Mathematical Methods of Procedural Terrain Generation

Student: Rory Lyons
Supervisor: Mads Haahr

April 2024

Abstract

Context: Procedural content generation (PCG) stands at the forefront of modern interactive entertainment, offering innovative solutions to generate vast and diverse digital content dynamically. Through algorithms and rulesets, we can automatically generate anything from animals to entire planets. This makes it very appealing to smaller studios and solo developers who require large amounts of content but have access to fewer resources.

Objectives: To combine multiple methods of Procedural Content Generation in the creation of a single procedural game world while also describing some of the mathematics underpinning these methods.

Methods: A system for biome based procedural terrain generation with perlin noise at it's core has been created. Other techniques and algorithms were then incorporated into this foundation, such as wave function collapse for laying out biomes and marching cubes for rendering more 3 dimensional terrain.

Results: These methods were successfully integrated together with room for expansion in the future leading to a fully playable demo.

Conclusions: These methods I've outlined seem to integrate quite well together and are an effective method of generating large spaces with varied appearances and unique features. The systems I've created here are viable to used be used in future projects requiring procedural terrain, and can be fitted to serve a multitude of different types of experiences.

Plagiarism Declaration:
I have read and understood the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also read and understood the guide, and completed the 'Ready Steady Write' Tutorial on avoiding plagiarism, located at https://libguides.tcd.ie/academic-integrity/ready-steady-write.

Contents

1 Introduction **1**

2 Perlin Noise **2**
- 2.1 Background . . . 3
- 2.2 Parameters . . . 4
  - 2.2.1 Scale . . . 4
  - 2.2.2 Lacunarity . . . 5
  - 2.2.3 Octaves and Persistence . . . 5
  - 2.2.4 Final Noise Function . . . 6
- 2.3 Biomes . . . 6
- 2.4 How It's Evaluated and Results . . . 10

3 Mesh **11**
- 3.1 2D . . . 11
- 3.2 3D . . . 13
- 3.3 Marching Cubes . . . 13

4 Seeding **16**
- 4.1 Purpose . . . 16
- 4.2 Implementation . . . 16
- 4.3 Chunks . . . 17
- 4.4 Unique Chunk Seeds . . . 18

5 Interpolation **20**
- 5.1 Noise Value . . . 21
- 5.2 Colour and Foliage . . . 22

6 Wave Function Collapse **23**
- 6.1 Background . . . 23
- 6.2 Entropy . . . 23
- 6.3 Algorithm . . . 23
- 6.4 Examples and Implementation . . . 25

7 Unity Implementation **27**
- 7.1 Initialization . . . 27
- 7.2 Manager Scripts . . . 28
- 7.3 Location System . . . 29
  - 7.3.1 Purpose . . . 29
  - 7.3.2 Example . . . 30

8 Conclusion **31**
8.1 Results 31
8.2 Thoughts and Challenges 31
8.3 Further Work 32
8.3.1 3D Terrain Integration 32
8.3.2 Integration of Additional Game Systems 32
8.3.3 Optimisation 33

9 References **34**

1 Introduction

In the realm of digital environments, the creation of realistic and captivating terrains is a fundamental aspect of many applications, ranging from video games to simulation software and virtual reality experiences. Traditionally, the construction of terrain models relied heavily on manual design or predefined algorithms, limiting the scope of generated landscapes and often requiring extensive human intervention.

However, in recent years, procedural content generation[16] has emerged as a powerful alternative. By employing mathematical algorithms and procedural techniques, this approach allows for the automatic generation of vast and diverse terrains, offering unparalleled scalability and flexibility. Moreover, procedural terrain generation enables the creation of dynamic landscapes that can adapt to various parameters, offering a more immersive and interactive experience for users.

In this project, we delve into the fascinating world of procedural terrain generation from a mathematical standpoint. Our objective is to explore the underlying principles and techniques involved in generating realistic terrains algorithmically. By leveraging mathematical concepts such as noise functions, and model synthesis, we aim to develop a deep understanding of how terrain features can be generated procedurally and how mathematical principles govern their appearance and behavior.

Through this project, we intend to accomplish the following objectives:

1. **Understanding Terrain Generation Algorithms:** We will study various algorithms commonly used in procedural terrain generation, including Perlin noise, wave function collapse, marching cubes,and various methods of interpolation.

2. **Implementation and Experimentation:** We will implement these algorithms and experiment with different parameters to observe their effects on the generated terrains. Through coding and simulation, we will gain insights into the interplay between mathematical parameters and terrain characteristics.

3. **Analysis and Evaluation:** We will critically analyze the generated terrains, evaluating their realism, visual appeal, and suitability for specific applications.

4. **Application and Future Directions:** Finally, we will discuss potential applications of procedural terrain generation across various domains, including gaming, virtual reality, and scientific simulations. Furthermore, we will explore avenues for future research and development in this exciting field.

Another one of the main motivations for undertaking this project is to build a game world to support a system being built by a graduate student. Their project involves the generation of procedural narrative puzzles. Ultimately this project should be capable of supporting their work and incorporating the systems that they have developed.

2 Perlin Noise

Noise in the context of computer graphics and procedural content generation is effectively a random number generator created with the purpose of producing much more harmonic and smooth outputs than a standard pseudo-random number generator. There are multiple types of these functions categorised by how they are created. Value noise[12] consists of a square lattice of integer points with a value generated between -1 and 1 at each point. These numbers are then interpolated to produce an image such as this, with darker regions representing negative values and lighter regions representing positive values.

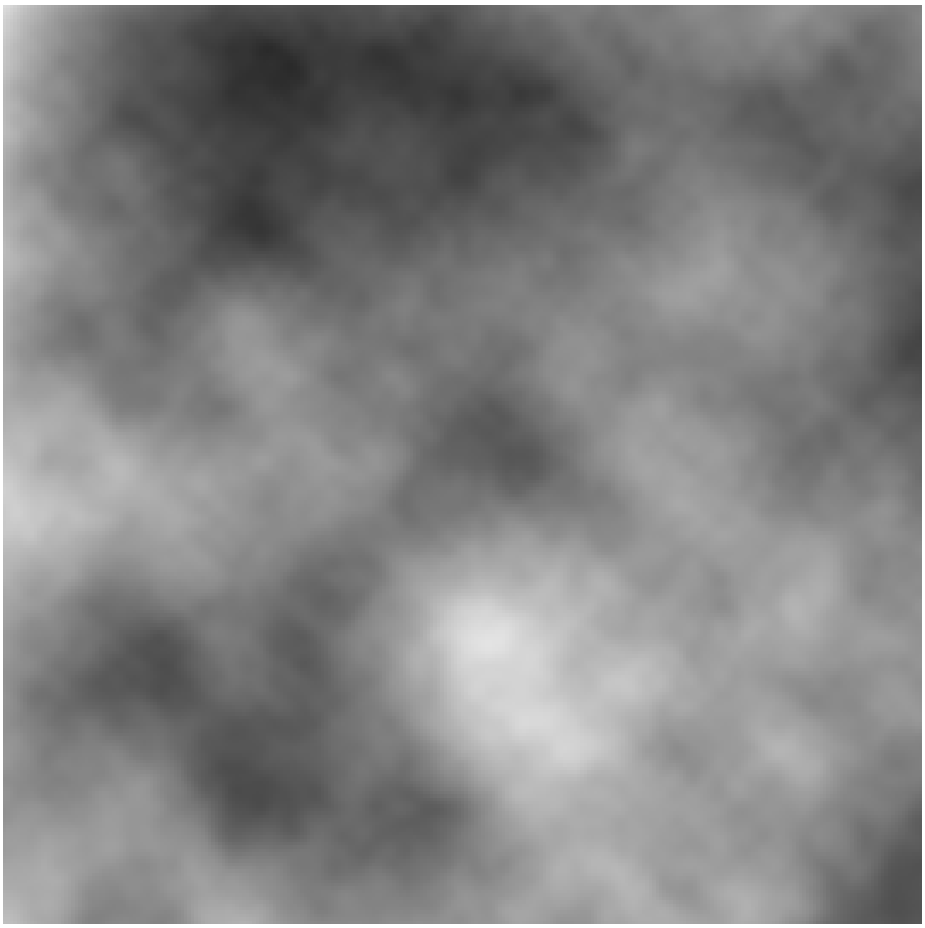


Figure 1: Example of modified value noise by Mat Muze[11]

Gradient noise is another method of generating noise functions, it involves a similar square lattice of integer points but rather than assigning a scalar value, it assigns a gradient. Dot products of these are then interpolated to produce the resulting noise[6].

2.1 Background

The first and most notable example of gradient noise was created by Ken Perlin in 1983. He was inspired to create it by his dissatisfaction with existing methods of procedural texture generation at the time[5]. Originally, it was used to create more natural looking surfaces for CGI, since then it has been used by visual effects artists all over the world to give greater texture and realism to their creations all while retaining computational efficiency.

Perlin noise can be utilized in 1, 2 or 3 Dimensions depending on the requirements of the user. Ken Perlin has since gone on to create Simplex Noise which was created to address some of the problems with his original Perlin noise function, such as removing directional artifacts and allowing it to scale infinitely to higher dimensions.

For the purposes of this project, I utilized both 2D and 3D Perlin noise. I will first go over the 2D implementation.

2D Perlin noise takes a 2-Dimensional Vector and produces a scalar value. The value returned will either be between -1 and 1 or 0 and 1 depending on the implementation. As Perlin noise is generated by interpolating values, this produces smooth curves as we iterate our input over 2-Dimensional surfaces. By simply generating a flat mesh(3) with vertices at integer locations in a square lattice, then adjusting the positions of these vertices in a third dimension using Perlin noise, we end up with something that could already be called "Procedurally Generated Terrain".

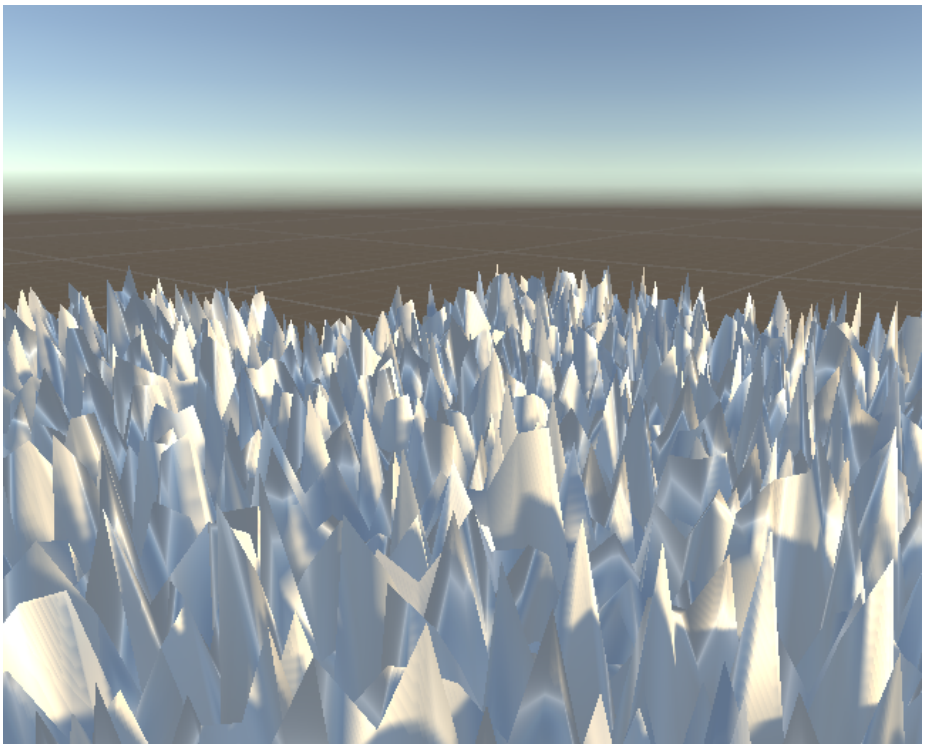


Figure 2: Terrain Generated Using Perlin Noise in Unity

As this image clearly shows, Perlin Noise applied by itself is practically useless. This

terrain is chaotic and steep with very little in the way of what you might consider ground. The solution to this problem entails the use of fractal noise.

Producing fractal noise involves taking multiple samples of noise, in this case Perlin Noise, and adding it back together[1][17].

2.2 Parameters

Creating fractal noise requires additional parameters to be defined. Adjusting these parameters is what allows us to generate unique looking outcomes for our generated terrain. For the purposes of this section, let us call the Perlin noise function:

$$P(X, Y)$$

and our final noise value at a given point:

$$N(X, Y)$$

2.2.1 Scale

The first parameter we define isn't necessarily related to our fractal noise but rather it is a useful property to adjust for any noise application. As we saw with the first iteration of terrain, it's full of sharp spikes and valleys and is much too chaotic. It would be nice to be able to zoom in and out of the noise to spread these sharp protrusions out into less steep surfaces. The scale parameter is effectively a number we divide the input position vector of the Perlin noise function by. This gives us

$$N(X, Y) = P(X/scale, Y/scale)$$

as our noise output. Turning the original messy output into something that is at least slightly more visually appealing.

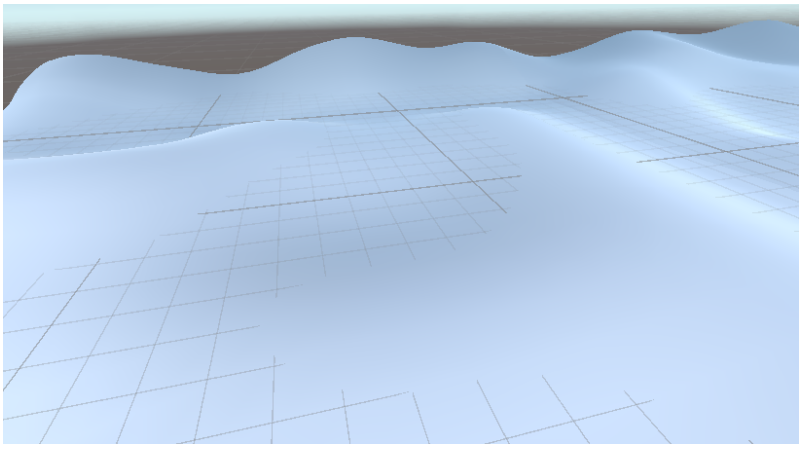


Figure 3: Terrain Generated With a Scale of 10

We then create a second scale parameter to scale our final noise value. Since perlin noise only generates values between 1 and -1, this second scaling parameter is needed if we want to create a mountain biome for example with much larger height variance. I'll call this parameter *mult*.

2.2.2 Lacunarity

As was said previously, fractal terrain is created by sampling the noise function multiple times and summing the results together. To prevent this from simply being our noise result taken to a power, each consecutive term of our sum is again scaled by our lacunarity property. Lacunarity is a Latin word meaning "gap". The gap, in this case, is between successive frequencies in the fractal construction.

To this end we adjust our input into the perlin noise function once again

$$X_n = \frac{X}{scale * lacunarity^n}$$

$$Y_n = \frac{Y}{scale * lacunarity^n}$$

n in this case is the term of the sum of Perlin noise values

2.2.3 Octaves and Persistence

We call each term of our noise sum an Octave. This combined with lacunarity effectively adjusts the "bumpiness of the generated terrain

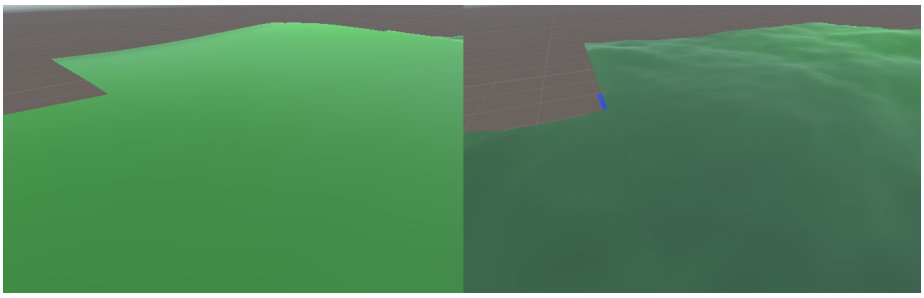


Figure 4: Terrain With 2 Octaves versus 10 Octaves

The final parameter we define is called persistence. This is a scale factor on each consecutive Octave in the sum such that each contributes less to the overall value. This has the effect of smoothing out the noise we get, rather than simply adding a collection of noise samples together, it is more akin to blending nearby values.

2.2.4 Final Noise Function

With all of these parameters now defined, we can construct the final noise function used in the terrain generation algorithm:

$$X_n = \frac{x}{scale * lacunarity^n}$$

$$Y_n = \frac{y}{scale * lacunarity^n}$$

$$N(x,y) = mult * \sum_{n=0}^{N-1} P(X_n, Y_n) * persistance^n$$

$$N = Number\ of\ Octaves$$

```
2 references
float GetBiomeNoise(float x, float z, Biome tempBiome)
{
    float amplitude = 1;
    float frequency = 1;
    float noiseHeight = 0;
    for (int i = 0; i < tempBiome.octaves; i++)
    {
        float sampleX = x / tempBiome.scale * frequency;
        float sampleZ = z / tempBiome.scale * frequency;
        float perlinValue = Mathf.PerlinNoise(sampleX + seed.x, sampleZ + seed.z) * 2 - 1;
        noiseHeight += perlinValue * amplitude;

        amplitude *= tempBiome.persistance;
        frequency *= tempBiome.lacunarity;
    }
    return noiseHeight * tempBiome.mult + tempBiome.heightOffset;
}
```

Figure 5: Code Implementation

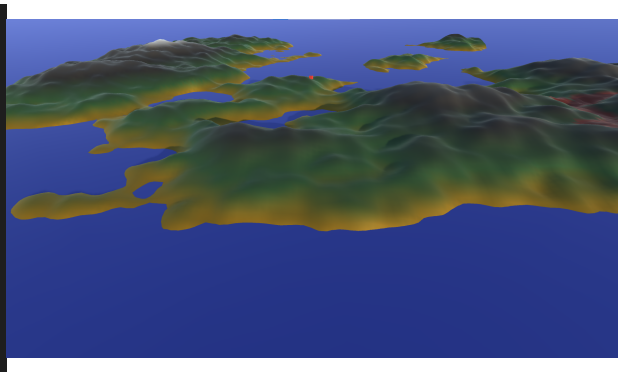


Figure 6: Terrain Using Noise Function

2.3 Biomes

A Biome in the context of video game terrain generation is generally a set of objects, be they foliage or animals alongside a colour palette to give the appearance of a certain type of area. E.g. forests full of trees or tall rocky mountains. Using the modified function, the parameters can be manipulated to produce a large variety of terrain textures. For example, by raising the scale and lowering the octaves, we achieve smooth rolling hills. By keeping the mult parameter small and increasing the octaves, we get flat bumpy ground perfect for adding trees on top of to create a forest biome.

Mixing and matching these creates uniquely textured terrain but a lot more work is needed to give the appearance of something believable. This will be gone over later in the Unity Implementation Section(7).

There are 6 unique biomes in this project. First there is the plains biome, which implements a small rolling hills effect, as mentioned before. Alongside this there is a list of

Figure 7: Parameters For The Plains Biome

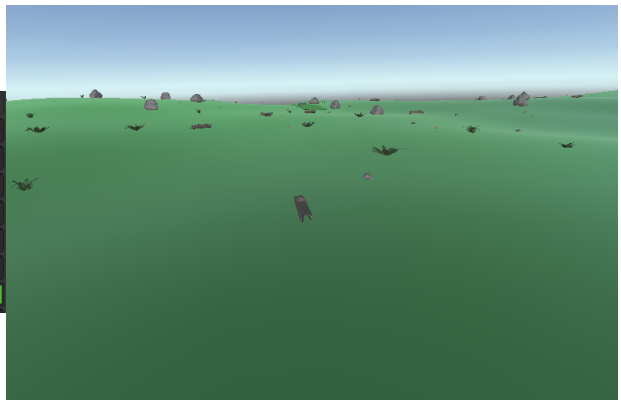


Figure 8: Generated Terrain

objects that can be placed in each biome. For the plains this consists of differently shaped rocks, a few kinds of grass, some scattered pieces of wood, and a few varieties of flowers.

The second biome would be the forest. As mentioned previously, this is a relatively flat bumpy ground with a variety of trees, rocks and fallen wood that can be generated on the ground. While this biome is one of the more visually appealing biomes in the project, the actual terrain mesh itself is very simple. It is almost entirely flat. The appeal instead comes from the generated objects populating the area.

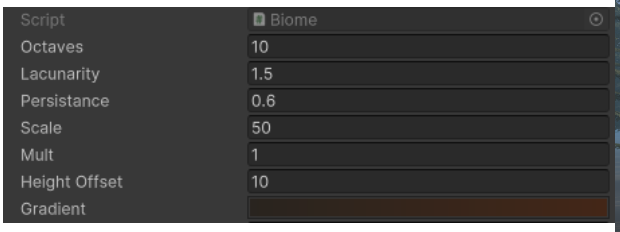


Figure 9: Parameters For The Forest Biome

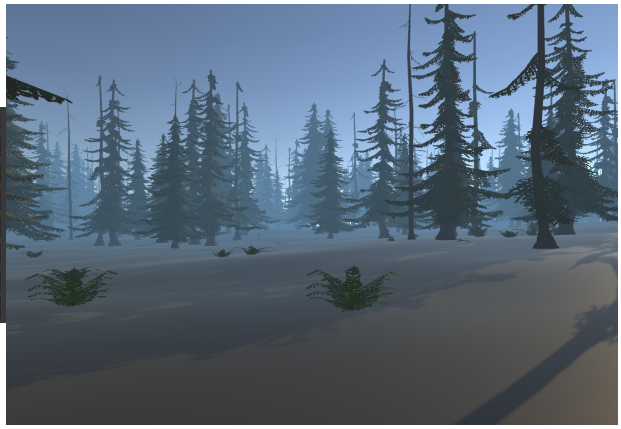


Figure 10: Generated Terrain

Third there is the mountain biome. This is created by using a very large mult parameter to create a much larger discrepancy between the low points and the high points. This is all raised up by a parameter I haven't mentioned yet which I've called the height parameter. This is just a value added to the noise to either raise or lower all of the terrain in the biome. In the case of the mountain biome, this keeps the lower values of the terrain above sea level and makes the peaks even higher. The objects that populate the mountain biome mostly consists of different shaped rocks and are relatively sparse.

Figure 11: Parameters For The Mountain Biome

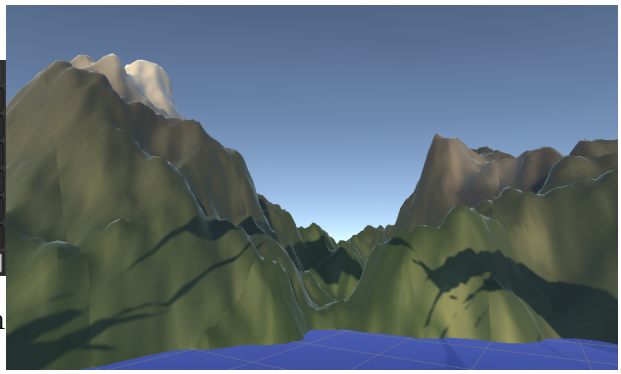


Figure 12: Generated Terrain

The fourth biome implemented is the Desert biome. It is a very hilly area with tall sand dunes, similar in appearance to the plains biome but with a larger mult value to allow for more variance in the heights of the dunes.

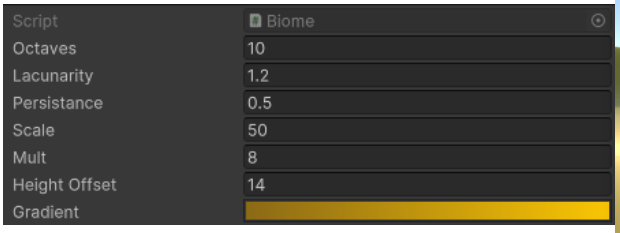


Figure 13: Parameters For The Desert Biome

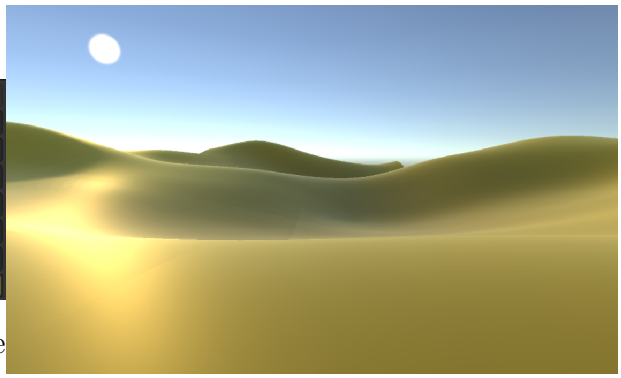


Figure 14: Generated Terrain

The last of the biomes with unique terrain is what I've called the island biome. This is somewhat similar to the mountains but without the height offset, transforming the high peaks into islands peaking above sea level. It is also more spread out to avoid the sharp mountain peaks and instead have larger, flatter islands to walk around on. By inserting this into the world and surrounding it with the other biomes, it gives the impression of a coastline or a lake that tapers off into our final biome.

Figure 15: Parameters For The Islands Biome

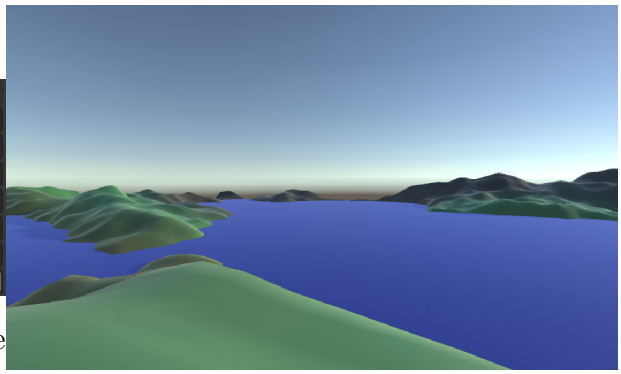


Figure 16: Generated Terrain

Lastly there is the ocean biome. The exact parameters for the noise function aren't too important here. The purpose of this biome is simply to have all the terrain be below sea level. Therefore it is simply created by using a relatively flat noise function with a large negative height offset. The ocean is used in this project to create some kind of boundary for the game space. The project is capable of generating new terrain almost indefinitely, at least as far as unity allows for that, but for my needs having a limited space surrounded by ocean was more than acceptable.

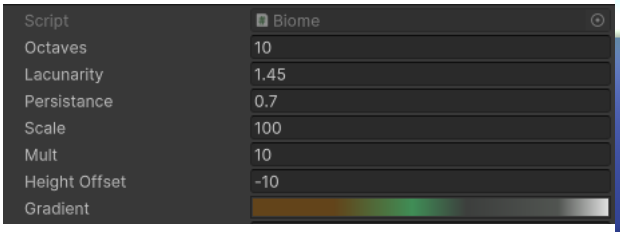


Figure 17: Parameters For The Ocean Biome

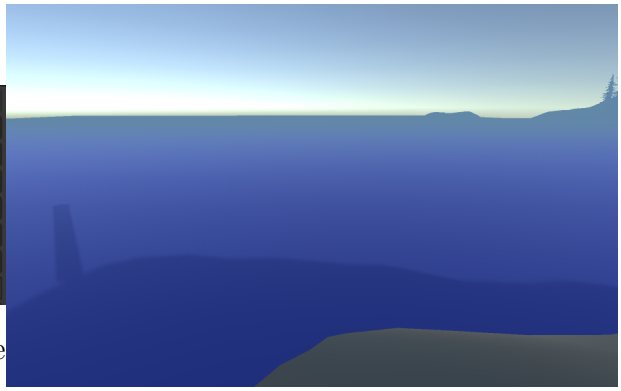


Figure 18: Generated Terrain

These biomes are then placed in the world at a single point in the 2-Dimensional flat plane using Wave Function Collapse(6).

2.4 How It's Evaluated and Results

Now that these biomes have been defined and placed into the world, all that remains is to load our game scene. We generate our mesh(3) vertex by vertex, at each we call our noise map class which outputs the correct noise value at that location. This takes into account all nearby biomes and if necessary interpolates(5) between them. The result is a versatile noise function which can generate many different styles of terrain and smoothly interpolate between them. Adding new biomes is as simple as defining new sets of parameters and adding them to the biome placement algorithm(6).

3 Mesh

A Mesh is a collection of vertices, edges, and faces that act as the foundation of a model in a video game[13]. To have all of our terrain appear in the game world we need to render it using a mesh. The way a mesh is created is by outlining a collection of vertices, edges are then defined between these vertices to make up triangles which build up the structure of the intended object. To render a single square we start with 4 vertices:

$$I = (0, 0, 0),\ J = (0, 1, 0)$$

$$K = (1, 0, 0),\ L = (1, 1, 0)$$

From these vertices we create our triangles. Triangles are rendered with a single face which means that they are only visible from one side. The deciding factor in which side they are visible from depends on the order in which the vertices are listed. Namely, for a triangle to be visible its vertices must be rendered in a clockwise rotation relative to the camera.

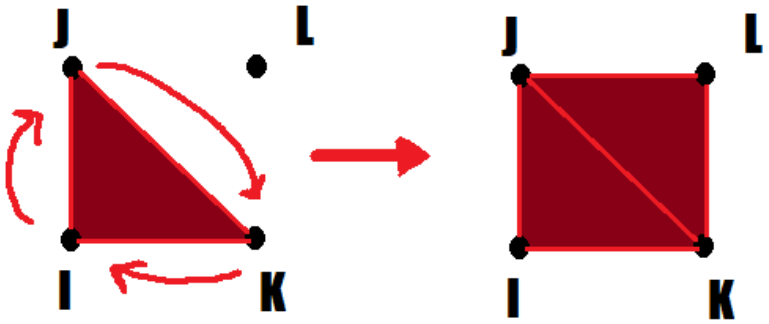


Figure 19: Triangle Rendered From 3 Vertices.

This means that for our square we must be careful to ensure that the two triangles which make up the shape are facing the same direction. We choose the permutation IJK for our first triangle and JLK for the second which produces the desired result.

3.1 2D

The terrain in our game is made up in much the same way. Rather than a single square, it is a large grid of squares. Each vertex is at a point in $\mathbb{Z}$x$\mathbb{Z}$ with it's position in the third dimension being decided by our noise function(2). Our position vectors are all of the form

$$V = (x \in \mathbb{Z}, N(x, z), z \in \mathbb{Z})$$

Once we've created our array of vertices, our next task is to draw the triangles in-between.

This involves adding each vertex into an array representing the triangles. The renderer will then loop through this triangles array to draw our mesh, taking each group of 3 vertices as a triangle. The code implementation looks like this

```
vertices = new Vector3[(xSize + 1) * (zSize + 1)];
Random.InitState(MapManager.seed.value + chunkSeed);
for (int i = 0, z = 0; z <= zSize; z++)
{
    for (int x = 0; x <= xSize; x++)
    {
        vertices[i] = new Vector3(x, MapManager.noiseMap.GetNoise(x + initX, z + initZ), z);
        GameObject plant = MapManager.noiseMap.GetPlant(vertices[i] + new Vector3(initX,0,initZ));
        if (plant != null && vertices[i].y > 0)
        {
            Vector3 position = vertices[i] + new Vector3(initX, 0, initZ);
            GameObject obj = Instantiate(plant, position, Quaternion.Euler(new Vector3(0,Random.Range(0f,360f),0)));
            obj.transform.parent = transform;
            foliage.Add(new Vector2Int((int)position.x, (int)position.z), obj);
        }
        i++;
    }
}

int vert = 0;
int tris = 0;
triangles = new int[6 * xSize * zSize];

for (int z = 0; z < zSize; z++)
{
    for (int x = 0; x < xSize; x++)
    {
        triangles[0 + tris] = vert + 0;
        triangles[1 + tris] = vert + xSize + 1;
        triangles[2 + tris] = vert + 1;
        triangles[3 + tris] = vert + 1;
        triangles[4 + tris] = vert + xSize + 1;
        triangles[5 + tris] = vert + xSize + 2;

        vert++;
        tris += 6;
    }
    vert++;
}
```

Figure 20: Code Implementation

The world in this project is rendered in chunks(4) which means that each chunk has its own array of vertices and triangles. Specifically each chunk is made up of $30 * 30$ squares. Therefore each chunk must contain 961 vertices. Which means the length of the triangle array per chunk is $6 * 30 * 30 = 5400$ as there ar 900 squares that require 6 inputs each into the triangles array.

Vertices at the edges of each chunk must be rendered twice to ensure that the terrain links together correctly. Since our noise function is a completely separate entity from the actual mesh generation, this works fine, as these vertices which are rendered twice will always have the same position because the height is purely decided by the x and z values.

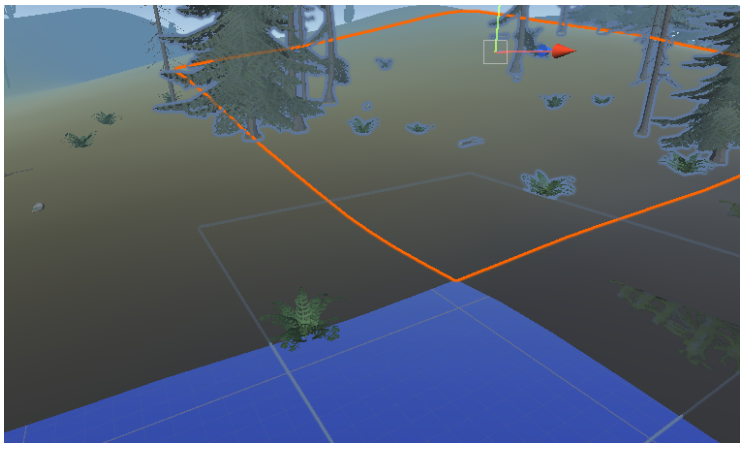


Figure 21: Intersection of Chunks

This terrain is relatively simple to generate as we are effectively just generating a flat plane.

The lack of overhangs or caves simplifies the mesh immensely. We never have to worry about which triangles do or do not need to be rendered as it is always the same for every chunk.

3.2 3D

3D meshes require a lot more effort. Modelling software like blender allows you to start with primitive shapes like cubes or spheres and distort them into the shape that you are looking to create. Distorting is a very appealing operation to perform on meshes as it does not change the order in which the vertices are processed. For example if you had a cube and you stretched it into a longer cuboid, the vertices would change position but as far as the triangles are concerned you are still using the same indices in the vertex array. A nice way of doing 3D meshes is to have a set of pieces that can be slotted together and then distorted to make the shape we are looking for.

One option is to divide our model by a grid of cubes with each cube containing a primitive shape which constructs the mesh. For example, each face of a cube could be used for these primitives producing a blocky approximation of our shape.

A more sophisticated version of this idea is the marching cubes algorithm[14].

3.3 Marching Cubes

Marching Cubes is an algorithm created in the 80's as a means of turning a surface into a polygonal mesh. Effectively it divides a volume into cubes and performs a calculation at each vertex of these cubes. This calculation generally provides a number or a boolean condition. A threshold is decided; if the number calculated at a vertex is above the threshold it is considered outside the mesh, otherwise it is considered inside the mesh.

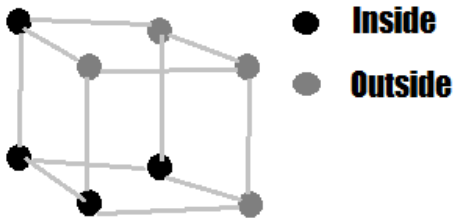


Figure 22: Included and Excluded Vertices

Each cube has 8 vertices, and each vertex can either be inside or outside the mesh. This gives 2^8 possible cube configurations, each one requiring a unique slice of mesh to be rendered. If we ignore mesh configurations which are simply rotations or reflections of other meshes, then we actually only have 15 different shapes to consider. These shapes are

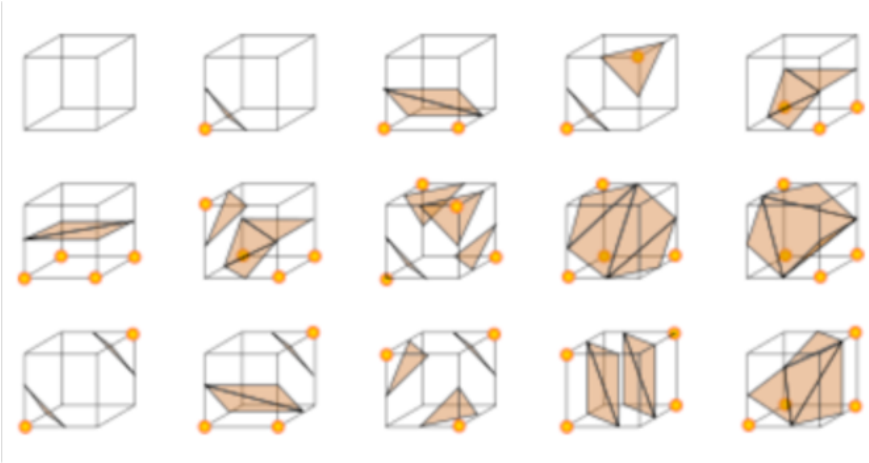


Figure 23: The 15 Mesh Types of Marching Cubes[15]

For this project, the calculations at each vertex are a 3D Perlin noise call. As discussed in the Noise(2) chapter, this returns a value between -1 and 1. If we choose our threshold value to be 0 then what marching cubes will construct is an approximation of the surface

$$N(x, y, z) = 0$$

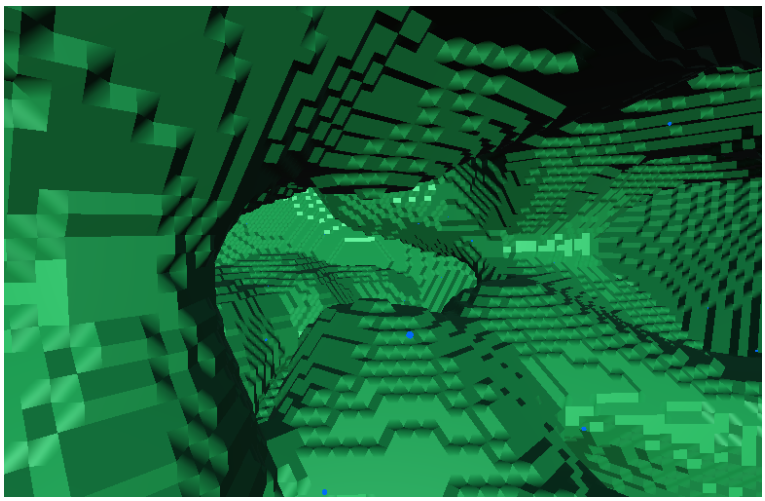


Figure 24: Approximation of 3D Perlin Noise Using Marching Cubes

As we can see, this is a very blocky approximation. One approach to fixing this would be to split our mesh into increasingly smaller and smaller cubes. While this approach would work, it requires a lot more computation. Another option is to linearly interpolate along the edges of the cubes to decide how far along that edge our triangle vertices should be. As we can see from 23 The vertices for the mesh are not the same as the vertices for

the cubes. In fact they are always halfway along a given edge. By sliding these vertices along that edge based on the noise values of the cubes vertices, we can get a much better approximation of the surface we are recreating. For example take this cube here:

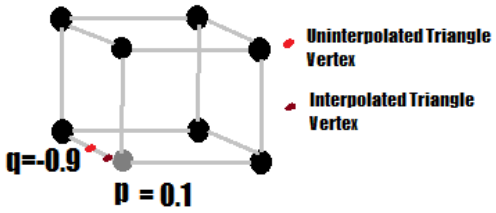


Figure 25: Interpolation of Mesh Vertex

It has a noise value of $p = 0.1$ at one vertex and $q = -0.9$ at the other. From this it is obvious that the point where $N(x, y, z) = 0$ is much closer to the first vertex than the second. The method of interpolation I came up with is

$$I(p, q) = \frac{|p|}{|p| + |q|}$$

Using this interpolation here is the result As we can see, this a much smoother and

```
1 reference
public Cube(Vector3Int position, float[] vertexVals, float d)
{
    this.position = position;
    this.d = d;
    triIndex = 0;
    if (vertexVals[0] < d) triIndex += 1;
    if (vertexVals[1] < d) triIndex += 2;
    if (vertexVals[2] < d) triIndex += 4;
    if (vertexVals[3] < d) triIndex += 8;
    if (vertexVals[4] < d) triIndex += 16;
    if (vertexVals[5] < d) triIndex += 32;
    if (vertexVals[6] < d) triIndex += 64;
    if (vertexVals[7] < d) triIndex += 128;


    verts = new Vector3[]
    {
        position + new Vector3(InterpVal(vertexVals[0],vertexVals[1]), 0,1),
        position + new Vector3(1,0,InterpVal(vertexVals[2],vertexVals[1])),
        position + new Vector3(InterpVal(vertexVals[3],vertexVals[2]),0,0),
        position + new Vector3(0,0,InterpVal(vertexVals[3],vertexVals[0])),
        position + new Vector3(InterpVal(vertexVals[4],vertexVals[5]), 1,1),
        position + new Vector3(1,1,InterpVal(vertexVals[6],vertexVals[5])),
        position + new Vector3(InterpVal(vertexVals[7],vertexVals[6]),1,0),
        position + new Vector3(0,1, InterpVal(vertexVals[7],vertexVals[4])),
        position + new Vector3(0,InterpVal(vertexVals[0],vertexVals[4]), 1),
        position + new Vector3(1,InterpVal(vertexVals[1],vertexVals[5]), 1),
        position + new Vector3(1,InterpVal(vertexVals[2],vertexVals[6]), 0),
        position + new Vector3(0,InterpVal(vertexVals[3],vertexVals[7]), 0),
    };
}
```

Figure 26: Code Implementation

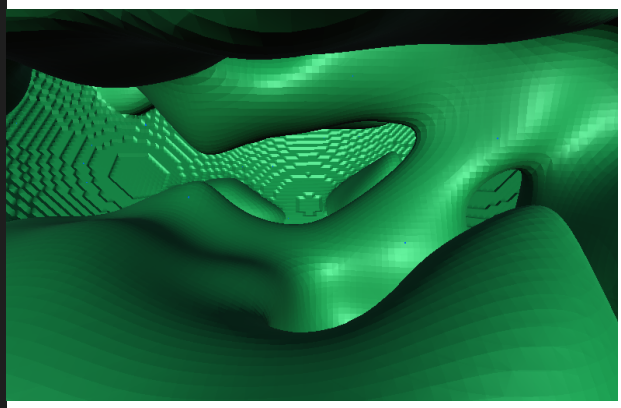


Figure 27: Smooth 3D Terrain

more visually appealing result. For the purpose of this project, marching cubes are being used to render procedural cave systems using 3D Perlin Noise with the ultimate goal of combining this system with the 2D Noise generation to create underground explorable cave systems for the player.

4 Seeding

4.1 Purpose

It would be beneficial to be able to replicate certain results of the procedural generation, either for testing purposes or to recreate a good result. As it is right now, all randomness is based on the in-built random function within unity meaning that we have no control over the results. The solution to this is to seed the random function every time we employ it. This means that we give it an integer as an input, which will cause it generate a unique sequence of numbers completely dependent on that integer.

This means that if we set the seed to 0 the terrain will be the exact same every time we run the application.

4.2 Implementation

Specifically we actually generate two seeds, effectively giving us a seed vector rather than an integer. This is because the noise function takes a 2-dimensional vector as an input. If we just used a single integer as the seed and added it to the inputs this would effectively just slide our terrain along the line $x = z$.

For seeding the standard random function we can simply take the sum of the entries of the seed vector

```
3 references
public Seed()
{
    x = Random.Range(-10000f, 10000f);
    z = Random.Range(-10000f, 10000f);
    value = (int)(x + z);
}
```

Figure 28: Code Implementation

The seed class generates two numbers between -10000 and 10000. Since our noise function takes floating point numbers as inputs, it makes sense to use random floats for the vector entries as this gives us more possible seed values, the integer seed is then simply their sum casted to an int.

Now, every time a random number is required, we simply provide global access to the seed and use it as the input. The seed has two constructors which allows us to set it manually if required

```
int zSize = MapManager.zSize;

vertices = new Vector3[(xSize + 1) * (zSize + 1)];
Random.InitState(MapManager.seed.value + chunkSeed);
for (int i = 0, z = 0; z <= zSize; z++)
{
    for (int x = 0; x <= xSize; x++)
    {
        vertices[i] = new Vector3(x, MapManager.noiseMa
```

Figure 29: Seeded Random Number

```
float frequency = 1;
float noiseHeight = 0;
for (int i = 0; i < tempBiome.octaves; i++)
{
    float sampleX = x / tempBiome.scale * frequency;
    float sampleZ = z / tempBiome.scale * frequency;
    float perlinValue = Mathf.PerlinNoise(sampleX + seed.x, sampleZ + seed.z) * 2 - 1;
    noiseHeight += perlinValue * amplitude;

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

Figure 30: Seeded Perlin Noise

4.3 Chunks

For performance reasons, the whole game world cannot be generated all at once. This means that we have to spread out the load by generating the world in smaller sections first. We call these sections chunks. Choosing the correct size chunk is a very important problem to solve. If the chunks are too big then each time a new one is loaded the program will slow down, leading to a very choppy experience which is unpleasant for the user. On the other hand, having chunks that are too small also leads to inefficencies. At the borders between chunks, the vertices of the mesh must be rendered twice to allow for the chunks to link together properly. Having very small chunks generates a lot of extra vertices to be calculated, although the triangle count of the mesh will stay the same. The number I settled on that had a good balance of both reducing choppiness while also minimizing the amount of chunks being rendered was a size of 30 units in unity. This means that the total number of vertices in a chunk is $31 * 31 = 961$ The way these chunks are created and

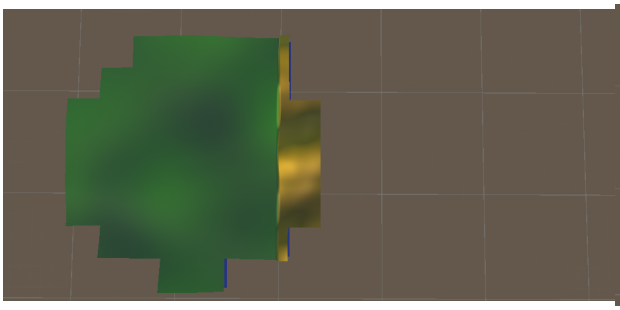


Figure 31: Chunks Before Movement

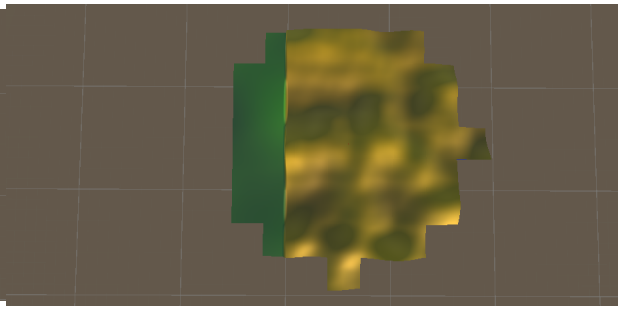


Figure 32: Chunks After

destroyed is decided by a separate radius. The player effectively has a ring around them, if the centre of a chunk is within that ring then it will be loaded in, otherwise it is destroyed. Since these chunks are laid out in a square grid, they are almost always loaded in one at a time. Originally the player had a square around them for chunk loading. This meant that multiple parallel chunks would be loaded at a time, leading to poor performance.

```
IEnumerator GenerateChunk(Vector2Int pos)
{
    if(!chunks.TryGetValue(pos, out Chunk test))
    {
        Chunk newChunk = Instantiate(chunkObj, new Vector3(pos.x * size, 0, pos.y * size), Quaternion.identity).GetComponent<Chunk>();
        newChunk.initX = pos.x * size;
        newChunk.initZ = pos.y * size;
        newChunk.index = pos;
        chunks.Add(pos, newChunk);
    }
    yield return new WaitForEndOfFrame();
    RenderAllLocations();
}
```

Figure 33: Code Implementation

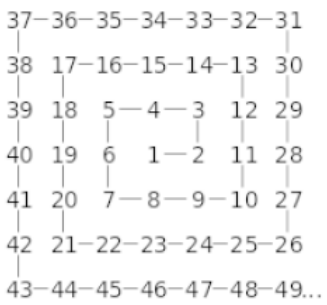
4.4 Unique Chunk Seeds

Since chunks are constantly being loaded and unloaded, there needs to be a way to make sure that they are loaded in the same way each time. The terrain itself is determined by it's position, meaning that chunks loaded in the same position will always have the same terrain height values. The problem comes from the objects[19][20] that are loaded onto the terrain as these are set randomly each time the chunk is loaded. The obvious solution to this is to just seed the random number generator the same way each time that chunk is loaded. This solution does indeed solve the problem but it leads to another one: chunks using the same object spawning function AKA chunks in the same biome will spawn objects in the exact same relative positions if we seed the random number generator with our seed.

What we need is a unique seed for each individual chunk to prevent this problem from occurring. Effectively, what we require is a computationally viable mapping

$$\mathbb{Z}x\mathbb{Z} \mapsto \mathbb{Z}$$

One aesthetically pleasing representation of this is a spiral mapping



Unfortunately there is no nice formula to easily get from $\mathbb{Z}x\mathbb{Z}$ to $\mathbb{Z}$ using this approach[22]. The approach we use in the project is to first construct this mapping of $\mathbb{N}x\mathbb{N}$ to $\mathbb{N}$[21]

$$f(x,z) = \begin{cases} x + z^2 & x \geq z \\ x^2 + x + z & otherwise \end{cases}$$

This is suitable for the upper right quadrant. If we then multiply our result by 2, we can use the same mapping for the upper left quadrant by simply multiplying those results by 2 and subtracting 1. For the lower halfplane we simply take the negative of the above results. This gives us a nice function for our unique chunk seeds.

$$f(x,z) = \begin{cases} 2(|x| + |z|^2) & x \geq z, \text{ and } x, z > 0 \\ 2(|x|^2 + |x| + |z|) & x < z, \text{ and } x, z > 0 \\ 2(|x| + |z|^2) - 1 & |x| \geq z, \text{ and } z > 0 > x \\ 2(|x|^2 + |x| + |z|) - 1 & |x| < z, \text{ and } z > 0 > x \\ -2(|x| + |z|^2) & |x| \geq |z|, \text{ and } x, z < 0 \\ -2(|x|^2 + |x| + |z|) & |x| < |z|, \text{ and } x, z < 0 \\ -2(|x| + |z|^2) + 1 & x \geq |z|, \text{ and } z < 0 < x \\ -2(|x|^2 + |x| + |z|) + 1 & x < |z|, \text{ and } z < 0 < x \\ 0 & x = z = 0 \end{cases}$$

We can add this number to our original seed at each chunk to get unique chunk seeds every time we run the application. This gives us fully procedural objects with location permanence as they are created and destroyed

```
Random.InitState(MapManager.seed.value + chunkSeed);
for (int i = 0, z = 0; z <= zSize; z++)
{
    for (int x = 0; x <= xSize; x++)
    {
        vertices[i] = new Vector3(x, MapManager.noiseMap.GetNoise(x + initX, z + initZ), z);
        GameObject plant = MapManager.noiseMap.GetPlant(vertices[i] + new Vector3(initX,0,initZ));
        if (plant != null && vertices[i].y > 0)
        {
            Vector3 position = vertices[i] + new Vector3(initX, 0, initZ);
            GameObject obj = Instantiate(plant, position, Quaternion.Euler(new Vector3(0,Random.Range(0f,360f),0)));
            obj.transform.parent = transform;
            foliage.Add(new Vector2Int((int)position.x, (int)position.z), obj);
        }
        i++;
    }
}
```

Figure 34: Code Implementation

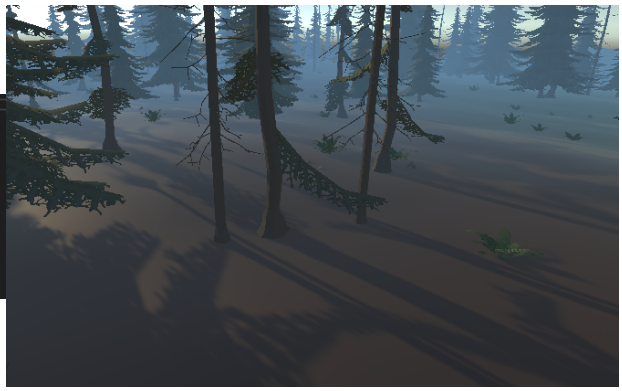


Figure 35: Objects Rendered in Scene

5 Interpolation

One of the main problems that needed to be solved was the problem of interpolation. Each vertex of our terrain mesh has 4 important properties: Height, colour, object, and position. The first 3 being determined by the last. Without interpolation the vertex will simply use the nearest biome as the basis for its generation. This leads to sharp cut offs at the line of points equidistant from 2 biomes.

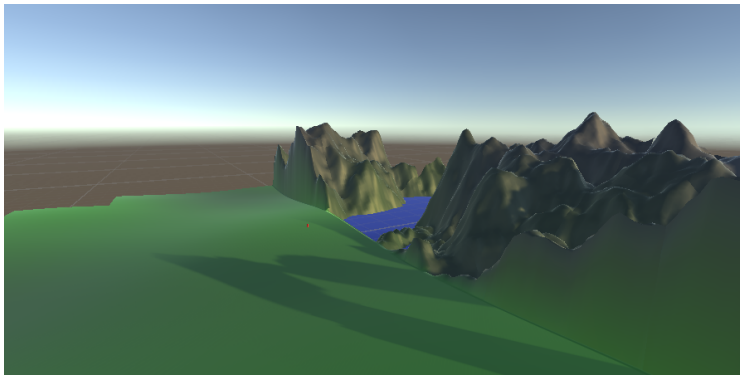


Figure 36: Hard Cutoff Between 2 Biomes

This is visually off-putting. We need some way to seamlessly blend these biomes together as the vertices approach one biome from the other. For 2 biomes this is straightforward: simply set a radius in which we want interpolation to begin, then when the distances between the biomes and a vertex are both within that distance we can interpolate them linearly along the line parallel to the equidistant divide of the biomes. The real problem

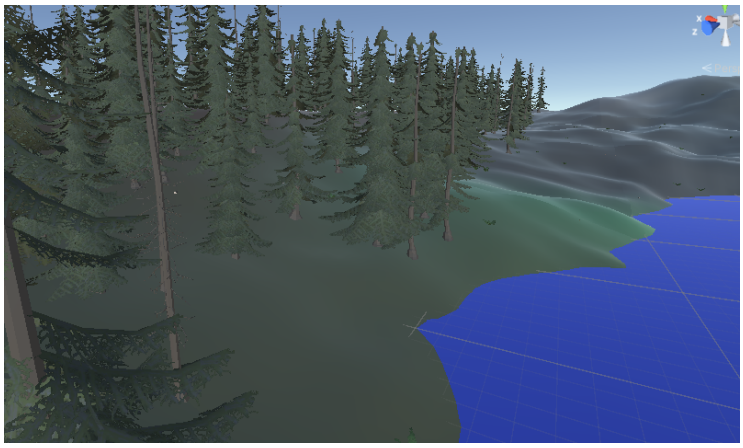


Figure 37: Interpolation Between 2 Biomes

comes from finding a good interpolation algorithm between an arbitrary number of biomes.

The algorithm I came up with starts by searching for the nearest biome and storing the distance between it and the vertex we are generating.

The next step is to get a list of all other relevant biomes. We then subtract the distance to the closest biome from all of the other biome distances. From this list of numbers we remove the biomes which have a higher distance difference than our predetermined radius so that they will not influence the interpolation. From here on let this radius be r.

If the closest biome is the only one remaining then we simply return it as the biome for that vertex. Otherwise we must interpolate between the remaining biomes. For each value d_i in our list of distance differences we calculate a new value.

$$x_i = \frac{r - d_i}{r}$$

and we also calculate the sum of these values

$$X = \sum_{i \in I} x_i$$

The process from here slightly differs depending on the exact property we are looking to calculate.

5.1 Noise Value

To calculate the noise value we will need to calculate what the height would be at the vertex for each biome remaining in our list. Let this value be $B_i(x, z)$. Our final interpolated noise value is then:

$$N(x, z) = \sum_{i \in I} \frac{x_i}{X} * B_i(x, z)$$

Using this as our noise value produces the smooth terrain we are after, with a seamless transistion between many biomes.

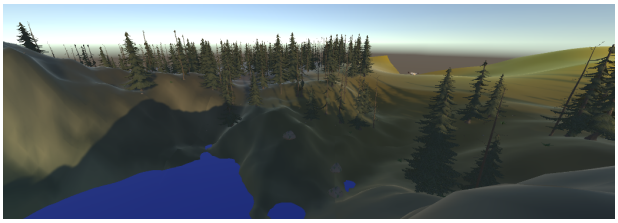


Figure 38: Interpolation of Many Biomes

```
15 references
public float GetNoise(float x, float z)
{
    List<BiomeCell> biomes = GetCurrentBiomes(x, z);
    BiomeCell closestBiome = FindClosestBiome(x, z,biomes);
    float closestDist = GetDist(x, z, closestBiome);

    List<BiomeCell> closeBiomes = new List<BiomeCell>();
    List<float> floats = new List<float>();
    foreach (BiomeCell biome in biomes)
    {
        float distDif = Mathf.Abs(GetDist(x, z, biome) - closestDist);
        if (distDif < interpDist)
        {
            closeBiomes.Add(biome);
            floats.Add(distDif);
        }
    }
    if (closeBiomes.Count <= 1) return GetBiomeNoise(x, z, closestBiome);


    float noiseSum = 0;
    for (int i = 0; i < floats.Count; i++)
    {
        floats[i] = (interpDist - floats[i]) / interpDist;

    }
    float floatsSum = floats.Sum();
    for (int i = 0; i < floats.Count; i++)
    {
        noiseSum += (floats[i]/floatsSum) * GetBiomeNoise(x, z, closeBiomes[i]);
    }

    return noiseSum;
}
```

Figure 39: Code Implementation

5.2 Colour and Foliage

The colour and foliage are interpolated in much the same way. In the case of colour we simply replace the biome height function with a biome colour function which is interpolated in the same way. In unity, colours are represented as a vector of 4 values therefore replacing the scalar height by a vector in this case is acceptable as it is a linear function of vectors.

Figure 40: Interpolation of Colours

```
1 reference
public Color GetColor(Vector3 vert)
{
    List<BiomeCell> biomes = GetCurrentBiomes(vert.x, vert.z);
    BiomeCell closestBiome = FindClosestBiome(vert.x, vert.z, biomes);
    float closestDist = GetDist(vert.x, vert.z, closestBiome);

    List<BiomeCell> closeBiomes = new List<BiomeCell>();
    List<float> floats = new List<float>();
    foreach (BiomeCell biome in biomes)
    {
        float distDif = Mathf.Abs(GetDist(vert.x, vert.z, biome) - closestDist);
        if (distDif < interpDist)
        {
            closeBiomes.Add(biome);
            floats.Add(distDif);
        }
    }
    if (closeBiomes.Count <= 1) return GetBiomeColor(vert.x, vert.z, closestBiome); ;


    Color noiseSum = Color.black;
    for (int i = 0; i < floats.Count; i++)
    {
        floats[i] = (interpDist - floats[i]) / interpDist;

    }
    float floatsSum = floats.Sum();
    for (int i = 0; i < floats.Count; i++)
    {
        noiseSum += (floats[i] / floatsSum) * GetBiomeColor(vert.x, vert.z, closeBiomes[i]);

    }

    return noiseSum;
}
```

Figure 41: Code Implementation

For foliage and spawned objects, rather than interpolating a value, we simply select a random biome from the list using the x_i values as weightings, then use that biome to generate our foliage.

6 Wave Function Collapse

6.1 Background

Wave Function Collapse is a procedural generation technique which borrows its name from quantum mechanics. It involves "collapsing" possibilities for sections of a structure you are trying to generate, similar to how observing a quantum superposition cause it's state to collapse.

A notable example of this algorithm's use is the video game Bad North[7] in which it is utilized to generate small islands.

The algorithm is a fairly recent method of procedural generation, being first outlined in a paper published in 2007 by Paul Merrell[4] before being popularised in 2016 after a Github repository was published showing off its implementation as a method of generating large textures from a small sample texture.

6.2 Entropy

One of the benefits to using this method is the order in which parts are collapsed. At each stage, the "entropy" for the section being generated gets evaluated. This is a measure of uncertainty based on the available options for that section taking into account their relative weighting. Another way to look at entropy is as a measure of the information received from the outcome of a trial. For example, if we were to flip a coin, there are 2 equally likely outcomes. Each outcome is fairly likely, so observing any given outcome is relatively unsurprising. A 6-sided die on the other hand has 6 equally likely outcomes, with, of course, a lower probability of any given outcome. Therefore knowing that the die rolled a 1 for example is a more significant result. This is the same as saying that the die has higher entropy than the coin. The function for calculating entropy is

$$H(X) = -\sum_{x \in X} p(x) log p(x)$$

Here X is the possible outcomes at a given section and $p(x)$ is the probability of one of those outcomes. This idea of entropy was created by Claude Shannon in 1948 and is often called Shannon entropy[3].There are multiple choices for the base of the logarithm but for the purposes of the Wave Function Collapse algorithm we simply need to compare values by relative size, therefore the base is irrelevant.

6.3 Algorithm

The algorithm at it's core is fairly simple. It is more or less a set of constraints placed onto a list of objects. These objects then get placed procedurally in a manner which satisfies the set of constraints. For example, if we take these 5 2D square tiles.

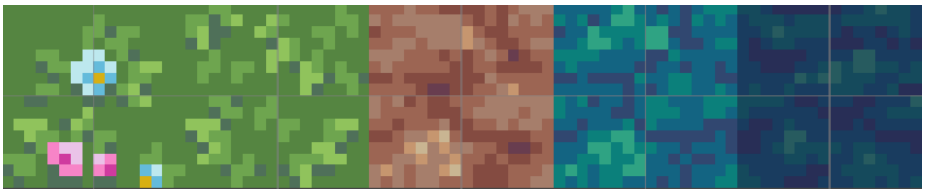


Figure 42: Spring Forest Tiles[18]

For our set of constraints, let us say that a given tile may only be placed next to one of its neighbours in the above image or itself. So the deep water tile can only be next to the shallow water tile or itself, while the grass tile can be next to the flowers,the dirt tile, or itself. Let us also say that each tile has an equal weighting to simplify the entropy calculation.

In this example we start with a 2D grid, each square of which has 5 possible entries. The entropy at each tile is then

$$H(X) = -\sum_{x \in X} 0.2 * log_2 0.2 = 2.322$$

It is enough, however, to simply have the entropy equal to the number of remaining possibilities for a tile in this example. Now that our system is set up, we simply apply the steps of the algorithm until each tile has been collapsed. This starts by taking the tile with the lowest entropy and collapsing it. Collapsing is simply taking a tile and performing a random trial using it's remaining possible entries and their relative weightings as the outcome. At the beginning each tile has equal entropy so we simply choose one at random and choose one of the 5 options. Say we chose a plain grass tile, this then updates our tile grid. In particular, the 4 neighbours directly above, below, and to the sides now only have three possible entries. Then the diagonal neighbours also have new restraints put upon them, in particular it is impossible for the deep water tile to be placed in those locations as it would involve it being next to either a grass tile, a flower tile, or a dirt tile, which is not allowed by our set of constraints. These tiles now have 4 possible entries. This information proliferates out and updates the entire grid. This means that entropy has to be recalculated for each grid member again. One of the issues with this algorithm is that it can fail, meaning that, depending on your constraints and options, you can end up in a situation where a tile has zero possibilities remaining. The point of calculating entropy is to minimize the risk of this happening. Once the entropy values have been recalculated, the next choice of tile to be collapsed is the one with the lowest entropy. This generally ensures that tiles with a lower number of possibilities get collapsed to avoid failure. In this example the tiles with the lowest entropy would be one of the four direct neighbours to the grass tile. So one of these will be collapsed into one of it's three options and the process repeats. Here is the algorithm in action acting on these 5 tiles

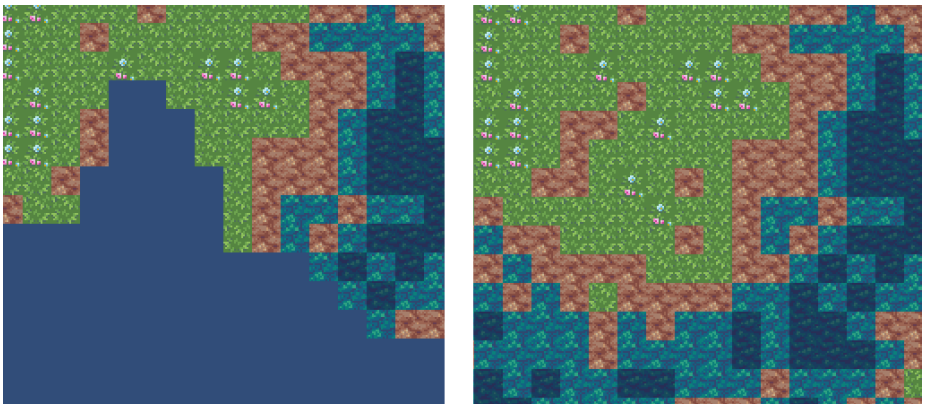


Figure 43: Wave Function Collapse On a Grid of Tiles

6.4 Examples and Implementation

While Wave Function Collapse can produce attractive results, it does come with its fair share of problems. Since entropy needs to be calculated for every tile at every step, it can be relatively slow and is completely unsuitable for extremely large models or grids, such as is often the case with procedural terrain generation. For example, in a Minecraft[9] world which can potentially be 60 million[8] blocks wide, this algorithm would be completely unsuitable. Infinite procedural generation has been done with Wave Function Collapse before. Twitter user @Marian42_ constructed a program for generating an infinite city using this algorithm but it takes a lot of adaptation.

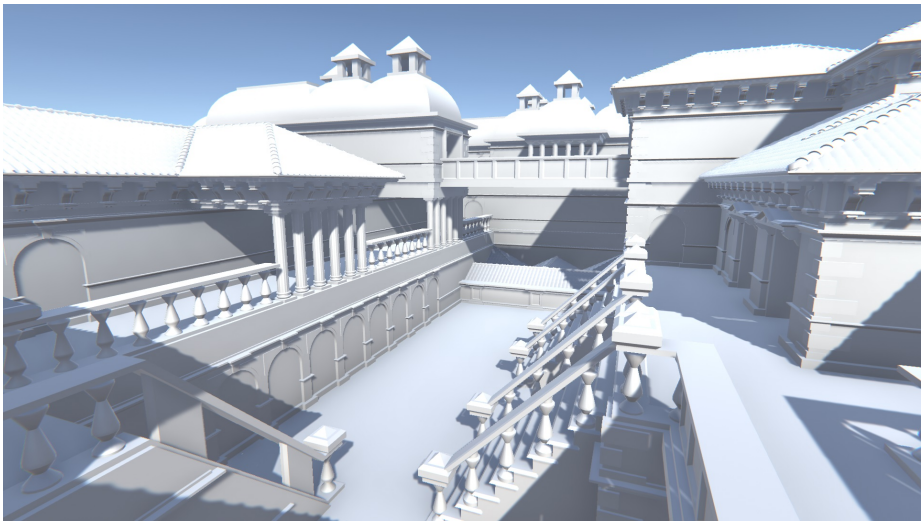


Figure 44: Infinite City Generated Using WFC[10]

The process for generating 3 dimensional structures as opposed to the 2D tile grid is basically the same. Instead of a 2D plane, it's a 3-dimensional grid. For this kind of

generation, the constraints will look considerably different. In the tile grid example, each tile has the same constraints for each neighbour. In 3D however, if you are trying to generate something like buildings then it will require constraints specific to each direction e.g. it wouldn't make sense for a roof piece to generate underneath a section of building, similarly it wouldn't make sense for anything to generate above the roof piece. This adds a lot of complexity, the main consequence of which is a much higher chance of the algorithm failing due to running out of possibilities for a section.

The function of this algorithm in the context of this project is to layout the biomes outlined in section 2.3. The constraints we set are similar to how they were for the tiles. Most biomes can only be next to the plains biome and the ocean biome can only be next to the islands biome. This ensures that we don't have a desert next to a forest, for example, and the lets the coastline have a more gradual transition into ocean with shallower waters around the islands.

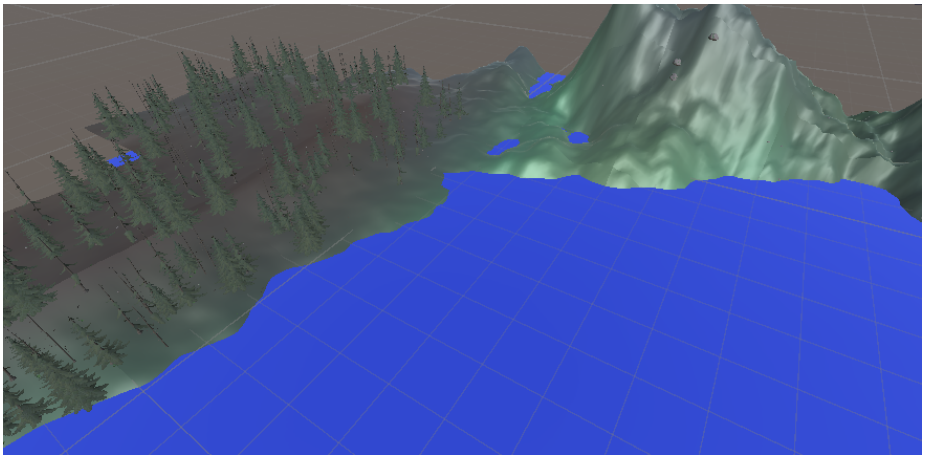


Figure 45: Biomes Placed Using WFC

This approach removes the ability to have an infinitely generated map, or at the very least WFC can only be used for a finite section. The noise function(2) decides on a value for a point by taking the location of nearby biomes and interpolating(5) between them. Therefore biomes are only located at a single point. WFC takes a grid of points and decides where the biomes go, given the set of constraints, creating a somewhat natural layout for the map. Any point outside of this grid of biomes is considered to be ocean, leaving us with an island of varying terrain.

7 Unity Implementation

Now that all of the systems have been constructed, we need to assemble the project in unity.

7.1 Initialization

The Map Manager script I wrote gives us a few parameters to adjust in the unity editor to initialize the program as required. We can adjust the chunk size, which is very important

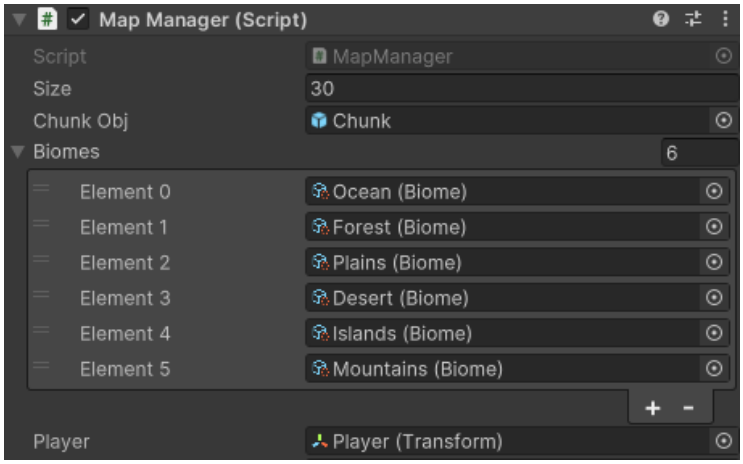


Figure 46: Map Manager Parameters

for optimisation. Also, all of the biome parameters are stored within sciptable objects, all of which can be freely adjusted depending on how we want our terrain to look.

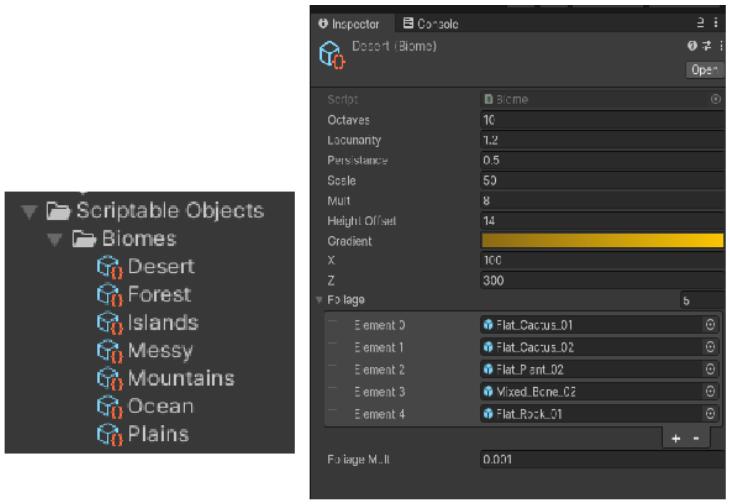


Figure 47: Biomes

7.2 Manager Scripts

At runtime, the Map Manager Script is what controls all of the terrain generation. On Awake, which is a unity function that is called first on the startup of a program, initializes our seed, noise map, and lists/dictionaries. We then call the ChunkLoad function for every

```
void Awake()
{
    biomeData = new Dictionary<BiomeType, Biome>()
    {
        {BiomeType.ocean, biomes[0] },
        {BiomeType.forest, biomes[1] },
        {BiomeType.plains, biomes[2] },
        {BiomeType.desert, biomes[3] },
        {BiomeType.islands, biomes[4] },
        {BiomeType.mountains, biomes[5] },
    };
    xSize = size;
    zSize = size;
    seed = new Seed();
    noiseMap = new NoiseMap(seed, BiomesGenerator.GetBiomes());
    chunks = new Dictionary<Vector2Int, Chunk>();
    roads = new List<Road>();
    locations = new List<Location>();
}
```

Figure 48: Code Implementation

frame. This function checks a 9x9 grid of chunks around the player and if they are within a radius they get added to a list. Every chunk in this list gets loaded if it hasn't been already. All loaded chunks which are not in this list get deleted. This is what allows us to have a virtually unlimited procedural world.

```
void ChunkLoad()
{
    Vector2Int currentChunk = ChunkFuncs.ObjectCurrentChunk(player.position);
    List<Vector2Int> nearbyChunks = new List<Vector2Int>();

    float radius = size * 4f;


    nearbyChunks.Add(currentChunk);
    for (int x = -4; x < 5; x++)
    {
        for(int z = -4; z < 5; z++)
        {
            if (!(x == 0 && z == 0) && ContainedInCircle(currentChunk + new Vector2Int(x, z), radius, player.position - new Vector3(size / 2f, 0, size / 2f))) nearbyChunks.Add(currentChunk + new Vector2Int(x, z));
        }
    }

    foreach (var chunk in chunks)
    {
        if (!nearbyChunks.Contains(chunk.Value.index))
        {
            StartCoroutine(DeleteChunk(chunk.Key));
        }
    }
    foreach(var current in nearbyChunks)
    {
        Chunk checkChunk;
        if (!chunks.TryGetValue(current, out checkChunk))
        {
            StartCoroutine(GenerateChunk(current));

        }
    }
}
```

Figure 49: Code Implementation

7.3 Location System

7.3.1 Purpose

While this project was capable of generating large amounts of terrain with varying biomes, it is missing the ability to generate more specific points of interest and terrain features, such as a river or a dirt path. For this purpose we need to write some sort of system for generating these structures while making sure that they load and unload alongside the chunks(4). It wouldn't be optimal to have an object loaded on one side of the island while a player was standing on the other end. This is where the location system comes in. It is effectively a set of 4 lists. First, there is the list of vertices which contains the information of any heights that need to be adjusted to incorporate this terrain feature. Then we have a list of colours for these vertices if any need to be changed, e.g. if we were to generate a road then we would have a list of vertices containing the path it follows alongside a list of colours containing the colour of the road. Next, we have a list of objects that need to be spawned. All three of these lists are of the same length. This allows the algorithm to simply iterate over the list of vertices and have the relevant colours and objects for that vertex be contained at the same index in the other lists. Finally, there is a list of the chunks which this terrain feature is a part of. When the game is loading a chunk, it now simply checks if there are any points of interest contained inside of it, meaning that we do not need to have these loaded at all times.

```
public abstract class Location
{
    protected Vector2Int centre;

    public List<Vector2Int> vertices;
    public List<float> vertexValues;
    public List<Color> vertexColours;
    public List<Vector2Int> chunks;
    public List<GameObject> objects;
    3 references
    protected void ConstructLocation()
    {
        CreateShape();
        SetValues();
        SetColours();
        chunks = ChunkFuncs.CheckChunks(vertices);
    }
    4 references
    protected abstract void CreateShape();
    4 references
    protected abstract void SetValues();
    4 references
    protected abstract void SetColours();
}
```

Figure 50: Location Class Code

```
1 reference
public void RenderLocation(Location location)
{
    for(int i = 0; i < location.vertices.Count; i++)
    {
        int posIndex = ChunkFuncs.PositionToIndex(location.vertices[i], index);
        if(posIndex != -1)
        {
            GameObject plant;
            if (foliage.TryGetValue(location.vertices[i], out plant))
            {
                foliage.Remove(location.vertices[i]);
                Destroy(plant);
            }
            if(location.vertexValues != null)
            {
                vertices[posIndex].y = location.vertexValues[i];
            }
            if(location.vertexColours != null)
            {
                colours[posIndex] = location.vertexColours[i];
            }
            if(location.objects != null)
            {
                Vector3 position = vertices[posIndex] + new Vector3(initX, 0, initZ);
                if (location.objects[i]!=null)
                {

                    GameObject obj = Instantiate(location.objects[i], position, Quaternion.identity);
                    obj.transform.parent = transform;
                    foliage.Add(new Vector2Int((int)position.x, (int)position.z), obj);
                }
            }
        }
    }
    UpdateMesh();
}
```

Figure 51: Location Function in Chunk

The Map Manager(7) calls the Render Location function on a chunk to be spawned and passes in the relevant location information. This allows all the location information to be incorporated into the chunk, meaning that when the chunk is destroyed all relevant location information is destroyed as well.

7.3.2 Example

An example of one of these locations is the road class. This class uses the A* pathfinding algorithm to construct a path between two vertices in our terrain. this path takes into account the height of the terrain, prioritising vertices that are closer to sea level. The path traced out by this algorithm is added to the list of vertices for the location. All of these vertices then get coloured a colour of our choosing.

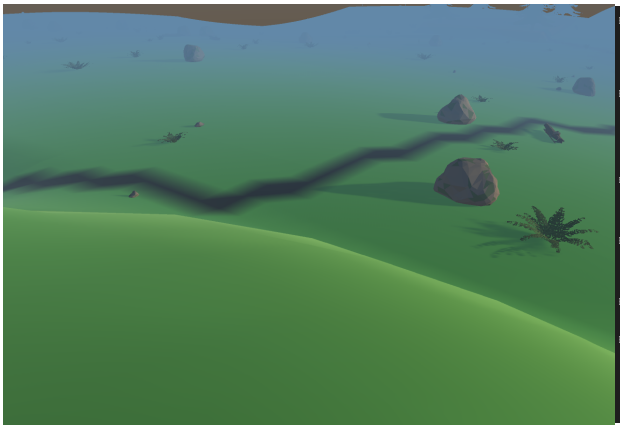


Figure 52: Road Location

```
3 references
public class Road : Location
{
    public Vector2Int end;

    Color colour;
    0 references
    public Road(Vector2Int start, Vector2Int end, Color colour)
    {
        centre = start;
        this.end = end;
        ConstructLocation();
    }
    2 references
    protected override void CreateShape()
    {
        vertices = NoisePathGenerator.GeneratePath(centre, end);
    }
    2 references
    protected override void SetValues()
    {
        vertexValues = null;
    }
    2 references
    protected override void SetColours()
    {
        vertexColours = new List<Color>();
        foreach(var vertex in vertices)
        {
            vertexColours.Add(colour);
        }
    }
}
```

Figure 53: Code Implementation

8 Conclusion

8.1 Results

I believe that all objectives for this project were satisfied. The final result is an amalgamation of multiple methods of procedural content generation into a cohesive game world. The project is also set up in a way which will allow for easy expansion and addition of new features and mechanics. While the overall success of our procedural terrain is a subjective metric, I believe the terrain to be visually appealing and more than sufficient to be used as the basis to build upon for a larger project in the future.

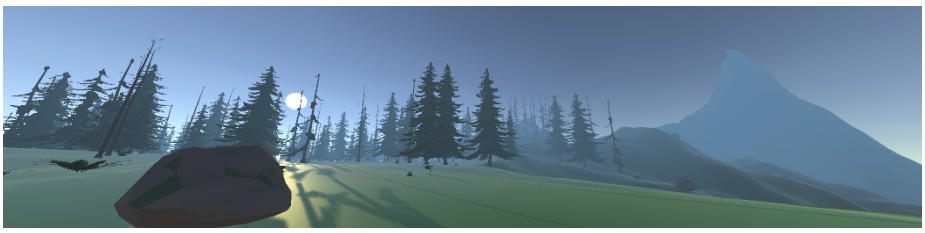


Figure 54: Final Result of Generated Terrain

This use case of Wave Function Collapse along with some of the methods of interpolation are original ideas developed over the course of this project. While Wave Function Collapse is used in many game development projects, it is typically used for generating entire levels or textures/models. I believe that using this algorithm more sparingly in simply laying out the types of terrain, rather than using it to generate the terrain itself, is a more practical use case in most scenarios due to the poor performance of the algorithm. In this regard the algorithm use has been quite successful in assisting in the generation of the main island of the game world.

8.2 Thoughts and Challenges

I believe that looking at this subject from a mathematical perspective, rather than a computer science perspective, led to certain unique solutions to problems that I would not have used otherwise.

For example, my method of uniquely seeding the chunks(4) was inspired by the proof of the countability of the rational numbers. One of the main challenges that needed to be overcome during the development of this project was the interpolation of the biome functions. This system went through many iterations over the course of the year, with the version outlined in the project being my final solution. I believe there may be more elegant solutions here but this setup certainly provides good results

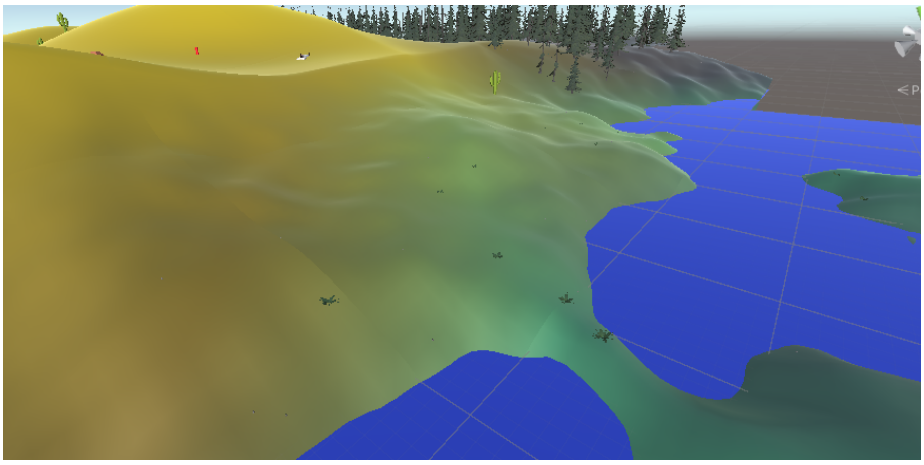


Figure 55: Terrain and Terrain Colour Interpolation

I decided to go with a colour gradient for the ground colouring rather than a textured approach. While I believe this is sufficient and allows for easily adjusting the terrain colour based on height, going with textures might have been a more visually appealing option while allowing for greater variety in terrain. This is possibly something to experiment with in the future.

8.3 Further Work

8.3.1 3D Terrain Integration

Integrating the 3D terrain generated by marching cubes is one of the major features I want to add in the future. Currently it operates as a separate scene which is still useful in that the player can simply transition to the cave scene when required. I believe it would be much better to have these caves integrated into the main overworld scene as explorable underground zones. This would involve combining the 3D noise map class with the regular noise map class and having marching cubes render all 3D terrain that is below ground level, while also leaving openings in the 2D Noise Terrain to give access to these caves systems.

There are many problems to overcome with this implementation and I believe it will be a relatively large undertaking, but one I would love to tackle in the future.

8.3.2 Integration of Additional Game Systems

As stated in the introduction(1), this project was developed with the intention of supporting the work of a graduate project in procedural narrative puzzle generation. This project will be used as a foundation for adding their systems. This will involve porting the project to the Unreal Engine and C++.

8.3.3 Optimisation

Perhaps the most important work to be done in the future is the optimisation of the multiple methods of procedural content generation explored here. While this project was focused on the efficacy of combining these various methods of PCG, it left a lot of the optimisation work to the side.

There are plenty of routes to go down to achieve higher performance. A good example of this would be the use of shaders to render portions of the map. Shaders are separate programs which run on the GPU rather than the CPU. While the CPU is good for running a computing complex commands, the GPU excels at performing many simple commands simultaneously. This is because GPUs are created with the intention of updating individual pixels on your monitor every frame. This functionality can be repurposed to simultaneously compute many of the noise values required at various parts of both the 2D and 3D noise generation.

9 References

[1] F. Kenton Musgrave, *Procedural Fractal Terrains*, FractalWorlds.com, 2015.

[2] Maxim Gumin, *Wave Function Collapse*, GitHub repository, github.com/mxgmn/WaveFunctionCollapse, 2022.

[3] Claude Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal, 1948.

[4] Paul Merrell, *Example-Based Model Synthesis*, Symposium on Interactive 3D Graphics (i3D), 2007.

[5] Ken Perlin, *An Image Synthesizer*, ACM SIGGRAPH Computer GraphicsVolume 19Issue 3, 1985.

[6] Stefan Gustavson, *Simplex Noise Demystified*

[7] Plausible Concept, *Bad North* 2018, Video Game.

[8] Minecraft Community, *Minecraft Size of the World* minecraft.gamepedia.com/The_Overworld, 2017.

[9] Mojang, *Minecraft* 2011, Video Game.

[10] Marian42, *Infinte Procedurally Generated City With the Wave Function Collapse Algorithm* marian42.de/article/wfc/, 2019.

[11] Mat Muze, *2D Value noise rescaled and added onto itself to create fractal noise* 2011. https://en.wikipedia.org/wiki/Value_noise#/media/File:Value_noise_2D.png

[12] Hugo Elias, *Perlin Noise*, https://web.archive.org/web/20080724063449/http://freespace.virgin.net/h Explanation of Value Noise Mislabeled as Perlin Noise.

[13] Unity, *Common Unity Terms* https://unity.com/how-to/beginner/game-development-terms#common-unity-terms. https://en.wikipedia.org/wiki/Value_noise#/media/File:Value_noise_2D.png

[14] Harvey Cline and William Lorensen, *System and method for the display of surface structures contained within the interior region of a solid body*, 1987.

[15] Matthew Fisher, *Marching Cubes*, Matt's Webcorner, Stanford, 2014.

[16] Noor Shaker, Julian Togelius, Mark J. Nelson, *Procedural Content Generation in Games*, Springer International Publishing, 2016.

[17] Sebastian Lague, *Procedural Terrain Generation* Youtube Series, https://www.youtube.com/@SebastianLague/playlists, 2016.

[18] Seliel the Shaper, *Mana Seed Spring Forest* https://seliel-the-shaper.itch.io/spring-forest, 2020.

[19] Proxy Games, *Low Poly Free Vegetation Kit* https://assetstore.unity.com/packages/3d/environments/poly-vegetation-kit-lite-176906.

[20] 23 Space Robots and Counting..., *Free Low Poly Desert Pack* https://assetstore.unity.com/packages/3d/environments/low-poly-vegetation-kit-lite-176906.

[21] Matthew Szudzik, *An Elegant Pairing Function* Wolfram Science Conference, 2006.

[22] Brian M. Scott (https://math.stackexchange.com/users/12042/brian-m-scott), On a two dimensional grid is there a formula I can use to spiral coordinates in an outward pattern?, URL (version: 2023-05-18): https://math.stackexchange.com/q/163093