Landmass Generation (E21: fixing gaps). [online] Youtube.com. Available at: <https://www.youtube.com/watch?v=c2BUgXdjZkg> [Accessed 25 February 2021].

[29] Archer, T., 2011. Procedurally generating terrain. In 44th annual midwest instruction and computing symposium, Duluth (pp. 378-393).

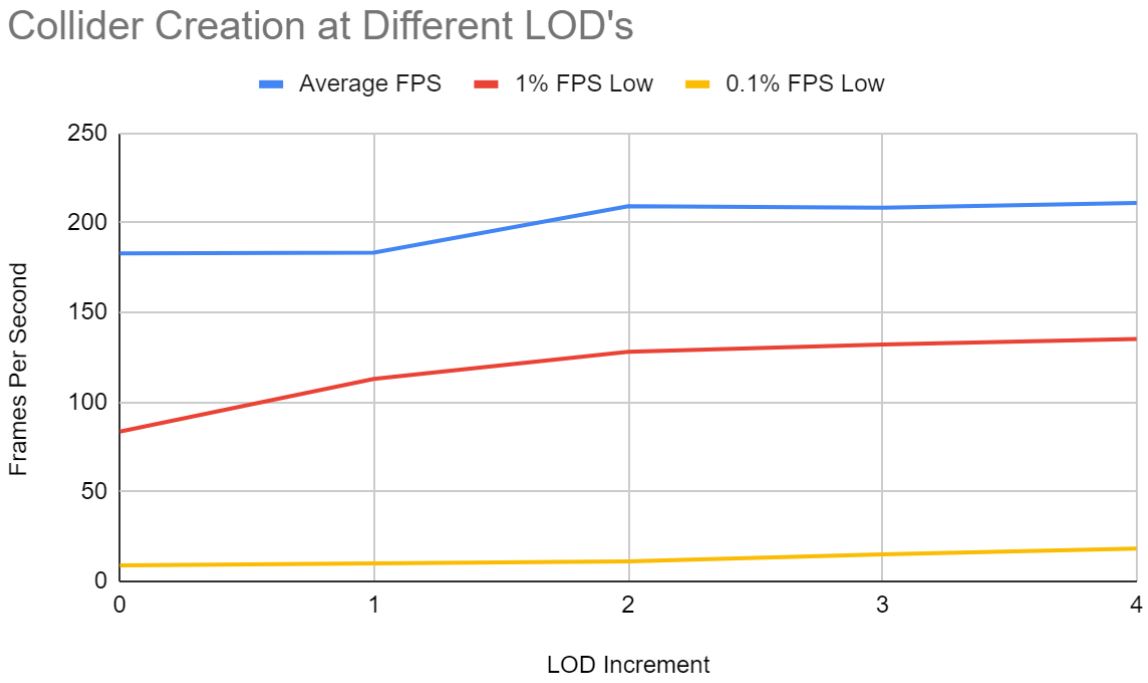
APPENDIX

A) CUSTOM SCRIPT FOR MEASURING FPS

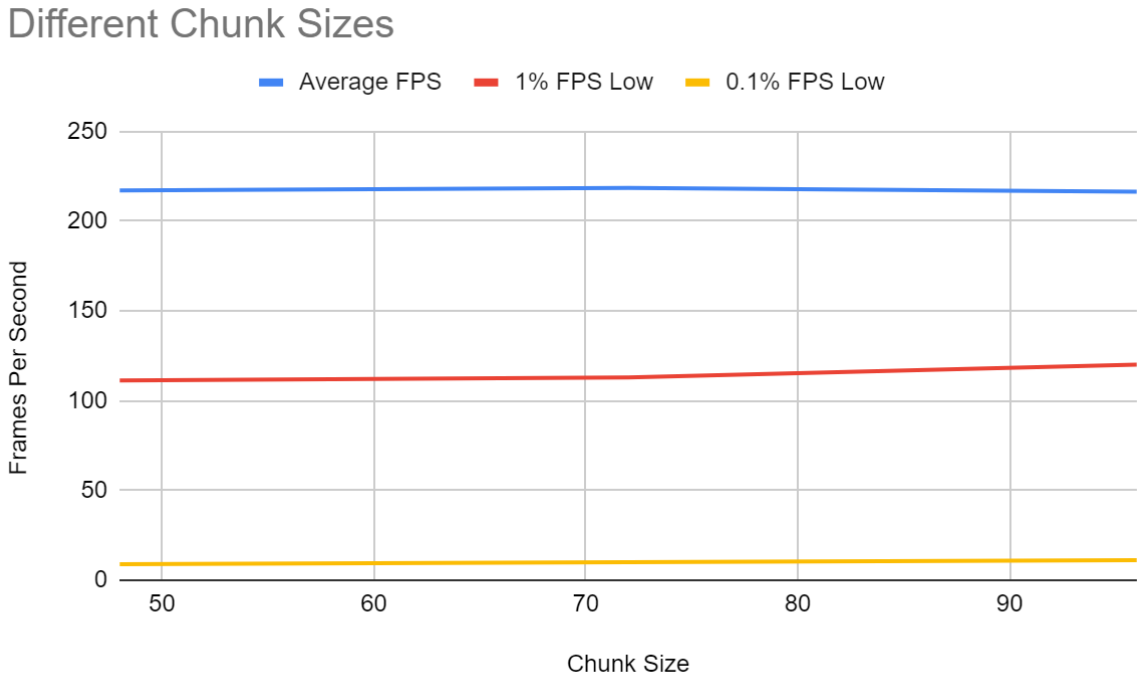
```
Unity Script | 0 references
public class FPSScript : MonoBehaviour
{
    float deltaTime = 0.0f;
    float onePercentLoopCounter;
    float pointOnePercentLoopCounter;

    ArrayList fpsStore = new ArrayList();
    float averageFPS = 0.0f;
    float onePercentLow = 0.0f;
    float pointOnePercentLow = 0.0f;


    Unity Message | 0 references
    void Update()
    {
        deltaTime += (Time.unscaledDeltaTime - deltaTime) * 0.1f;
    }

    Unity Message | 0 references
    void OnGUI()
    {
        int w = Screen.width, h = Screen.height;

        GUIStyle style = new GUIStyle();

        Rect rect = new Rect(0, 0, w, h * 2 / 100);
        style.alignment = TextAnchor.UpperLeft;
        style.fontSize = h * 2 / 100;
        style.normal.textColor = new Color(0.0f, 0.0f, 0.5f, 1.0f);
        float msec = deltaTime * 1000.0f;
        float fps = 1.0f / deltaTime;
        fpsStore.Add(fps);

        if (onePercentLoopCounter >= 100)
        {
            onePercentLow = 0;
            fpsStore.Sort();

            for (int i = 0; i < fpsStore.Count * 0.01f; i++)
            {
                onePercentLow += (float) fpsStore[i];
            }
            onePercentLow /= fpsStore.Count * 0.01f;
            onePercentLoopCounter = 0;
        }

        if(pointOnePercentLoopCounter >= 1000)
        {
            pointOnePercentLow = 0;
            fpsStore.Sort();

            for (int i = 0; i < fpsStore.Count * 0.001f; i++)
            {
                pointOnePercentLow += (float) fpsStore[i];
            }
            pointOnePercentLow /= fpsStore.Count * 0.001f;
            pointOnePercentLoopCounter = 0;

            for (int i = 0; i < fpsStore.Count; i++)
            {
                averageFPS += (float)fpsStore[i];
            }
            averageFPS /= fpsStore.Count;
        }


        string text = string.Format("{0:0.0} ms ({1:0.} fps)\n{4:0.0} (Average fps)\n{2:0.0} (1% low fps)\n{3:0.0} (0.1% low fps)", msec, fps, onePercentLow, pointOnePercentLow, averageFPS);
        GUI.Label(rect, text, style);
        onePercentLoopCounter++;
        pointOnePercentLoopCounter++;
    }
}
```

B) GRAPHS FROM TESTING

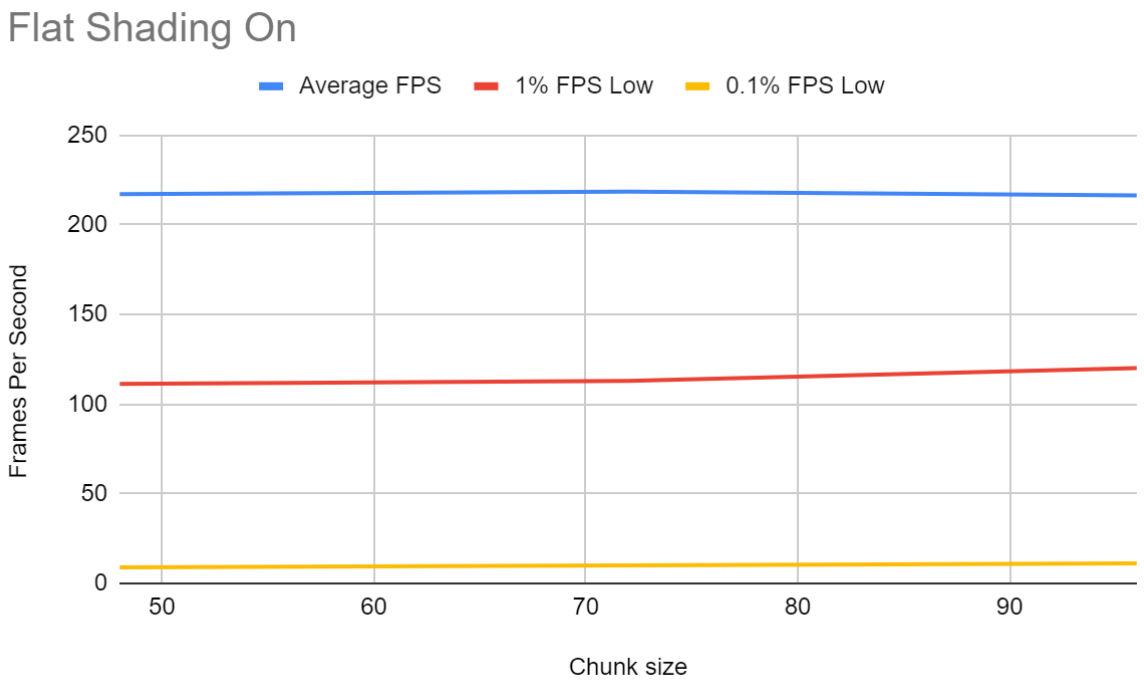
I) COLLIDER CREATION AT DIFFERENT LOD'S



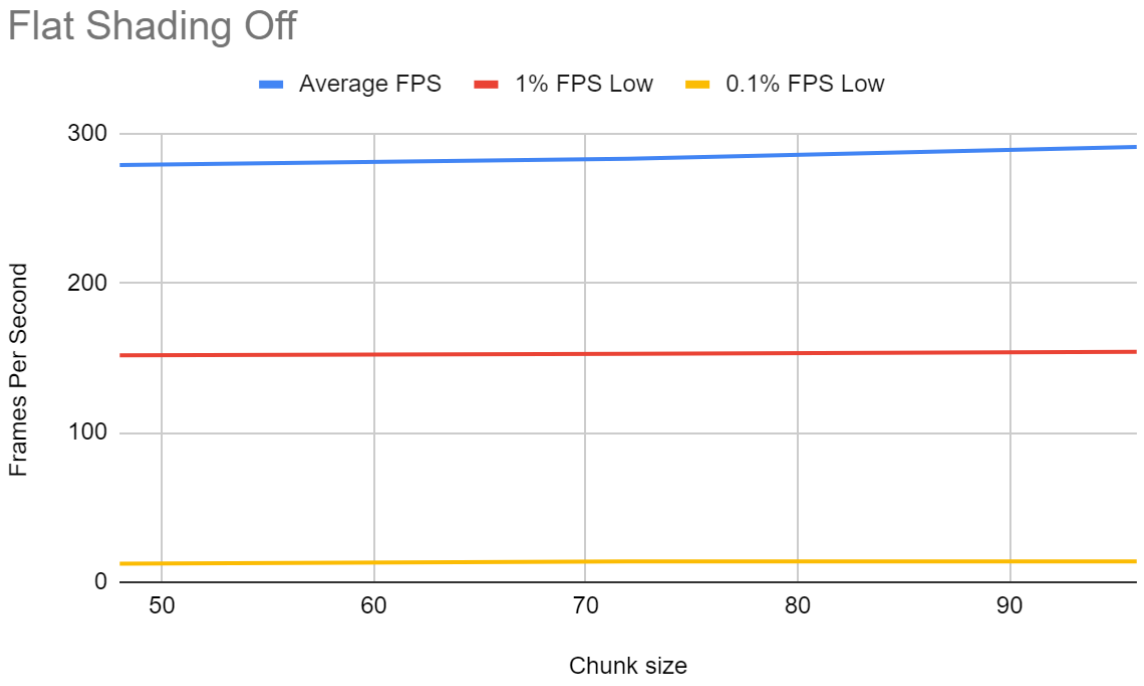
II) DIFFERENT CHUNK SIZES



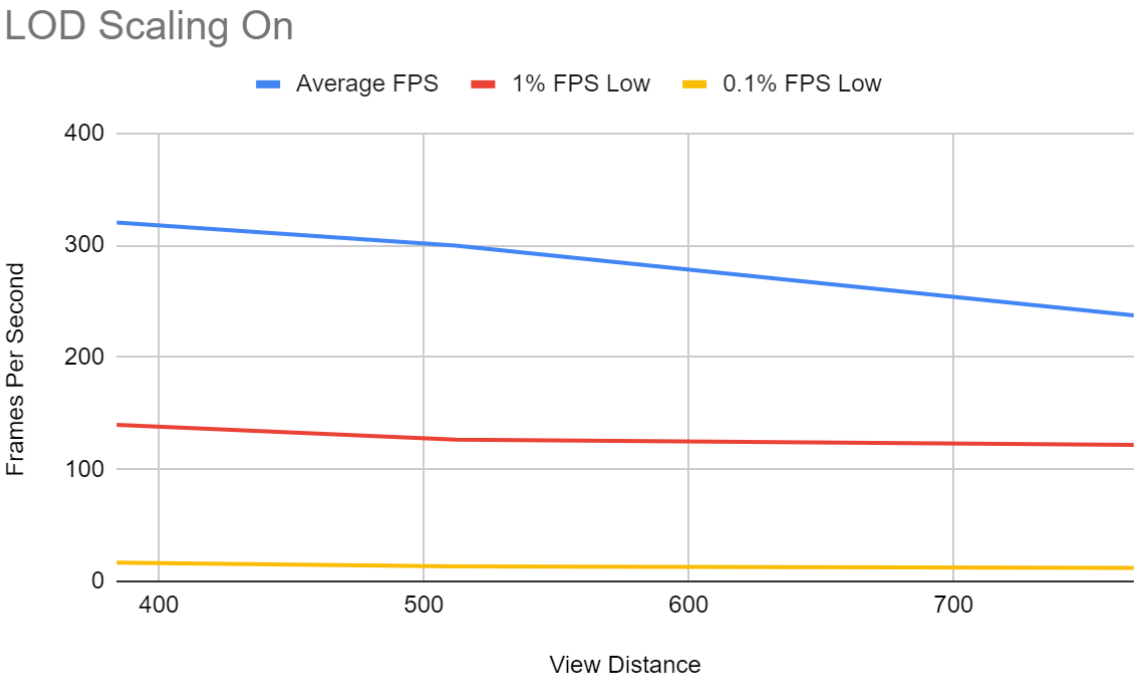
III) FLAT SHADING ON



IV) FLAT SHADING OFF



V) LOD SCALING ON



VI) LOD SCALING OFF