School of Computer Science

Procedural (random) map generation in computer games

Angel Barquin Pereira
18101382

August 21st, 2021

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
BSc (Computer Science and Information Technology)

Contents

List of Figures

1 Introduction

1.1 Project Aim

This project aims to understand, research, and develop a procedural map generation tool. This tool will mainly serve for the creation of game maps. Procedural generation is present in multiple games as is a very effective method to:

- Cut costs at the time of making a game and save developer time by doing some of the game design for us.
- Save on space by not needing to store assets on memory by generating them in real-time and storing them in RAM.

Examples of procedurally generated assets could be maps, player models, textures, and more. Upon researching the types of content generation, I came across the difference between the terms procedural and random, and they appear to be almost opposing terms. The focus of my project will be based on trying to achieve the least randomness and most procedural goal, meaning we are supposed to get the same results if we keep using the same seed value that acts as our input.

1.2 Concepts

1.2.1 Seed

Seed is an input or starting point. They are used constantly for random number generators in either pseudo-randomness (where it is not too random) or pure random seeking generation. Seeds are usually taken from the milliseconds of the current time. In procedural generation, they are the starting point of your content, the very first thing that goes into an algorithm. Deterministic generation: said to be the procedural content generation that produces the same content given the same input. Examples of

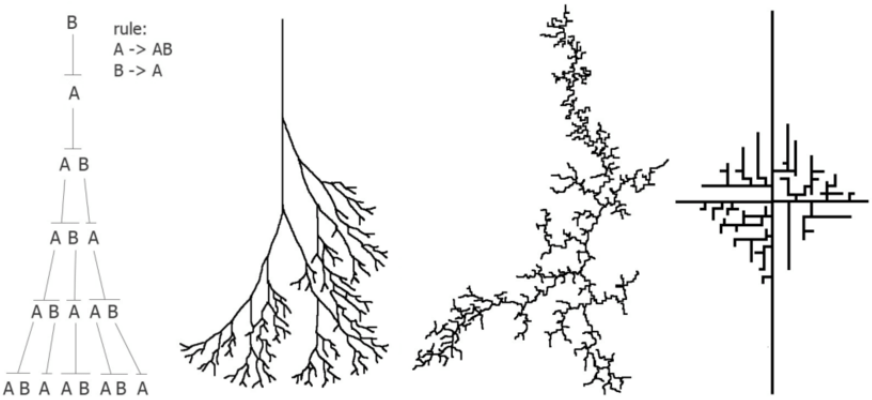
this would be Conway's game of life (1), where certain cell initial positions will always end up a certain way at the end. It is the opposite of Stochastic generation, where a random factor in the algorithm makes the result different each time.

1.2.2 Stochastic generation

Stochastic generation is the opposite of deterministic generation. Stochastic generation looks more into producing different results with every iteration. If we always give it the same initial input, it will always end up a bit different. The result will never be able to be replicated exactly. Stochastic algorithms feature more randomness in between and could be affected by external factors like game weather at the time. Popular types of Stochastic generation are evolutionary algorithms that contain the natural selection theories by Darwin, mutation, and cross-breeding.

1.2.3 L-Systems

A type of procedural generation based on an initial state and grammatic rules. For example, the most basic L-system invented by Lindenmayer (creator)(2) is the Algae system. It starts with the letter B, and the rules say: A gives birth to AB, B gives birth to another A. If we count the length of each iteration, we get the Fibonacci Sequence of numbers: 1, 2, 3, 5, 8…(3)



L-systems are mostly used for plant generation.

1.2.4 Perlin Noise

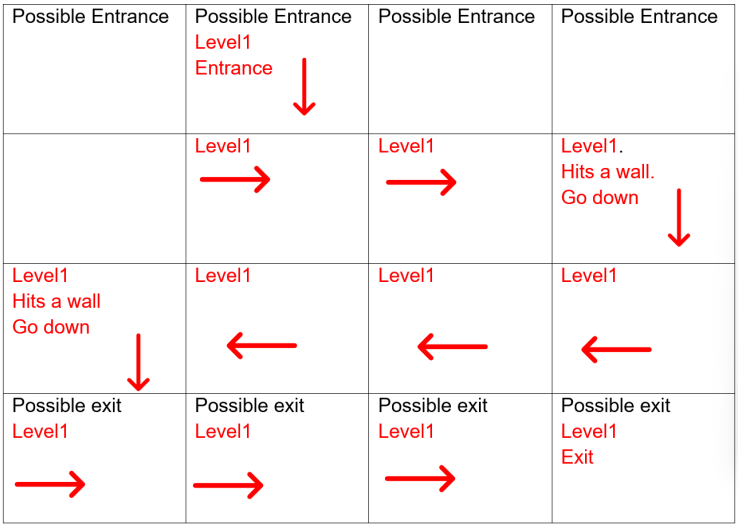
First simulated noise made by Perlin based that can be used for creating landscapes or any texture using the wavelengths of sound. The random numbers produced by these wavelengths range between 0 and 1. If we were to get a more natural landscape or texture, we would have to look into smaller numbers (0.05 instead of 0.5), or we would have to reduce the amplitude and frequency of the waves. This noise can be created artificially (noise generating formulas) or picked up by a microphone.

2 Research

2.1 Games using procedural generation

There is an array of games dating back to the 70s and 80s using the first very basic procedural generation with pseudo number generation algorithms. They tried to make maps like roguelike games and the first using procedural generation in 3D, The Sentinel, claiming it has 10,000 levels only taking a maximum of 64 kilobytes. I have compiled a list of games featuring procedural generation that I think are worth mentioning.

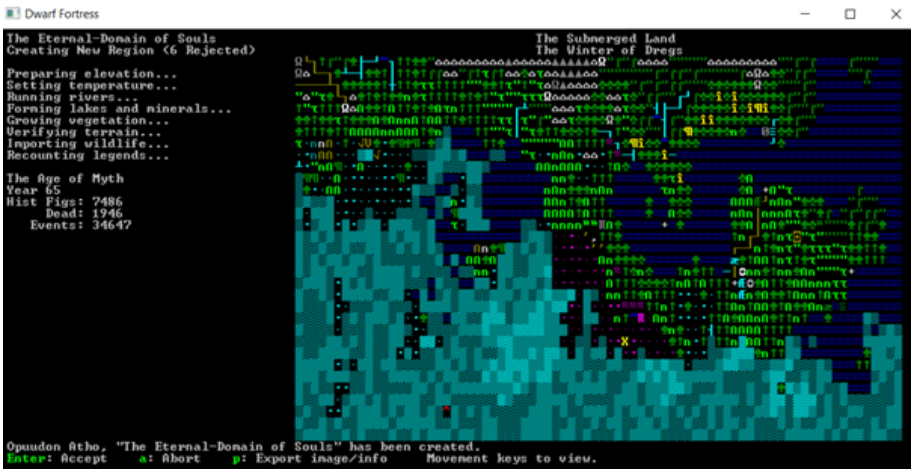
- Spelunky (4): a 2D platformer game where levels are generated every time you die. These levels are never memorised, and they also do not follow a completely random generation, which makes Spelunky special. The level maps are always generated following a set of rules. The level is divided into a 4x4 grid, thus giving it 16 different rooms populated using different templates, so it does not look too weird. The entrance starts at a random room at the top and starts a path of going down, left, or right. If it hits a wall, it will always go down. If it reaches the bottom grids and a wall, it will exit the level there.



In the level 1 example, the only time where it goes down randomly is from the entrance room. Later, it only goes down whenever it hits a wall. We also must remember that this level will never be generated the same if we die and try to play it again.

- Dwarf Fortress (4): a 2D simulator game that creates worlds depending on a few parameters inputted by the user. Each world is unique to every user, and its development constantly varies, giving birth to entirely random and unexpected events within the world affecting the life of the dwarfs. This game heavily influenced Minecraft.

Once the game is created, the years go by, and events happen. You can then stop the process and scroll around the different locations that have been made.



There are many different layers of systems in this world that help shape it.

2.2 Map Magic 2

Map Magic is one of the most popular terrain generation tools in Unity along with Gaia. It utilises a node-based system with a graphical Editor, in which you can link up multiple components to create a terrain. Unfortunately, we cannot create rivers

without purchasing an add-on to the tool called "MapMagic 2: Splines".

From experimenting with the tool, the controls felt unintuitive. Unable to MouseLook with the middle mouse button, instead having to hold the M key, which is quite far from the WASD movement keys. Fullscreen in most other applications and games tends to be one of the F keys or Alt+Enter, however, in Map Magic 2, it is the letter F.

Using the tool felt slightly overwhelming, however, the node approach is compelling, and it is possible to create specific terrain to suit user needs.

2.3 Hal Paper

It is a published paper about Hydraulic erosion(5) that covers some of the different ways scientists in the past have used to simulate water erosion on terrains using models like diffusion and fluid simulations with particle systems. They present their novel algorithm in the field that takes advantage of the power of dedicated computer graphics (GPU's) and focuses greater on the velocity of the flowing water.

"We represent terrain and water surfaces as height fields on 2D regular grids."(5) Their solution uses layers of 2D arrays, the water, and the terrain. Both entities carry properties like water level d and terrain level b. They both update simultaneously and perform erosion, evaporation, segmentation, and deposition. Many examples that apply erosion to terrain, use this paper as the basis for their logic and application. For example, the portable library called TinyErode(6), which is written in C++. It looks like most papers use the same method for erosion.

2.4 WebGL Erosion Sandbox Web Demo

It is a terrain tool written with typescript that uses the same erosion formulas presented in the Hal Paper. It was developed by LanLou123(7), and runs on WebGL inside a browser. It makes it highly accessible and easy to experiment with.

2.5 Implementation of a method for hydraulic erosion (Paper)

This is a research paper written by Hans Theobald Beyer(8) for simulating erosion. They consider previous erosion implementations like the one developed in the Hal paper(5), however, they try to develop a more straightforward technique without simulating flow maps. Instead, they try to simulate a raindrop at a time that erodes the terrain.

3 Development

This chapter covers the tool's implementation: explanations of features, code snippets, and the different elements of the terrain. The project is created inside Unity Engine because of how easy it is to start with and the proficiency in C# that I hold. In addition, Unity has many built-in systems and functions, such as a rendering pipeline and a physics system. The tool I have developed can procedurally generate a map at the click of a button using a seed and multiple other parameters. I will be covering all the features in my procedural map generator.

3.1 Unity Terrain

At the beginning of the project, I had tried rendering my own terrain by creating my own terrain meshes, using Unity vertices and triangles. After experimenting with this and creating a simple terrain, I realised that I would be limited in adding more advanced features for map creation as I would need to build everything for my custom mesh.

However, Unity has powerful objects called Terrains, which holds a lot of data with everything necessary to make terrains. The data held includes alpha maps, height maps, UV maps, texture maps, and more. It also facilitates easy access to all this data, provides good documentation and functions. This data is accessible through TerrainData(9) within the Terrain GameObject. In addition, there is a multitude of online resources with different tips and problems in connection to Terrains within Unity. Further solidifying my choice in using Unity Terrain.

3.2 Random Seed

In many games using procedural map generation, players can share a generated map by sharing a seed used to generate the map. A seed ensures that a procedurally generated map can be recreated given the same input parameters and seed.

To create a map with random properties means we need to use random values. Supposing we use the Unity random function. In that case, these random values will be different across devices and project reboots/reloads, as Unity initialises Random with a high-entropy seed from the operating system and stores it in native memory. Resulting in random values, which are always the same until Unity decides to initialise random again, making the tool relatively static.

Instead, this tool allows the user to provide a seed, which is then used to initialise Random using Random.InitState(). Now for any calls made to Random, Unity will use the seed provided to ensure it returns the same number.

3.3 Creating Terrain

As first introduced in 1.2.4, the terrain generation tool heavily relies on using Perlin noise to create the initial landscape. Applying Perlin noise is a popular method used for semi-realistic terrain creation and is well documented. The key to creating realistic appearing terrain is to layer multiple levels of Perlin noise at different frequencies and wavelengths.

3.3.1 Initial Terrain

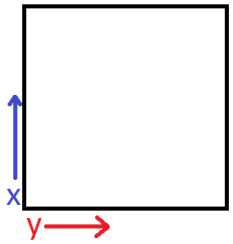
In the beginning, we retrieve the heightmap resolution of the terrain as set by the user. In Unity, heightmap resolutions are defined as $2^n + 1$ up to 4097. A slider is exposed to the user in Unity ranging from 1-12, and in the code, the resulting resolution is created by bit-shifting 1, by the detail level the user passes. Bit shifting by 3 is the equivalent to 2^3 and so forth. Finally, 1 is added to the resolution.

Next, using Random.Range a random offset is made for the x and y of the terrain, which will retrieve Perlin values. This is done so that each seed will generate a unique terrain.

The Terrain size is then set via TerrainData by inputting the width, max-height, and height. The terrain is formed by setting different heights in the heightmap of the

TerrainData. The heightmap stores values between 0-1, 0 representing the lowest possible height, and 1 representing the set max-height of the terrain.

The initial terrain formed is a wavy terrain using Perlin noise: Iterating through every point in the heightmap, using two loops for x and y.



The resulting generated terrain can be seen in figure 3.1.

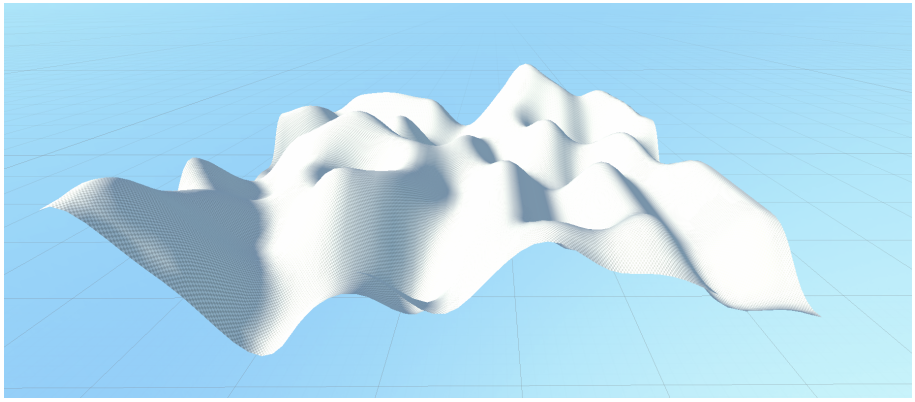


Figure 3.1: Initial terrain formed using Perlin noise

In addition, the user can set both the frequency and wavelength to be used, which will adjust the scale of the mountains and how many mountains are there.

3.3.2 Islands

To further create realistic terrain and provide more options to the user, an island feature is available. Creating realistic island terrain while adding a plane of water completes the map.

In naturally formed islands, the highest point is always at the centre of the island. Following the logic suggested by KrazyTheFox(10), a mountain can be created in the middle using gradients, calculated from the central point and travelling outwards. In the centre of the island, the distance to the highest point would be 0, meaning we apply the total size of the Perlin noise. As we move outwards further away from the highest point, the distance is subtracted from height, creating a mountain in the centre of the island. A variable called distanceToCentre is used, which uses the distance formula to get the distance between two points 3.2. The centre of the island is

picked using a random number within the heightmap resolution, with x and y.

Distance formula

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Figure 3.2: Distance Formula.

The resulting distance is stored in a 2D gradient array that is identical in size to the heightmap resolution. When applying the values returned from Perlin noise from the initial terrain in 3.3.1 we minus the distance from that height, which then results in the correct height being set.

The resulting outcome can be seen in figure 3.3. As you can see all of the other terrain is lowered relatively from the highest point.

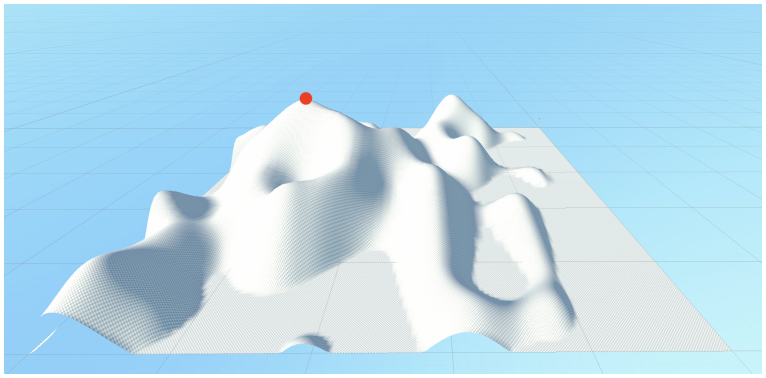
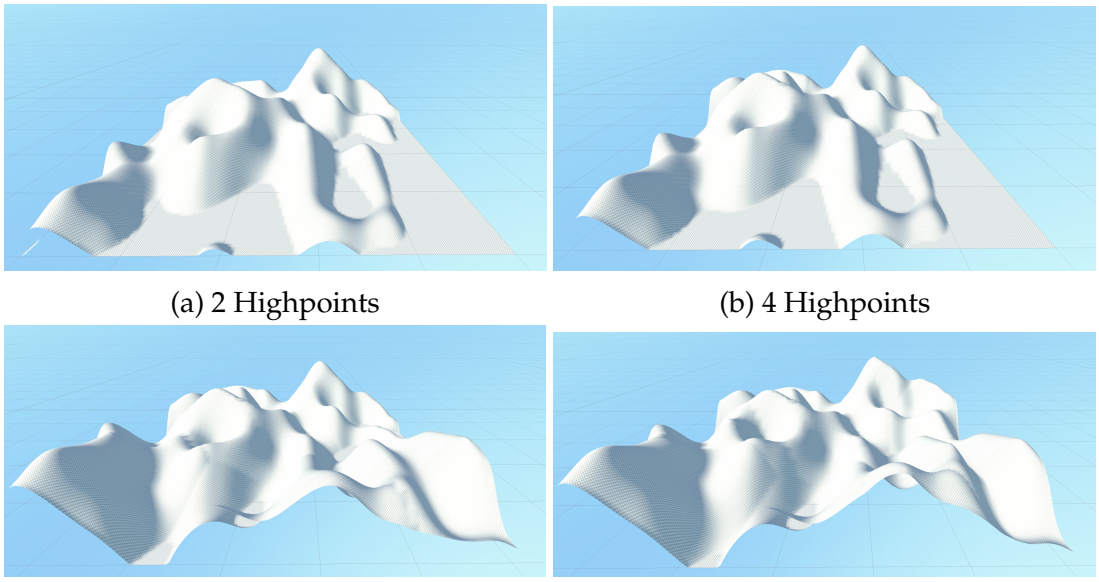


Figure 3.3: Single highpoint set. The red point is the highpoint

Further expanding the idea of islands, multiple islands can be created by selecting multiple highpoints. The user can select as many highpoints as they want, and an according terrain is generated as in figure 3.4. To implement this, the same process as described before is done for the number of highpoints wanted, with the distance data stored in the gradient array being overwritten only if it is smaller than what is currently stored. Smaller because we minus the distance from the current height.



(a) 2 Highpoints (b) 4 Highpoints

(c) 8 Highpoints (d) 16 Highpoints

Figure 3.4: Results of different highpoints

3.3.3 Distortion Layering

Generating a terrain from a single Perlin noise pass-through does not create a dynamic or realistic terrain, as terrains have mountainous and flat regions. Within a mountain itself, it would have places where it is more rough and other places where it is smoother.

This can be achieved by layering multiple Perlin noise maps together, with differing frequencies and wavelengths as in figure 3.5. The distortion can be either added or subtracted from the current value in the heightmap.

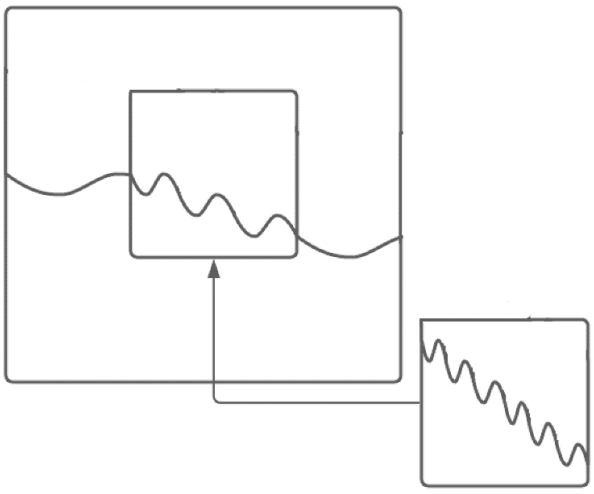


Figure 3.5: Perlin noise is added to the existing noise at a higher frequency to form more realistic mountain terrain.

During development, I found that subtracting from the current height gives a better result, as once the minimum heightmap value of 0 is reached, it results in flat terrain. Using addition could mean that it would reach the max heightmap value of 1, resulting in terrain with mountains that have completely flat peaks as in figure 3.6.

```
for(int i = 1; i < distortionWavelength.Length; i++)
{
    //x passes, distortion for nicer grainier mountains and more realistic mountains
    for (int x = 0; x < resolution; x++)
    {
        for (int y = 0; y < resolution; y++) {

            // we can substract, multiply or add, they all work to make distorsion, two parameters scale and frequency
            // prefer to subtract as hitting the maximum height will make very weird mountains, however 0 height is just
            // flat terrain
            heights[x, y] -= CalculatePerlin(x, y, distortionWavelength[i], distortionFrequency[i]);
        }// for y
    }// for x
} //for i
```

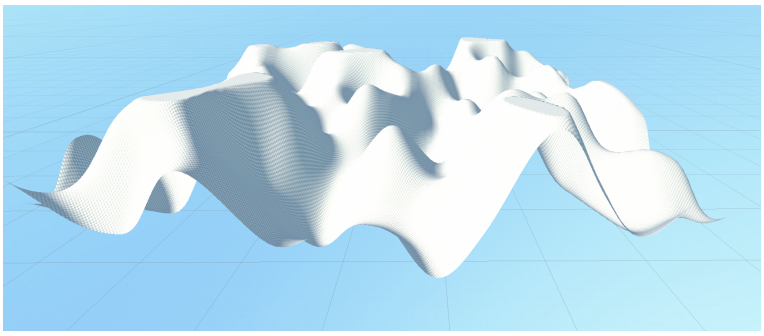
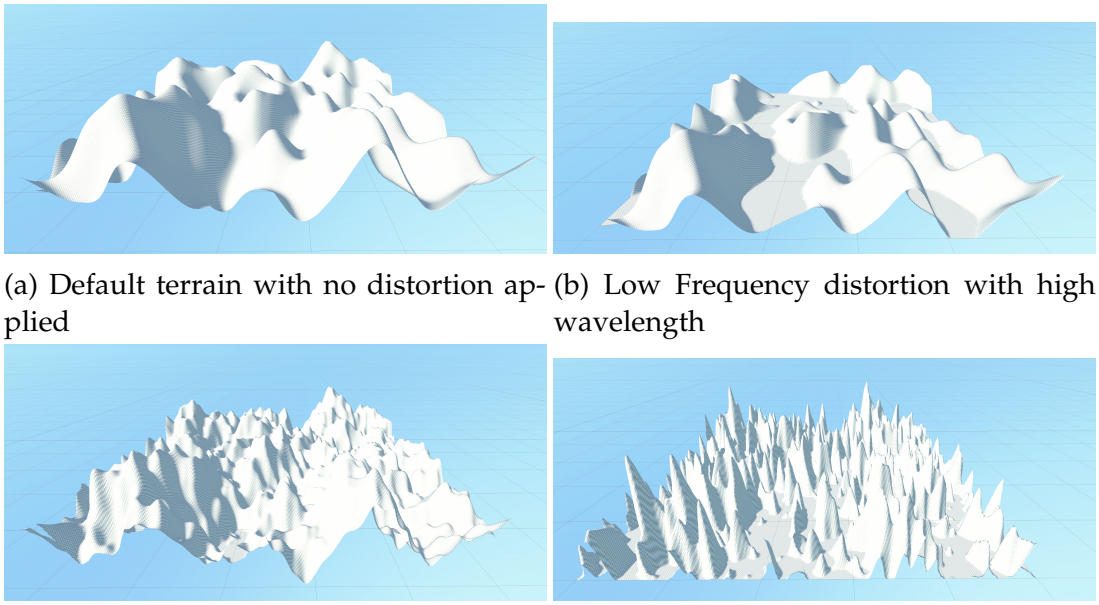


Figure 3.6: Distortion noise is added to the current heights resulting in flat peaks

The results of layering an additional Perlin noise distortion can be seen in figure 3.7. Adjusting the tool's parameters can now result in much more mountainous mountains, flat areas, or hills with valleys.



(a) Default terrain with no distortion applied

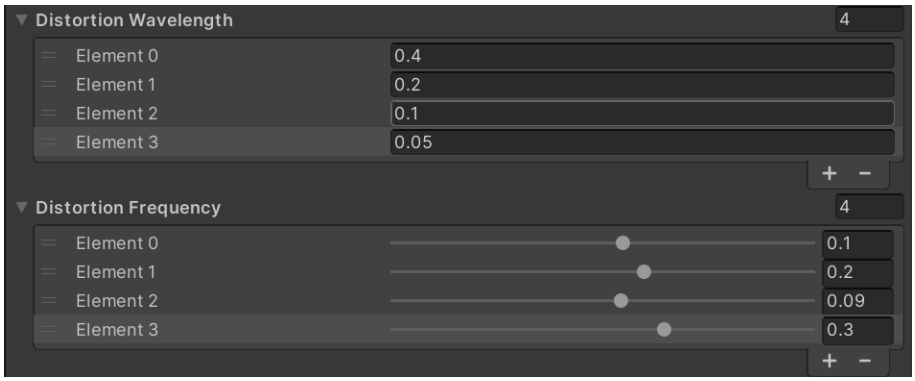
(b) Low Frequency distortion with high wavelength

(c) High frequency distortion with low wavelength

(d) High frequency distortion with high wavelength

Figure 3.7: Double layer distortion parameters

Having the ability to use multiple distortions dramatically enhances the possibilities of maps that can be generated. This tool allows using any number of defined frequencies and wavelengths to terraform the terrain. They are stored in distortionWavelegnth and distortionFrequency arrays.



3.4 Smoothing the Mountains

To smooth the terrain, the approach taken loops through every point in the heightmap, and the neighbouring points are found in a 1 point radius. Each point's height is retrieved and added to a temporary averageHeight variable for the point. Finally, the height of the point is set as the average, where the averageHeight is divided by the number of neighbours which were used. If a neighbouring point goes

outside the terrain, it is not added, and the number of compared neighbours is less. Importantly, this smoothing implementation does not wrap around the map, as the terrain itself does not wrap around the map. If it did, it could create steep terrain on the edges of the map.

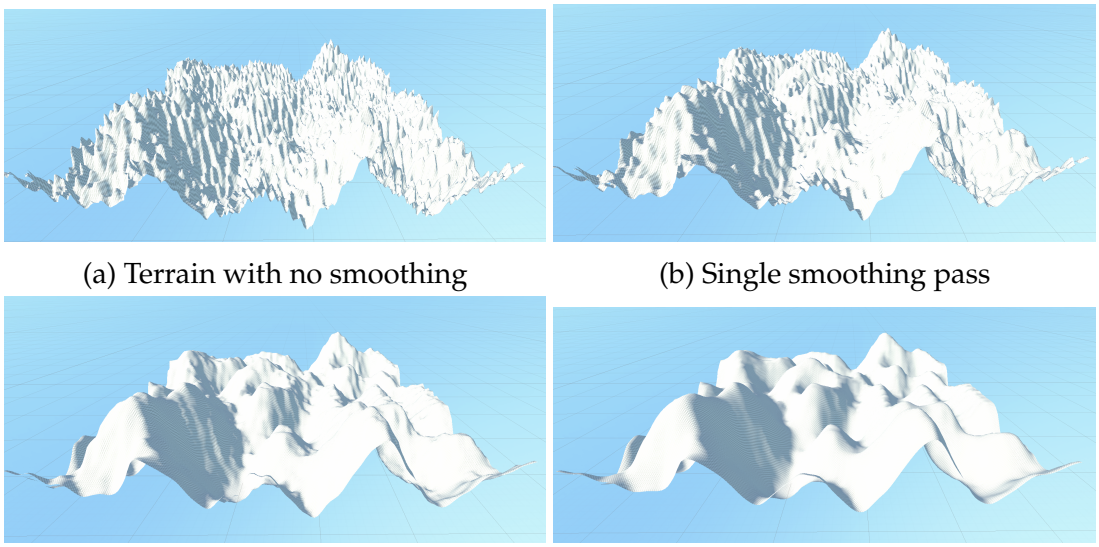
```
//keep track of the total height to average
float averageHeight = 0f;
//track neighbours used to divide for average
int neighbours = 0;
//neighbours
for (int xOffset = -1; xOffset <= 1; xOffset++)
    {
        for (int yOffset = -1; yOffset <= 1; yOffset++)
        {
            // finding the index of the neighbour we will be comparing
            int tempX = (x + xOffset);
            int tempY = (y + yOffset);

            // ensuring we don't wrap around and go out of bounds
            if (tempX < 0 || tempY < 0 || tempX > (resolution - 1) || tempY > (resolution - 1))
            {
                continue;
            }

            averageHeight += copiedHeights[tempX, tempY];
            neighbours++;
        } //for
} //for

// dividing and setting the new average height
copiedHeights[x , y] = averageHeight / neighbours;
```

This is then repeated for the number of smoothing passes the user specifies, which can mean no smoothing at all or a lot of smoothing. Looking at figure 3.8 you can see the different effects on the terrain.



(a) Terrain with no smoothing

(b) Single smoothing pass

(c) Four smoothing passes

(d) Eight smoothing passes

Figure 3.8: Smoothing effect on a rough Terrain

3.5 Rivers

In the natural world, terrains that form both mountains and flat areas usually contain rivers. Adding rivers to the tool proved to be a difficult task. However, I will outline the logic and techniques used to create rivers. The script Rivers.cs determines how the rivers are formed and rendered.

3.5.1 River Path Generation

The user can define how many rivers they want to be created in the terrain, ranging from no rivers at all to hundreds. The main TerrainCreator script calls to the Rivers script, passing the TerrainData as well as numberOfRivers.

In nature, rivers usually follow a set of rules:

- Originate in the mountains.
- Flow downhill.
- Follow the easiest path downhill.
- Narrower in steeper, fast moving areas and wider in flat areas.

The algorithm created did not follow all of the rules determined. However, it did apply the most important ones. Since the user can determine any amount of rivers wanted, it would not be practical to select only the highest points as the terrain could possibly not have enough high points for the number of rivers the user sets.

The algorithm created is as follows:

1. Choose a random x and y as a starting point of the river on the terrain.
2. To determine the next point the river will move to, analyse all of the neighbouring points
3. From the neighbouring points, select the steepest neighbour which is also lower than the current point
4. Record the next point taken in the river path array, without modifying the current heightmap data.
5. Repeat for the length of the river or stop the river if the terrain becomes too flat.
6. Repeat for the number of rivers wanted.

All of the riverPaths are recorded in a separate 2D array which mimics the resolution. This is done so as not to modify the existing heightmaps during river generation.

Having faced issues previously where if we modify the heightmap, the next lowest and steepest point will move back to the point it came from. Recording this information in a separate array prevents this.

Finally, the riverPaths array is looped through, and anywhere where the value is not zero, means a river is supposed to be placed. The same coordinates in the heightmap are accessed and reduced to indent the terrain and signify a river.

```
// of the 3d array, for each river, go through the 2D array of the map and check that
// there is a river at that point
// if the value is more than 0, means the river goes through the point and the height
// is reduced at that point
for (int i = 0; i < riverPaths.GetLength(0); i++)
{
    for (int x = 0; x < copiedHeights.GetLength(0); x++)
    {
        for (int y = 0; y < copiedHeights.GetLength(1); y++)
        {
            if (riverPaths[i, x, y] > 0)
            {
                copiedHeights[x, y] -= 0.05f;
            }
        }
    }
}
```

The outcome of the river generation can be seen in figure 3.9.

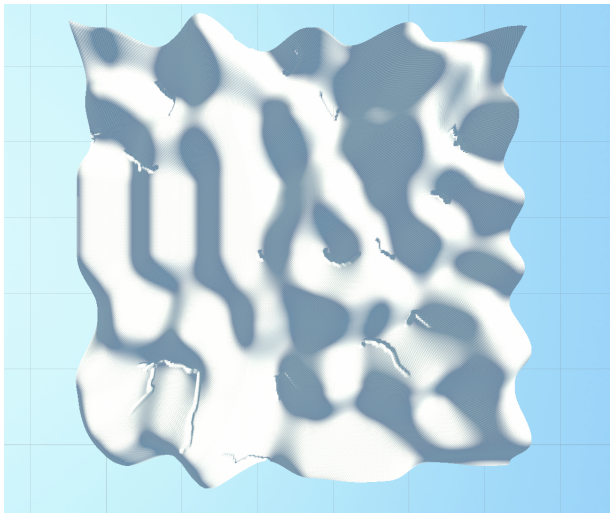


Figure 3.9: Rivers created with the River.cs script

3.5.2 Procedural Water Mesh Creation

The rivers created previously become alive when the appropriate meshes and textures are applied to them. To achieve this, low-level geometry needs to be utilised within Unity. Creating our own meshes, creating GameObjects, and assigning the parameters.

To create a custom mesh in Unity, two key components need to be assigned to the mesh, the vertices, and the triangles. In a mesh, the vertices are points inside the mesh, while the triangles hold information about how each vertice connects to create a polygon. To create a single polygon, 3 vertices are needed. In Unity, the triangles array holds the order of the vertices in which the polygon will be rendered.

Since a river is a meandering line, creating a mesh using squares is possible utilising two polygons each. During the riverPath creation, we utilise a 3D array to store the path of each river, knowing the direction of flow as the ints increase from 1 to the river end. Afterwards, the river's total length is stored inside the RiverLengths array, as it is necessary to create the vertex and triangles array before populating it with information.

The number of vertices needed is the river's length multiplied by two as we need a vertex on each side of the river bank for every point. A single square needs 4 vertices to make a polygon. Looking at figure 3.10 you can see the first (black) point there are two vertices, and then on the next (black) point, two more are created. The number of triangles necessary is 6 for each square since every square is made from two polygons that need 3 triangles each. So the total number of triangles necessary is 6 times the total length of the river minus one, since when you reach the end of the river, there is no further polygons to render. Further referring to figure 3.10 the black polygon has vertices 0, 2, and 3 while the red polygon has vertices 0, 3, and 1.

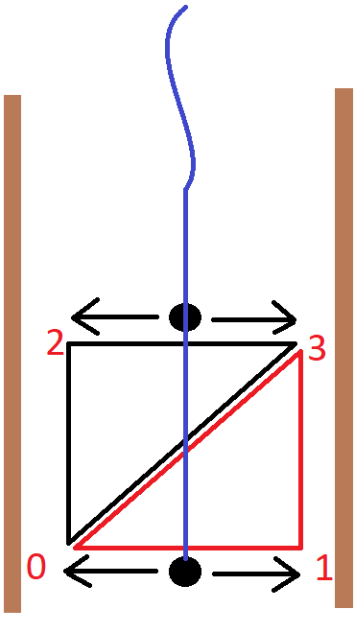


Figure 3.10: How vertices are determined and polygons drawn

The brown lines in the figure 3.10 represent the banks of the river, the terrain mesh. To find the position of each vertex, a raycast needs to be sent perpendicular to the

direction of the river, to know where on the bank to render the mesh. To do this, the next point in the river is found, from which subtracting the two positions results in the direction between the two points. With the direction determined, a perpendicular direction can be calculated using methods outlined in a Unity forum post(11). A raycast is then cast to the left and right of the river. Once it collides with the mesh of the terrain, the point becomes a vertex of the river. A maximum raycast distance of 5 is set in case the terrain is not hit.

```
Vector3 origin = new Vector3(terrainData.size.x * (currentX / resolution),
Vector3 destination = new Vector3(terrainData.size.x * (nextX / resolution)
Vector3 direction = (destination - origin).normalized;

Vector3 left = Vector3.Cross(direction, Vector3.up).normalized;
Vector3 right = -left;


Physics.Raycast((origin + new Vector3(0,3,0)), left, out hit, 5);
vertices[j] = hit.point;

j++;
Physics.Raycast((origin + new Vector3(0,3,0)), right, out hit, 5);
vertices[j] = hit.point;

currentPosition++;
```

This process is repeated for the river's length, creating the necessary amount of vertices, two for each point on the river.

Finally, the triangles need to be assigned vertices in order to render them correctly. To do this 6 triangles are assigned to make two polygons which form a square-like shape.

In Unity, the triangles need to be assigned in a clockwise direction to render upwards towards the camera. To render in a clockwise direction, referring to figure 3.10 a set of triangles are assigned as follows.

```
for (int vertex = 0; triangle < triangles.Length - 1; vertex+=2)
{
    //first triangle
    triangles[triangle + 0] = vertex + 0;
    triangles[triangle + 1] = vertex + 2;
    triangles[triangle + 2] = vertex + 3;
    //second triangle
    triangles[triangle + 3] = vertex + 0;
    triangles[triangle + 4] = vertex + 3;
    triangles[triangle + 5] = vertex + 1;
    //square formed

    //increment triangles as done 6 points
    triangle += 6;

} //for
```

This process is repeated for the river's length, for the number of squares necessary.

Once the vertices and polygons are completed, a Mesh type object is created to which these values are assigned. Finally, a "River" GameObject is created in Unity, and the mesh is assigned to the object. To prevent these objects from persisting every time we generate a new terrain, they are found and destroyed from the scene at the very beginning in TerrainCreator.cs.

3.6 Hydraulic Erosion

This type of erosion happens when rain droplets fall onto the terrain. The user can specify the number of raindrops the terrain will receive and, in effect, is eroded by. The implementation used inside the tool is based on the research paper written by Hans Theobald Beye(8) as well as the overview and introduction provided by Sebastian Lague(12).

The logic steps in creating erosion by water are

1. Choose a random x and y point on the map, on which the droplet lands.
2. Determine the gradient of the terrain where the droplet is, and get the direction it is moving in. The direction is multiplied by a variable called inertia, which can make the droplet move in the same direction it came from (even if it's uphill), or move downwards in the downhill direction.
3. Move the droplet in the direction, we always move by a single square, not skipping any
4. If the droplet is moving out of bounds, or has no direction stop simulating it
5. Find the new height of the droplet at the location it moved to, and get the change in height. A negative value means we moved downwards, while a positive value means we went upwards
6. Calculate how much soil the droplet can carry, based on the height change, speed, water content and soil capacity factor.
7. If the droplet carries more soil than the capacity, or if it moves uphill some will be deposited
8. If going up a hill, deposit equal amount of soil for the height change, or if there isn't enough soil left for the whole height change, then deposit all of it.
9. If not moving uphill, deposit some of the excess soil determined by the depositSpeed

10. However if the droplet isn't moving uphill or has less soil than the capacity the terrain will be eroded
11. During erosion, we erode by the capacity the droplet has left controlled by an erosionSpeed, or by the height change. We cannot erode more than the height change as it would leave big holes in the terrain.
12. Once the erosion amount is determined, the previous point is eroded by that amount, which means minusing from it's height and adding the change to the current soil the droplet is carrying.
13. Finally the speed of the droplet is updated, as well as the current water content using the formulas provided by Hans(8).
14. This is repeated for the lifetime of the droplet. Furthermore, this is then repeated for the number of droplets the user specified

The resulting terrain is eroded. As the implementation is based on the method outlined by Hans(8), the same parameters are utilised and exposed to the user to be able to modify the erosion.

3.7 Texturing the Terrain

In expanding the tool further, the final step is applying textures to the terrain. A procedural map generation tool would not be complete if it only produced a terrain mesh, with no textures on it, appearing blank. The aim is to colour the terrain with different textures depending on the point's height, position, and circumstance. The starting point used for this section can be found in the Unity documentation of SetAlphamaps(13) and also with the help of a Unity forum post (14).

Unfortunately, there is not extensive documentation by Unity on how to assign textures and use splatmaps to create alphamap layers for the terrain. Looking through the code example provided in the Unity forum post (14), the steps taken to texture the terrain in this tool are as follows

- The user needs to assign textures as layers to the terrain property. The ColourTerrain.cs script uses three textures.
- Normalise all values to ensure being able to apply values as percentages.
- Lopping through each point in the alphamap using x and y, there is also a z value stored, a float array, for the length of the number of layers in the layer palette.

- Every z-axis has a percentage weight of the texture used, making it easy to blend different textures if needed.
- For each value stored in the z float array, which represents a texture, it contains a percentage weight representing how much of the texture to apply.
- Assign texture percentages based on parameters and rules to texture the terrain appropriately.

This way, the algorithm can apply textures depending on the height of the terrain. Additionally, the maxAltitude and minAltitude variables could be exposed to the Inspector GUI by passing the values in the method call. However, this is not currently implemented. Referring to figure 3.11, it is illustrated how terrains can be blended as well as defined to apply certain textures specific heights.

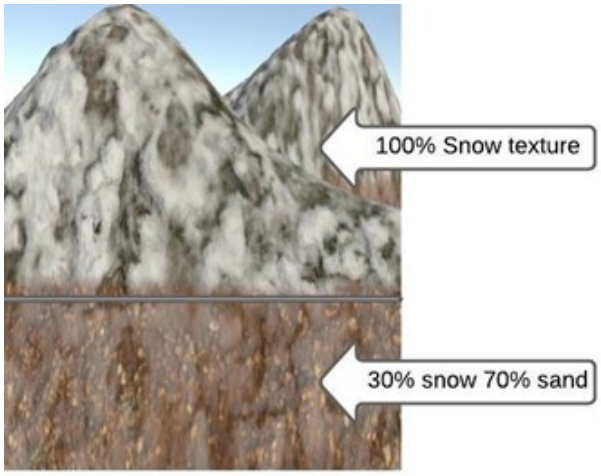


Figure 3.11: Texture application based on heights as well as texture blending

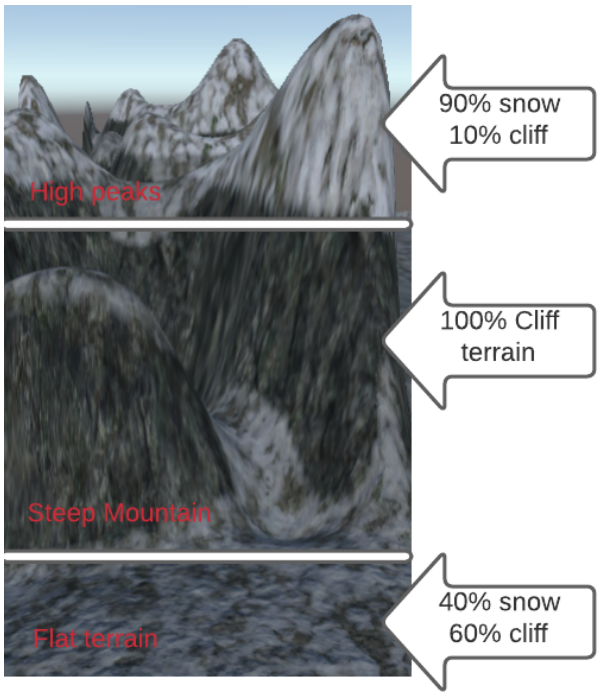
The texture colouring algorithm can be summarised with the following parts:

- Variables: rules variables, alphamap 3D array, weight array
- Loop: iteration of the x and y points of the alphamap
- Calculations: Normalisation of variables, rules themselves(height, steepness)

3.7.1 Steepness

The method terrainData.GetSteepness returns the steepness of the terrain at a given point. Unfortunately, Unity's documentation about the GetSteepeness function is lacking. However, a particular Unity forum post (15) mentions that with some experimentation, they had determined that it returns a value between 0 and 90, which represents the degrees.

An absolute right angle is an angle of 90 degrees, which is why the maximum steepness is 90. While texturing the terrain, the result from GetSteepness is normalised, where the absolute right angle is 1, and no angle is 0. This makes it easier to set and determine the parameter for the texturing. The point is then compared to a minimum and a maximum steepness set up as a parameter variable, and a texture is assigned accordingly.



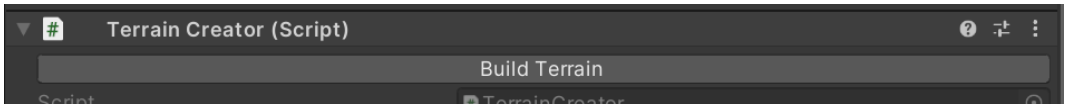
3.8 Unity Editor

Using Unity Editor, it is possible to implement some custom UI and function calls to enable terrain generation in Editor mode without pressing play inside Unity. Achieving this, an additional script is added, which extends Editor from UnityEditor. This script is placed in a folder called Editor in order for Unity to pick it up. The definition at the top of the file determines that this is a custom editor for the TerrainCreator script. It is overriding the method OnInspectorGui(), which Unity uses to render the UI in the inspector for this particular script. In the inspector GUI, a button is created called "Build Terrain" when this button is clicked in the inspector, the method BuildTerrain() within the TerrainCreator script is called.

This allows the user to quickly adjust the parameters inside the inspector, and press a button to generate a new terrain. This greatly speeds up the process rather than pressing play inside Unity.

```
using UnityEngine;
using UnityEditor;

[CustomEditor(typeof(TerrainCreator))]
Unity Script | 0 references
public class TerrainCreatorEditor : Editor
{
    0 references
    override public void OnInspectorGUI()
    {
        //serializedObject.Update
        TerrainCreator terrainCreator = (TerrainCreator)target;
        if (GUILayout.Button("Build Terrain"))
        {
            terrainCreator.BuildTerrain();
        }
        DrawDefaultInspector();
    }
}
```



3.9 Camera Implementation

The camera used in the play state of the tool features the typical W, A, S, D keys for movement. The user can fly upwards using the space bar, while the mouse is used to look around. Inside play-mode, the cursor is hidden with *Cursor.visible = false;* and locked with *Cursor.lockState = CursorLockMode.Locked;* so it doesn't go outside the game screen.

3.9.1 Camera clamping and locking

Using InputAxis, for the Mouse X and Y, the raw data from the mouse input is received. However, if applying this directly to the camera, the user could spin the camera off-axis and cause confusion in movement, position in world-space as well as dizziness.

To prevent spinning out of the Y-axis, the Y angle of the input is clamped, so the camera cannot go upside down. The minimum and maximum limits determine this. After which, Mathf.Clamp() is used to return a value between the minimum and maximum.

```
1 reference
float ClampAngle(float angle, float min, float max)
{
    if (angle < -360f)
    {
        angle += 360f;
    }
    if (angle > 360)
    {
        angle -= 360f;
    }
    return Mathf.Clamp(angle, min, max);
}
```

3.9.2 Speed variant

The user can move inside the game at two different pre-determined speeds, normalSpeedControl, and fastSpeedControl. By default, normalSpeedControl is used. However, as long as the user holds down the shift key, the fastSpeedControl is applied for movement.

```
// speed up camera
if (Input.GetKey(KeyCode.LeftShift))
{
    controlSpeed = fastControlSpeed;
}
else
{
    controlSpeed = normalControlSpeed;
}
```

4 Results

This chapter will cover the results of my work, as well as the issues and challenges faced. The created tool has a lot of potential. However, there were some aspects that I was not able to resolve.

4.1 Issues

While creating the procedural map generation tool, I was unable to implement some features desired fully. Instead, there is a partial implementation, and this section will cover it.

4.1.1 Hydraulic Erosion

Creating erosion in the terrain was quite difficult. There are a lot of methods and resources available however it is highly complex. As mentioned before, I attempted to implement the method as Hans(8) described in their paper through the help of Sebastian(12). When running erosion, my implementation results in multiple peaks and dips as in figure 4.1

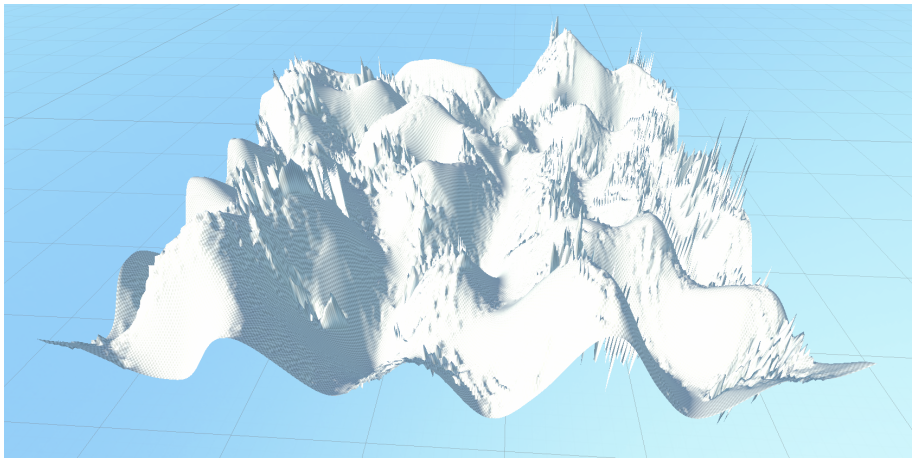
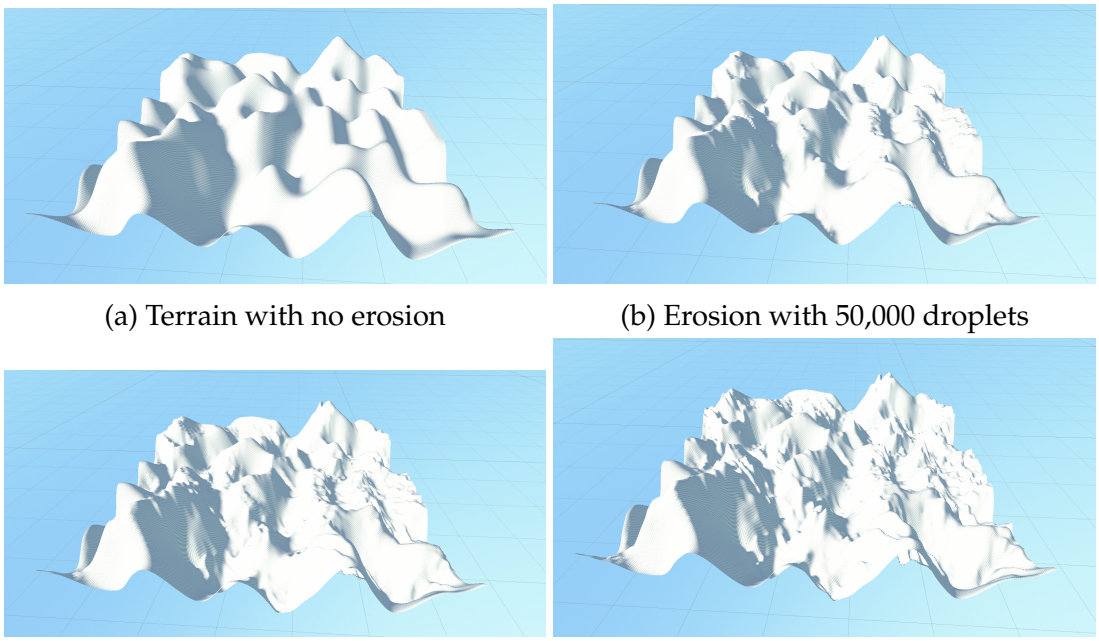


Figure 4.1: Erosion on terrain implementation with 20,000 droplets

I could not resolve this issue as I lacked a complete understanding of the erosion process and how to emulate it on computers. Using flow maps and calculating gradients proved a problematic concept. However, as a compromise, if applying smoothing after my erosion step, I could create eroded-like appearing terrains. The result is not perfect, but it does create more dynamic results and looks reasonably good.



(a) Terrain with no erosion

(b) Erosion with 50,000 droplets

(c) Erosion with 100,000 droplets

(d) Erosion with 200,000 droplets

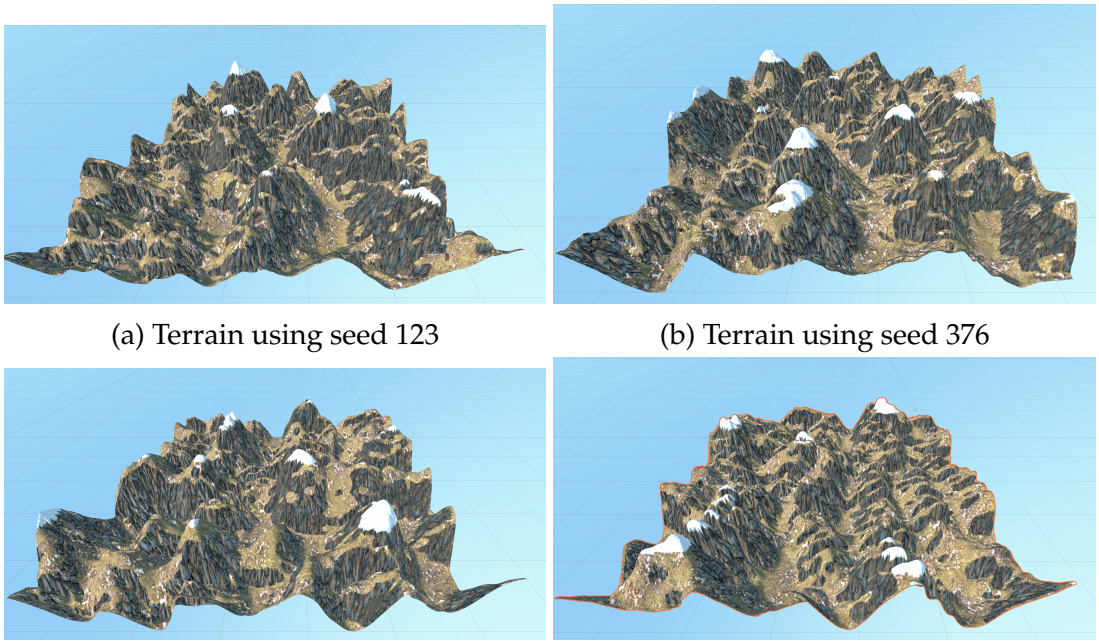
Figure 4.2: Erosion on terrain with 3 smoothing passes

4.1.2 Rivers

Creating rivers posed a difficult challenge as well, especially generating the meshes. As mentioned previously, I had some experience with Unity mesh generation after trying to make my own terrain mesh by following an outline from Brackeys(16). However, after attempting to generate the meshes, I had no success. There would not be any meshes rendered in the terrain. Instead, there would be GameObjects with the meshes attached to them. However, these meshes did not show.

4.2 Outcomes

The created tool can procedurally generate interesting terrains and give much control to the user over what they want. Here are a few examples of generated terrain, all using the same parameters but different seeds seen in figure 4.3.



(a) Terrain using seed 123

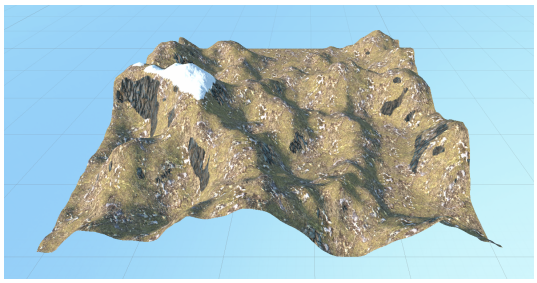
(b) Terrain using seed 376

(c) Terrain using seed 550

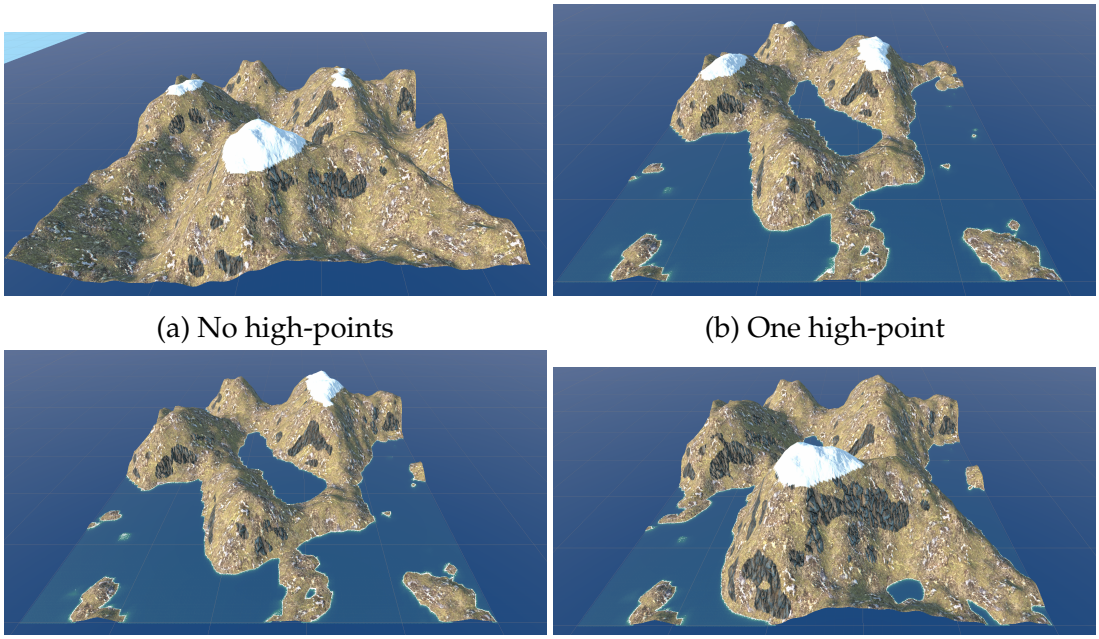
(d) Terrain using seed 741

Figure 4.3: Procedurally generated map using identical parameters except changing the seed

Furthermore, it can create hills with valleys, resulting in flatter overall terrain by adjusting the parameters of layered distortions.



Following, here are some island terrains generated using the tool, and the water is a plane with a water material applied. Placing the water slightly above the lowest level of the terrain and utilising the high-points feature. All the terrains in the figure 4.4 use the same parameters to create islands but have differing amounts of high-points.



(a) No high-points (b) One high-point

(c) Two high-points (d) Four high-points

Figure 4.4: Procedurally Generated Island terrains with differing high-points

The result is a procedural map generation tool that uses deterministic generation to create terrains. It is powerful and customisable. However, it does have a few shortfalls. Overall many interesting terrains can be created fully textured.

5 Conclusion

5.1 Future Work

The tool I created is incredibly flexible and can generate a multitude of different maps. However, there is much further potential in the tool.

The highest priority work would be to fix and fully implement rivers and erosion. Regarding the rivers, to fix the mesh generation and expand the rule-set of river spawning. This would include the rivers becoming wider in flatter areas, narrower in steeper areas, and creating pools/lakes at the bottom if the terrain allows. Moving onto erosion, to further explore and understand erosion techniques, to be able to fully simulate erosion on the procedurally generated maps. Perhaps explore more complete erosion simulations.

The texturing of the terrain holds a multitude of possibilities. Currently, only 3 textures are supported. I could further expand this by allowing the user to input any number of textures through the Inspector. Furthermore, assign rules for each texture, like the height, steepness, or if they want it to blend with another texture within the Inspector as well, as currently, the code needs to be adjusted to change the rules.

Supporting the expanded texturing, the generation of biomes would greatly expand the potential. Follow rules of the natural world to generate different types of biomes, like sunshine constantly shinning in an area resulting in it become arid, or an area with a lot of rainfall and river down-flow resulting in rich, lush vegetation with lots of plants and trees. Along with other weather effects and simulations for more realistic maps. The textures are then applied appropriately.

Ideally, the tool would adopt a node-based system like Map Magic 2. Allowing to stack multiple different effects in any order desired to create more complex procedurally generated maps.

5.2 Conclusion and Thoughts

This is the most challenging project I have undertaken. It was time-consuming and very frustrating at times. Especially when features do not work as expected and spending time trying to fix the issues but not succeeding. I also learned to accept moving forward with the project onto other features because of time constraints.

I found the topic of procedural map generation interesting and very creative. The most fun aspect of the project was conceptualising and identifying the different steps of map generation. The different possible systems which could be combined to create beautiful and realistic terrains.

For the future, I do want to continue working on this tool, however, at my own pace. The circumstances of lockdown, COVID, and deadlines had made this a stressful experience. Not having the pressure to complete the tool will make it easier to come back to and approach and continue improving and finishing the tool.

There are still many concepts I heavily struggle to understand however, with more time and patience, I hope to get them more and finish the tool.

Furthermore, I feel that I made this more difficult for myself than necessary by jumping to complex concepts quickly. Features like the mesh generation for rivers and the erosion simulation. I wish I had tackled some other concepts firstly, like vegetation creation or forest spawning, to gain more experience first and provide more impressive initial visual upgrades.

With this tool, I hope to provide something that does not currently exist on the Unity asset place. This tool would provide erosion, biome generation, and river creation for free. As at the moment, the current procedural map generation tools with these features are paid.

Finally, hopefully inspire someone else to create an even more advanced tool from my work.

6 Assets

Here are the assets that were used in the project.

- Stylized Water For URP https://assetstore.unity.com/packages/vfx/shaders/stylized-water-for-urp-162025.
- Terrain Textures - 4K https://assetstore.unity.com/packages/2d/textures-materials/terrain-textures-4k-179139.
- Fantasy Skybox FREE https://assetstore.unity.com/packages/2d/textures-materials/sky/fantasy-skybox-free-18353

Bibliography

[1] Wikipedia. Conway's game of life, 2021. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[2] Wikipedia. L-system, 2021. https://en.wikipedia.org/wiki/L-system.

[3] Irene Cazzaro. Cellular automata between life science and parametric design: Examples of stochastic models to simulate natural processes and generate morphogenetic artefacts. *Advances in Intelligent Systems and Computing*, pages 632–643, 01 2019. doi: 10.1007/978-3-319-95588-9_52.

[4] Richard Moss. 7 uses of procedural generation that all developers should study, January 2016. https://www.gamasutra.com/view/news/262869/7_uses_of_procedural_generation_that_all_developers_should_study.php.

[5] Bao-Gang Hu Xing Mei, Philippe Decaudin. Fast hydraulic erosion simulation and visualization on gpu. *15th Pacific Conference on Computer Graphics and Applications (PG07)*, May 2011. doi: 10.1109/pg.2007.15.

[6] tay10r. Tinyerode, May 2021. https://github.com/tay10r/TinyErode.

[7] LanLou123. Webgl-erosion, April 2019. https://github.com/LanLou123/Webgl-Erosion.

[8] Hans Theobald Beyer. Implementation of a method for hydraulic erosion. November 2015. https://www.firespark.de/resources/downloads/implementation%20of%20a%20methode%20for%20hydraulic%20erosion.pdf.

[9] Unity. Terraindata.setalphamaps, August 2021. https://docs.unity3d.com/ScriptReference/TerrainData.SetAlphamaps.html.

[10] KrazyTheFox. Gradient island, June 2013. https://www.reddit.com/r/gamedev/comments/1g4eae/need_help_generating_an_island_using_perlin_noise/cagsrki?utm_source=share&utm_medium=web2x&context=3.

[11] Bunny83. Perpendicular to a 3d direction vector, March 2017. https://answers.unity.com/questions/1333667/perpendicular-to-a-3d-direction-vector.html.

[12] Sebastian Lague. Coding adventure: Hydraulic erosion, February 2019. https://www.youtube.com/watch?v=eaXk97ujbPQ.

[13] Unity. Terraindata, August 2021. https://docs.unity3d.com/ScriptReference/TerrainData.html.

[14] abertrand. Adding terrain textures procedurally, December 2015. https://forum.unity.com/threads/adding-terrain-textures-procedurally.374327/.

[15] ArtOfWarfare. terraindata.getsteepness() usage, December 2015. https://answers.unity.com/questions/452204/terraindatagetsteepness-usage.html.

[16] Brackeys. Procedural terrain in unity! - mesh generation, November 2018. https://www.youtube.com/watch?v=64NblGkAabk.