WETHERBY
SENIOR SCHOOL

LEVEL 3 AQA EXTENDED PROJECT QUALIFICATION

Is it possible to create a procedurally generated terrain simulation using Perlin Noise in Unity?

Jules Tournier
(Candidate Number 1653)

Table of Contents

Introduction

The development of a complex game is a time-consuming task that requires a significant amount of content generation, including terrains, objects, characters, etc. This requires a lot of effort from a designing team. However, you can generate resource saving terrain, characters and more through Procedural Generation. This means that with little to no input, you can program infinite content for your players. PG can be used to create environments (Vandrake, 2020). In this project, I will delve into examples of procedural generation, and create my own procedurally generated landmass simulator in Unity.

Unity is a game engine, designed to make 2D and 3D simulations and games using C#. I will also mention how I will assess my program using user testers. Hopefully with all this information I would be able to create a 2D procedurally generated terrain simulator that passes the criteria's I set for it

I chose this topic for a couple reasons. Firstly, I was very interested in software and game development and enjoyed learning the different ways programmers tackled issues. I also recently played a game called "No man's sky" and was amazed with how they not only created procedural terrain, but an entire procedural universe. Finally, I am interested in studying computer science in university and believed that coding an artifact for my extended project qualification may help me to understand how to tackle similar assignments I may be set in university.

I hope my code can be used as the foundation to build an app or a piece of software that will be used to teach students about geography and changes in elevation in real world terrain

Early History:

Before Procedural Landmass Generation, all games had pre-determined worlds. The first game with procedural world generation, was the 1978 game 'Beneath Apple Mano', a game designed for the Apple II. It had randomly generated dungeons, which was the first existing version of procedural world generation (*An Analog History of Procedural Content Generation: by Gillian Smith*).

Modern Usage:

Nowadays, Procedural Landmass Generation, is used to create infinite land with high detail. It can even have randomised plants, animals, mountains, and caves. An example of this is Terraria, which is a 2-dimension world made of large pixels. These large pixels are the building blocks of the game, as they are used to design the world which the player stands on. These pixels are placed by the computer and will generate a procedurally generated random world every time.

Procedural Generation can also be used to quickly create a visualisation of populated cities (Werner Gaisbauer, 2020) or can be used in education to generate new and random math questions to help teachers provide material for their students (Yi Xu, 2021). Another example is simulating procedurally generated crowds. This can be used to plan airports or other populated areas. (Price, 2017)

Another very similar form of landmass generation which also utilises Procedural Generation is the random generation of cave formations. (Benjamin Mark, 2020)

No Man's sky (Example):

'No Man's sky' is a game developed by Hello games in 2016. What's unique about this game is that it consists of 18 quintillion planets, each completely unique and fully explorable. Each planet is extremely detailed, as well as being proportionally sized compared to our planetary counterparts. For example, to get to the other side of a planet by foot could take some weeks! Furthermore, each particle, rain drop, snowflake or blade of grass is accounted for and randomly generated through an extremely complex program.

The director of the game, Sean Murray, said that "When you're on a planet, you're surrounded by a skybox—a cube that someone has painted stars or clouds onto. If there is a day to night cycle, it happens because they are slowly transitioning between a series of different boxes. The skybox is also a barrier beyond which the player can never pass. The stars are merely points of light. In No Man's Sky however, every star is a place that you can go. The universe is infinite. The edges extend out into a lifeless abyss that you can plunge into forever."

To achieve this, the team at Hello Games had to code a physics engine of 600,000 lines, to simulate planets, suns, moons, asteroids, and space stations. To create this incredibly large universe, it started with a single input of an arbitrary numerical seed. This number is mutated into more seeds by cascading a series of algorithms (also called a computerized pseudo-randomness generator). These seeds will determine the characteristics of each game element. Once the seed is entered within the program, every star, planet, and organism are fixed indelibly. The only way to alter these are from forces outside the system, in this case the player.

"Because it's a simulation," Murray stated. "there's so much you can do. You can break the speed of light—no problem. Speed is just a number. Gravity and its effects are just numbers. It's our universe, so we get to be Gods in a sense."

Theory

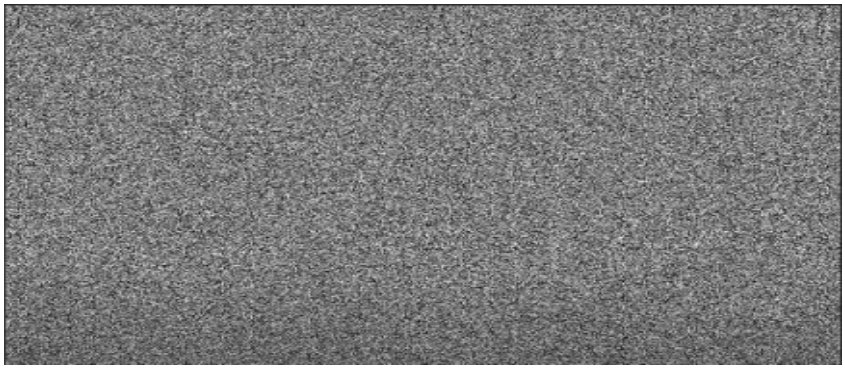
In this section, I will outline the basic theory behind PLG though noise generation. There are 4 different types of noise considered in my analysis: Regular noise, simple X noise and Perlin noise.

Alternatives: I also explored alternative methods of Procedural Generation, but ultimately decided that noise was the most effective for large scale terrain generation. However, another great one was the marching cube algorithm. But it was slightly more effective when it came to designing PLG caves or ocean floors. (Generating Complex Procedural Terrains Using the GPU, 2020)

Regular noise:

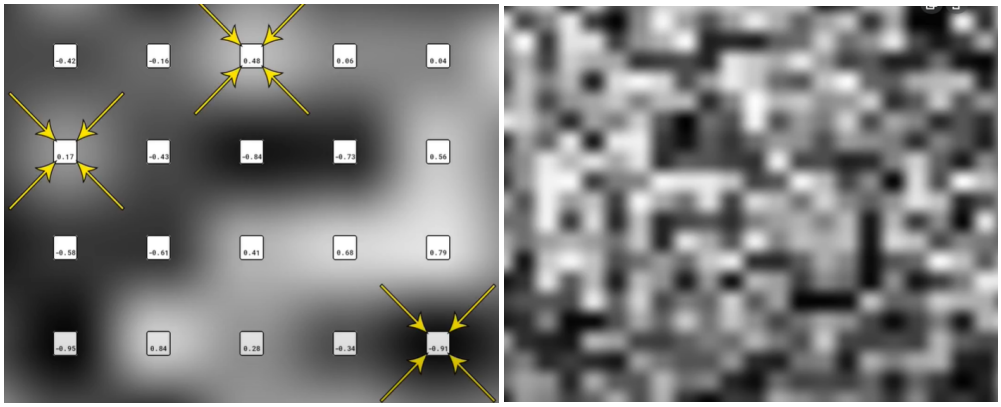
In this section, I will outline the basic theory behind PLG though noise generation. There are 4 different types of noise considered in my analysis: Regular noise, simple X noise and Perlin noise.

Regular noise:



The above image is a section of regular noise. It is a random assortment of black and white pixels, black representing 0 and white representing 1. For my procedurally generated terrain, I plan to assign these numbers to height levels and create a 2-dimensional terrain where values between 0 and 1 represent hights of the terrain. The issue with using regular noise of noise, is that there is no value in between 0 and 1. Using regular noise would make the terrain incoherent and abrupt.

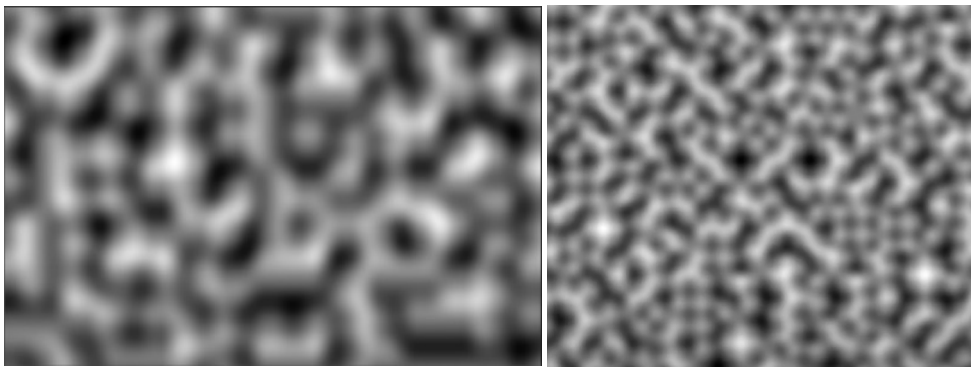
Value noise:



A second form of noise is 'value noise'. This works similarly to regular noise, however beyond black and white, it also has grey pixels. These pixels have values between 0 and 1, meaning that we have a much wider array of values including decimals. This form of noise consists of evenly spirited chunks with a point at the centre which has either the highest or lowest value (represented by the squares in the image above). The colours in between two of these points will create a gradient showing the

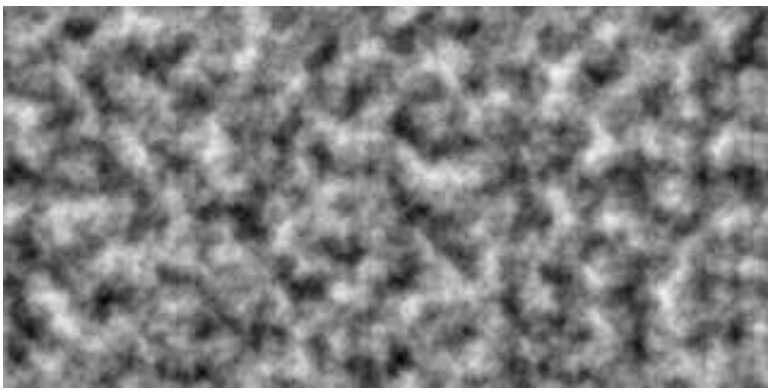
value increasing/decreasing along the way to equal the value of the centre point. This is further demonstrated when increasing the frequency (decreasing the distance between the centre values) of the noise, revealing that it looks pixelated and having sharper edges (as seen on the image on the right). This tells us that the noise is structured using central points.

Perlin noise:

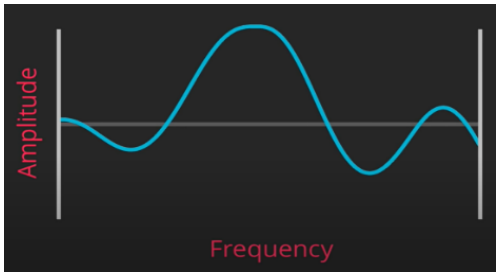


The above image is a section of Perlin noise. This form of noise is also a pseudorandom assortment of pixels. It also has values in-between the black and white base values. However, the main difference is the way its gradient works. Where the value noise has a non-random gradient, the Perlin noise's gradient does not change entirely based off the value next to it, but instead includes an element of randomness. This creates a smoother and reliable form of random generation, as seen when you increase the frequency with the image to the left. My first step in coding the procedural generation terrain will be to create this.

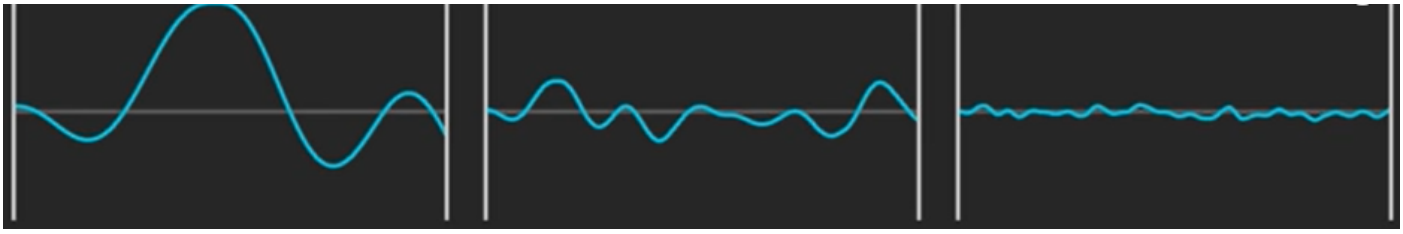
Fractal Perlin noise:



This last image is a section of fractal Perlin noise. This is a more developed version of Perlin noise which is achieved by introducing a new value into the equation called octaves. Octaves gives the noise map much more detail and depth. To explain and visualise this further, the image of a wave bellow is the 2-demensional representation of a region of Perlin noise. The amplitude represents the height level, and the frequency represents the distance between each pixel.

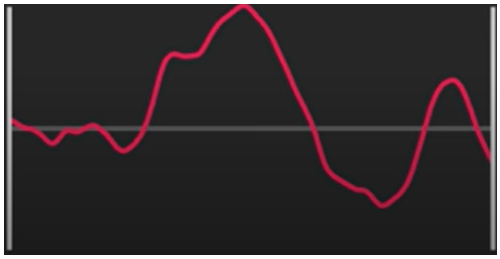


However, the above image of Perlin noise isn't detailed enough. To further improve on it we will introduce additional octaves. Each octave will increase in detail to represent different parts of the landscape. As shown in the below illustration, Octave 1 will be the main outline, therefore being the smoothest and least detailed. Octave 2 will be more detailed and will be for more in-depth detail such as rock, grass, and sea. Finally, Octave 3 will represent the smallest details, such as sand and snow. The next two additional values will add more control to the configuration of octaves. The first one is called Lacunarity. This value increases the frequency of each octave to the power of 1 (as shown in the images below). The second is persistence. This works just like lacunarity but effects the amplitude of each octave instead of the frequency. As each octave increases in detail, its influence should diminish. For example, snow and sand should have less of an effect on the shape and outline of the main terrain shape.



Octave 1	Octave 2	Octave 3
e.g. (The basic terrain shape)	e.g. (Mountain, oceans, green spaces)	e.g. (Sand, snow)
Frequency = lacunarity ^ 0	Frequency = lacunarity ^ 1	Frequency = lacunarity ^ 2
Amplitude = persistence ^ 0	Amplitude = persistence ^ 1	Amplitude = persistence ^ 2

By combining the extra detail of octaves and the control gained from persistence and lacunarity, we get a far more detailed height map which, when tweaked, can create realistic terrain (this is represented by the wave diagram below). This improved method of Perlin noise is one of the main features I will be trying to code, to create a procedurally generated terrain simulator.



Noise generation can go even further when combined with 3D worlds and deep learning

Coding The Procedurally Generated Terrain Simulation

Selecting the game engine and coding language:

My first step in coding the procedural generation simulation was choosing the game engine and the coding language.

For the game engine, my choices were between Unity, Python, and Unreal engine. The reason I didn't choose Python is that, although I am already comfortable with that coding language, it isn't made for this kind of project. Python is great for calculations and number-based simulations; however, it is not powerful enough for this level of simulation. Unreal engine is very similar to Unity, but according to *star loop studios*, Unity is a beginner friendly game engine with a wide variety of functions, whereas Unreal is more advanced and has higher performance and graphics. Furthermore, Unity runs on C# which was a coding language I was already interested in learning. However, this still meant that I had to learn a new coding language from scratch. To do this I used multiple resources such as the *free coding course from Code Academy*, as well as reading books on the subject (Ferrone, 2021)

Selecting the procedural generation method:

The next step was choosing how I was going to create this procedural generation. From the research I gathered, the best two methods are either Perlin noise, or the Marching Cubes algorithm. After reading *"Generating Complex Procedural Terrains Using the GPU" from the "GPU gems 3" book by Nvidia*, I've learned that the Marching Cube algorithm is better suited for more specialised procedural generation such as rocky formations, caves, and the ocean floor. However, Perlin noise procedural generation is great for 2D infinite generation, as seen in the "*Building an Infinite Procedurally Generated World" article by Shawn Anderson.*

Coding the procedural generation terrain:

I went through four stages of development to achieve my procedurally generated terrain simulation. Firstly, I created a base Perlin noise and found a way to display it. Secondly, I increased the complexity of the noise generation by adding variables to generate a fractal Perlin noise. Thirdly, I added colours to simulate terrain. Finally, I turned it into a simulation that a user can experience and interact with.

Generating and displaying base Perlin noise:

In the first stage, I created a base Perlin noise. I accomplished this by making use of the Perlin noise function in C#, available from the unity library.

```
float perlinValue = Mathf.PerlinNoise (sampleX, sampleY);
```

This allowed me to create an array of values between 0 and 1. I then had to generate and display this array on a plane in the Unity editor, as well as figure out its width and height. This process would effectively create the basis for the land map by using the numerical values that had been mathematically generated

```
using UnityEngine;
using System.Collections;

Unity Script | 3 references
public class MapGenerator : MonoBehaviour {

    // values defining the map (lacanarity and persistence later)
    public int mapWidth;
    public int mapHeight;
    public float noiseScale;

    2 references
    public void GenerateMap() {
        float[,] noiseMap = Noise.GenerateNoiseMap (mapWidth, mapHeight, noiseScale); //fetches the 2D noise map from the Noise script


        MapDisplay display = FindObjectOfType<MapDisplay> (); //calls the mapdisplay with the noise map
        display.DrawNoiseMap (noiseMap);
    }

}
```

This code creates values for the map width, height and scale; it also stores the array of values from the 'Mathf' function, so that the map display script has easy access to it.

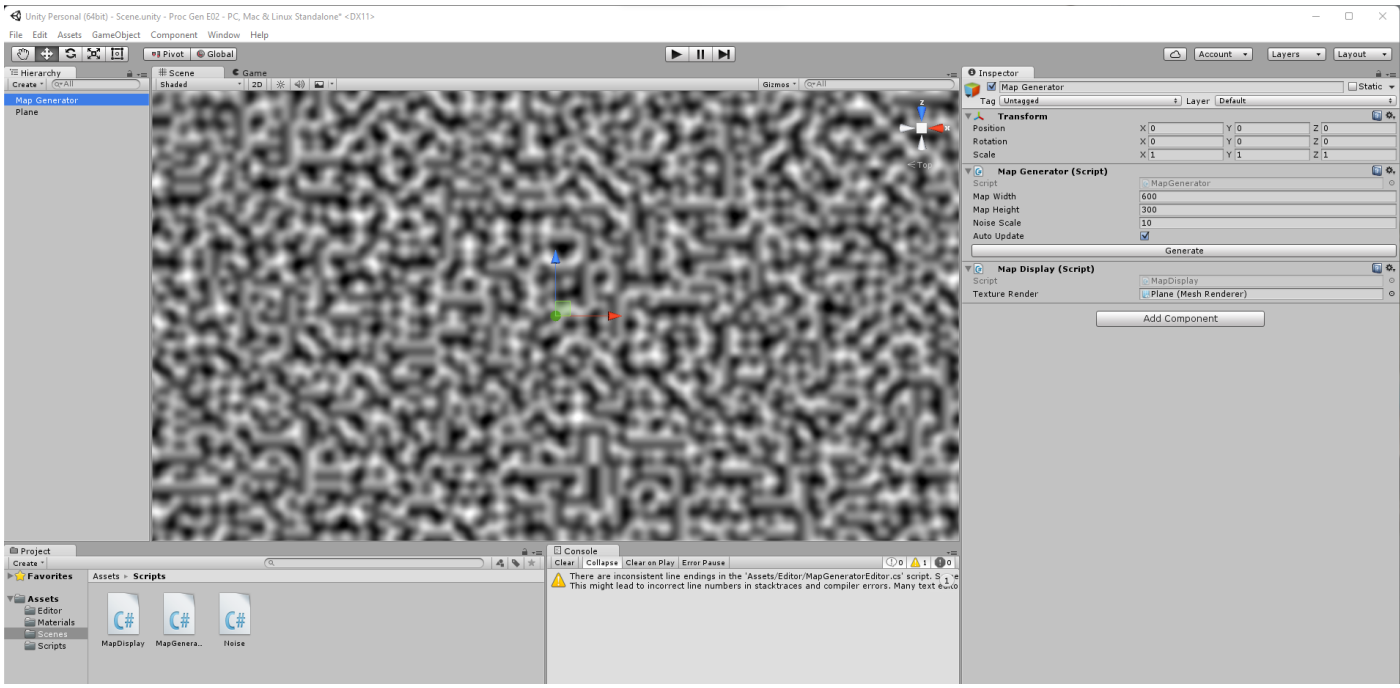
```
using UnityEngine;
using System.Collections;

Unity Script | 2 references
public class MapDisplay : MonoBehaviour {

    public Renderer textureRender; // references the redered of that plane to be able to set its texture

    1 reference
    public void DrawNoiseMap(float[,] noiseMap) { //takes in 2D float arrar of noise map
        int width = noiseMap.GetLength (0); // figures out the width and height of the noise map
        int height = noiseMap.GetLength (1);

        Texture2D texture = new Texture2D(width, height); // creates 2D texture with a width and height

        Color[] colourMap = new Color[width * height]; // set color of each pixel in the texture with an array
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                colourMap [y * width + x] = Color.Lerp (Color.black, Color.white, noiseMap [x, y]); // creates the index for the map, and then sets the values between 1-0 with a color between black and white.
            }
        }
        texture.SetPixels (colourMap); //applies the colors to the texture
        texture.Apply ();

        textureRender.sharedMaterial.mainTexture = texture; // applies the texture to the texture renderer, without entering gamemore
        textureRender.transform.localScale = new Vector3 (width, 1, height); // set the size of the plane to the size of the map
    }
}
```

This code gets the 2D array from the map generator and uses the height and width value of the plane to create a 2D texture. However, I ran into an issue where nothing was being displayed. I learnt that I had to create a material to be displayed on the plane, as a texture cannot be directly projected onto a plane.

Lastly, I created an editor to allow easy access to the values so that I could easily change them. I also added a 'generate' button to display the map on the plane whenever it is pressed, as well as an 'auto generate' function so that the display is automatically regenerated after every change.

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor (typeof (MapGenerator))] //says that it is a custom editor for the Map Generator
Unity Script | 0 references
public class MapGeneratorEditor : Editor {

    0 references
    public override void OnInspectorGUI() {
        MapGenerator mapGen = (MapGenerator)target; // getting a reference to map generator (target, the object the customer editor is editing)

        //DrawDefaultInspector ();
        if (DrawDefaultInspector ()) { //if any value was changed, it will auto update the map (so when the scale was changed so will the pam without having to repress generate
            if (mapGen.autoUpdate) {
                mapGen.GenerateMap ();
            }
        }

        if (GUILayout.Button ("Generate")) { //adds a button, if pressed generates the map
            mapGen.GenerateMap ();
        }
    }
}
```

However, while doing all of this, I ran into an error with the current version of Unity I was using. The issue was that nothing was being displayed and just returned a black plane. I later found the issue could be solved by using a different version of Unity through exploring forums such as GitHub (more specifically version 5.30f4).



This first stage of development allowed me to create this plane of procedurally generated Perlin noise. The next development stage was to turn it into fractal Perlin noise.

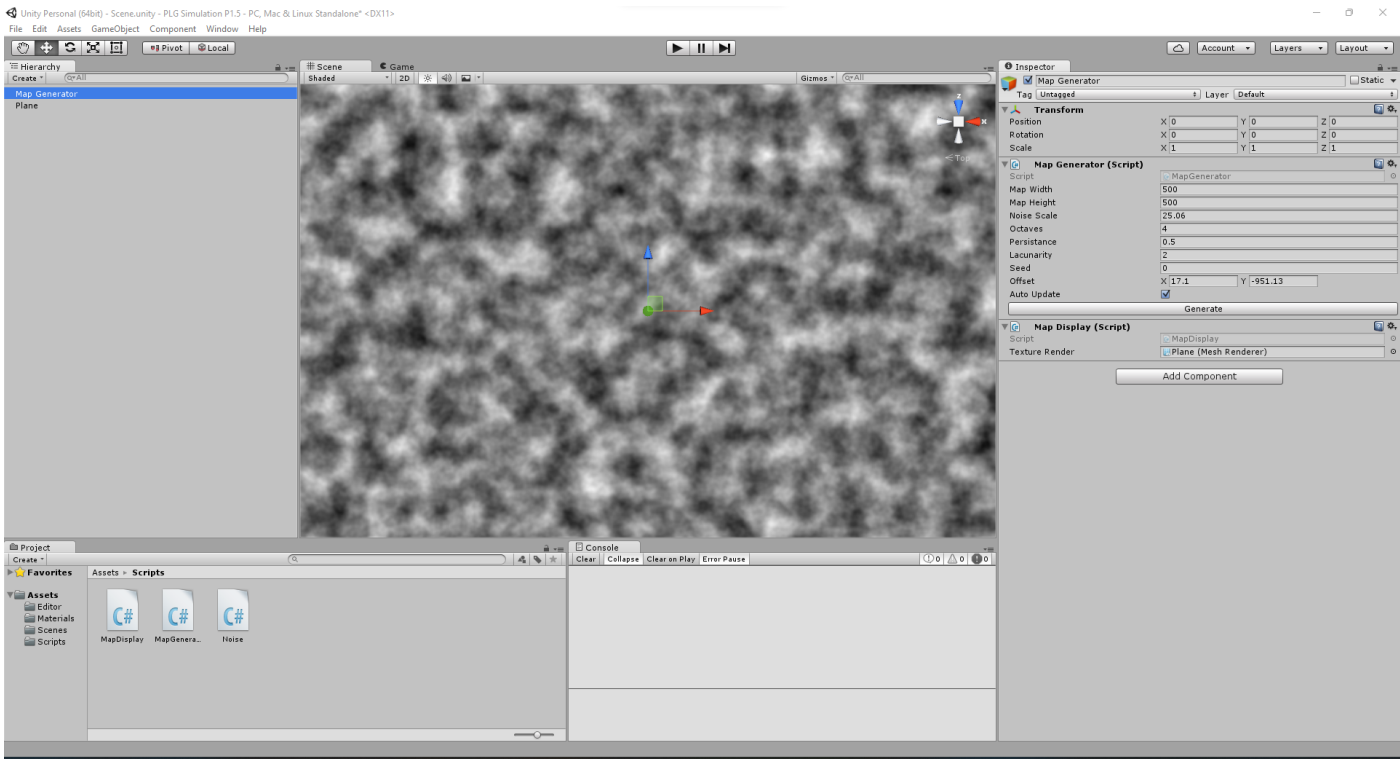
Increasing the complexity of the noise generation through fractal Perlin noise:

To create fractal Perlin noise, I introduced 3 new values to the noise script: persistence, lacunarity and octaves. I explained previously the effect these will have on the code in the theory section of the report.

I also introduced another value called ‘seed’. This is a pseudorandom number that stores the information of the noise values. If the number changes, the values change, creating a new random terrain. However, re-entering the same ‘seed’ number again gives you the previous generated terrain. I also added a value for the offset, which increase the noise value by 1. This is useful for testing, as it allows me to make small and gradual changes to the noise.

```
public static class Noise {

    // defines new octave, persistance, lacunarity and seed values
    public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, float scale, int octaves, float persistance, float lacunarity, int seed, Vector2 offset) {
        float[,] noiseMap = new float[mapWidth,mapHeight];

        // creates a pseudorandom number which is = to the seed
        System.Random prng = new System.Random(seed);
        Vector2[] octaveOffsets = new Vector2[octaves]; //each octave is sampled from a different location from an array
        for (int i = 0; i < octaves; i++)
        {
            float offsetX = prng.Next(-100000, 100000) + offset.x; //can't be too high or it will return the same value
            float offsetY = prng.Next(-100000, 100000) + offset.y;
            octaveOffsets[i] = new Vector2(offsetX, offsetY);
        }

        if (scale <= 0) {
            scale = 0.0001f;
        }

        // This normalises the noise map (by keep its values in the range of 0-1 by keeping track of the values)
        float maxNoiseHeight = float.MinValue;
        float minNoiseHeight = float.MaxValue;

        for (int y = 0; y < mapHeight; y++) {
            for (int x = 0; x < mapWidth; x++) {

                //Frequency and amplitude variables
                float amplitude = 1;
                float frequency = 1;
                float noiseHeight = 0;  // keeps track of the current hight value

                // Loops through the octaves
                for (int i = 0; i < octaves; i++)
                {
                    float sampleX = x / scale * frequency + octaveOffsets[i].x; //the higher the frequency = furether aprt the sample points.
                    float sampleY = y / scale * frequency + octaveOffsets[i].y;  //This means that hight values will change quicker

                    float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;    // This allows us to get better noise values as they can no be -1 - 1 instead of 0 - 1
                    noiseHeight += perlinValue * amplitude; //instead of noise map = perlinvalue, i will increase the noise hight by the perlin value of each ovtave

                    amplitude *= persistance; //amplitude increases each octave
                    frequency *= lacunarity;    // frequency increase each octave
                }
                //if current noise height > then current max noise, the we will update the max noise height to be = to current noise height (same with Noise height < min noise height)
                if (noiseHeight > maxNoiseHeight)
                {
                    maxNoiseHeight = noiseHeight;
                } else if (noiseHeight < minNoiseHeight)
                {
                    minNoiseHeight = noiseHeight;
                }

                // applies noise height to noise map
                noiseMap[x, y] = noiseHeight;

            }
        }
        // Once the range of noise is calculated, we loop through all the values
        // InverseLerp will give us a value between 0 and 1 ("Loops through inverselerp function which is a ... for example in the context of our code, it would...)
        for (int y = 0; y < mapHeight; y++) {
            for (int x = 0; x < mapWidth; x++) {
                noiseMap[x, y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight, noiseMap[x, y]);
            }
        }

        return noiseMap;
    }
}
```

These changes allowed me to create a fractal Perlin noise plane instead of the base Perlin noise plane.



Adding colour to achieve a terrain-like look:

The third stage of development was to implement colour into the plane by assigning the height values between 0 and 1 to colours and creating a new colour map. This was done by setting up thresholds for each tile type you want to support, e.g.:

- water if < 0.3
- grass if >= 0.3 and <= 0.6
- mountain if > 0.6

(Anderson, 2015)

```
using UnityEngine;
using System.Collections;

Unity Script | 3 references
public class MapGenerator : MonoBehaviour
{
    3 references
    public enum DrawMode { NoiseMap, ColorMap};
    public DrawMode drawMode; // create a new toggleable variable for the draw mode to swap between the color and height map without overwriting the code

    [HideInInspector] public int mapWidth;
    [HideInInspector] public int mapHeight;
    public float noiseScale;

    public int octaves;
    public float persistance;
    public int lacunarity;

    public int seed;
    [HideInInspector] public Vector2 offset;

    public bool autoUpdate;

    public TerrainTypes[] Biomes; //adds the biome, color and heightmap it represents in the editor

    2 references
    public void GenerateMap()
    {
        float[,] noiseMap = Noise.GenerateNoiseMap(mapWidth, mapHeight, noiseScale, octaves, persistance, lacunarity, seed, offset);

        Color[] colorMap = new Color[mapHeight * mapHeight]; // creates a new map called color map

        for (int y=0; y<mapHeight; y++) // this loops through the noise to find the height value
        {
            for (int x=0; x<mapWidth; x++)
            {
                float height = noiseMap[x, y];  // once the height value has been found it associates to the biome
                for (int i=0; i < Biomes.Length; i++)
                {
                    if (height <= Biomes [i].height)
                    {
                        colorMap[y * mapWidth + x] = Biomes [i].color; //saves the colour for that point
                        break;
                    }
                }
            }
        }
        MapDisplay display = FindObjectOfType<MapDisplay>(); //toggles the draw mode
        if (drawMode == DrawMode.NoiseMap)
        {
            display.DrawTexture(TextureGenerator.TextureFromHeightMap(noiseMap)); //draws the differnt types of maps
        }
        else if (drawMode == DrawMode.ColorMap)
        {
            display.DrawTexture(TextureGenerator.TextureFromColorMap(colorMap, mapWidth, mapHeight));
        }
    }
}

[System.Serializable]
1 reference
public struct TerrainTypes  // assign 0-1 values to colors
{
    public float height;    // float for heihgt value
    public Color color;     // float for color value
    public string name;     // Type of terrain
}
```

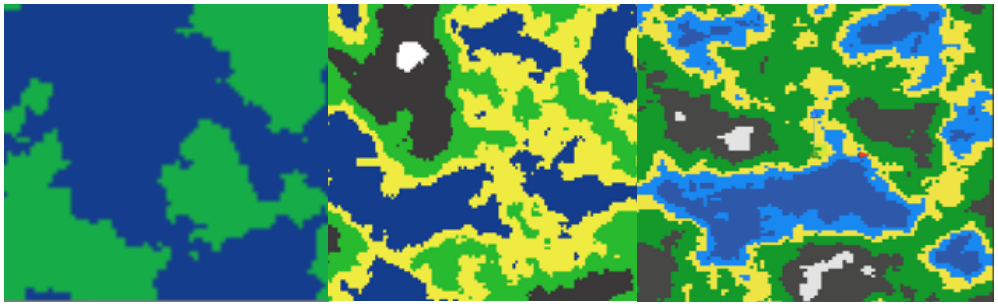
I first created a variable called ‘terrain type’ to store the colour, name, and height value (e.g., white, snow, 1). I then separated the noise map and the colour map by adding another variable called the ‘draw mode’ to toggle between the two. Finally, I created a new script called the texture generator to generate the colour map.

```
using UnityEngine;
using System.Collections;

2 references
public static class TextureGenerator //seperated the texture generation code to make it cleaner and easier to work with
{

    2 references
    public static Texture2D TextureFromColorMap(Color[] colourMap, int width, int height) //creates a texture from the color map
    {
        Texture2D texture = new Texture2D(width, height);
        texture.SetPixels(colourMap);
        texture.Apply();
        return texture;
    }


    1 reference
    public static Texture2D TextureFromHeightMap(float[,] heightMap) //this method gets a texture from a 2D heihght map
    {
        int width = heightMap.GetLength(0); // this is the code used to create the height map before
        int height = heightMap.GetLength(1);

        Color[] colourMap = new Color[width * height];
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                colourMap[y * width + x] = Color.Lerp(Color.black, Color.white, heightMap[x, y]);
            }
        }

        return TextureFromColorMap(colourMap, width, height); //instead of creating the texture above, we return the results from the methods above, this is quicker
    }

}
```

This script holds the code for both the noise map made previously and the colour map. This finally gives us the colour map. However, the map was blurry and pixelated so I added two new functions.

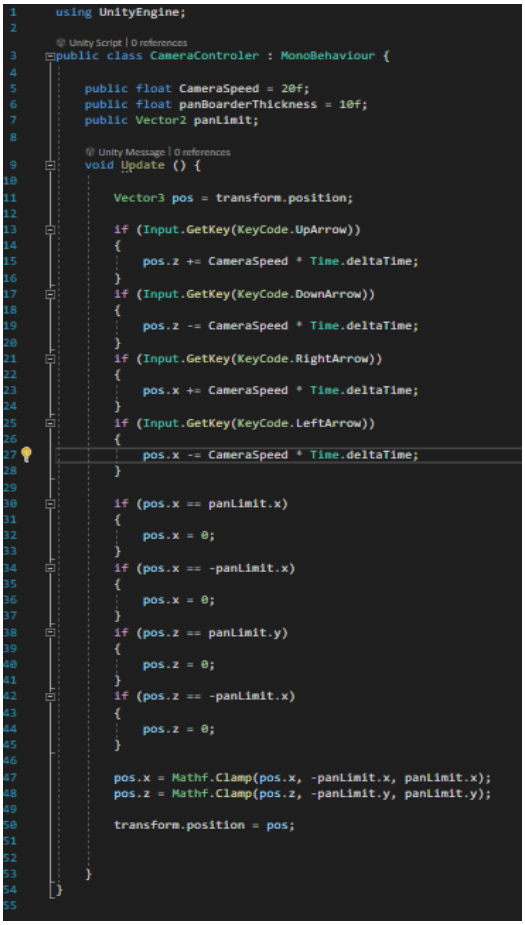
```
texture.filterMode = FilterMode.Point; //creates the pixelated texture
texture.wrapMode = TextureWrapMode.Clamp; //removes bluriness
```

All that was left was to find the right colour gradient. I went through multiple trials to finally find one colour gradient that looks realistic and proportional to real world geography.



Creating a user-friendly simulation:

In the final stage of development, I needed to create something that allowed the user to move around the terrain. I decided that the best method was to create a camera script to allow a user to move around the procedurally generated terrain. However, since the terrain isn't infinite, I needed to create something that prevented the camera from going too far past the edge of the terrain. To do this, I created a border preventing the camera going past the plane. I also made it possible for the camera teleports back to the centre of the plane when it reaches the end of the plane or border.



```
using UnityEngine;

Unity Script | 0 references
public class CameraControler : MonoBehaviour {

    public float CameraSpeed = 20f;
    public float panBoarderThickness = 10f;
    public Vector2 panLimit;

    Unity Message | 0 references
    void Update () {

        Vector3 pos = transform.position;

        if (Input.GetKey(KeyCode.UpArrow))
        {
            pos.z += CameraSpeed * Time.deltaTime;
        }
        if (Input.GetKey(KeyCode.DownArrow))
        {
            pos.z -= CameraSpeed * Time.deltaTime;
        }
        if (Input.GetKey(KeyCode.RightArrow))
        {
            pos.x += CameraSpeed * Time.deltaTime;
        }
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            pos.x -= CameraSpeed * Time.deltaTime;
        }

        if (pos.x == panLimit.x)
        {
            pos.x = 0;
        }
        if (pos.x == -panLimit.x)
        {
            pos.x = 0;
        }
        if (pos.z == panLimit.y)
        {
            pos.z = 0;
        }
        if (pos.z == -panLimit.x)
        {
            pos.z = 0;
        }

        pos.x = Mathf.Clamp(pos.x, -panLimit.x, panLimit.x);
        pos.z = Mathf.Clamp(pos.z, -panLimit.y, panLimit.y);

        transform.position = pos;


    }
}
```

For a more indepth report on the steps I went through to code my procedural generation simulation, please read the Coding log, located in the appendix.pr

Success Criteria

My objective with this EPQ was to develop a procedurally generated terrain simulation. To determine whether I had achieved my objective, I needed a way of measuring the quality of my simulation by establishing success criteria, laying out all the functions my program needed to perform.

These criteria include:

- Does the code generate terrain?
- Does the generated terrain include changes in elevation (like hills and mountains) as well as bodies of water?
- Do different seeds equal different terrains, and do the same seeds generate the same terrain again?
- Do the terrains design/texture change depending on height level to match its real-world representation (e.g., tall, generated mountains will have snow at peaks, and deep oceans will have a darker colour)?
- Is the terrain infinite and open to explore?

To identify whether each of these targets have been achieved, I used a 'user assisted testing' (UAT) form. This form assists potential users in testing the program. The purpose of the UAT is to assure that the program can not only accomplish its intended purpose, but also hold up to real world tasks. The tester may be able to identify errors that the programmer may have missed, and they will be able to give new insight on the program and its functionality.

For the UAT to be accurate, it must be carried out through multiple different users to get a wide range of results. Secondly, the users must be trained beforehand, this includes them knowing the success criteria of the program, how to navigate the program and what counts as a bug.

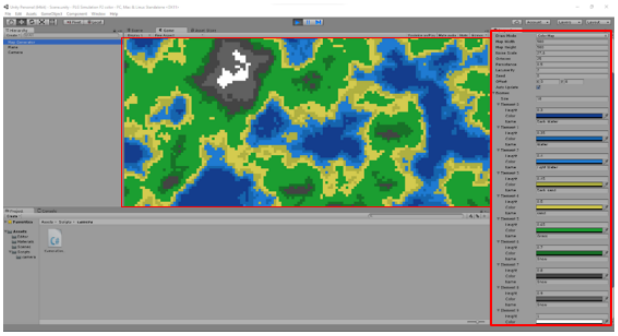
In my user assisted testing, 5 users used my software to identify if any features had been forgotten and if it contained any errors. They each filled out a form, ticking off all the features that had been accomplished and wrote down any errors they encountered.

User Acceptance Testing

A user acceptance testing (UAT) is a form of testing, completed by a possible user, instead of the programmers. The purpose of this is to make sure that the program can not only accomplish its intended purpose, but also hold up to real world tasks. Furthermore, the tester may be able to identify bugs/errors that the programmer may have missed as they will be able to give new insight on the program.

Your task is to use the program, and make sure that it matches the success criteria that it has made for itself. Furthermore, any bugs or errors you encounter while testing should be reported in the appropriate section.

Navigating the program.



The image above shows the screen you will be seeing when testing the artifact. This will be made up of 4 sections. However, you will only need to focus on the section on the right called the inspector, and the middle section, called the 'Game' scene (these two are highlighted in red above).

The inspector consists of multiple different variables, the first one being the draw mode. This will change the 'Game' scene from a colour map to a noise map. The next couple variables are the noise scale, octaves, persistence and the lacunarity. Feel free to change the variables for different results. The default variables for these are Noise scale = 41, Octaves = 26, Persistence = 0.5, Lacunarity = 2. The next variable is the seed. This is a variable that retains the data of the procedural generation it is assigned to and allows you to create a new terrain with every new combination of number, while saving the data so that if you input the same number again, the same terrain will be generated. Finally, we have the Biomes. These last couple of elements are a height value, assigned to a colour. You can change the heights, or the colour they are assigned to, changing the colours in the scene.

Next is the 'Game' scene. This is where you will be able to see the changes you made in the inspector. After making a change, click anywhere on the game scene and then use the arrows on your keyboard to move around.

If you made a change to the inspector and can't change it back, here is how to reset it to the original. Click the play button on the top middle screen, then click it again. Go to the inspector and click on generate (this is situated at the bottom of this inspector when you scroll down), then go back and click on the 'Game' scene to be able to move around again.

User Acceptance Testing

User ID	Tested by	Date tested
002		

Success criteria:

- ☐ Does the code generate terrain, and does it is random/procedurally generated?
- ☐ Does the generated terrain show geological forms like changes in elevation and bodies of water?
- ☐ Do different seeds equal different terrains, and do the same seeds generate the same terrain?
- ☐ Do the terrains design/texture change depending on height level to match its real-world representation (e.g., tall, generated mountains will have snow at peaks, and deep oceans will have a darker colour)?
- ☐ Is the terrain open to explore?

If any criteria were not met, please state why. If they were met, then leave blank.

Criteria number	Reason for unsuccess
1)	
2)	
3)	
4)	
5)	

Bugs/Errors:

If you notice any bugs or errors in your testing that are unrelated to the criteria, please comment them bellow (if possible, please leave screen shots)

The image above shows the sheet that the testers filled out.

Firstly, the UAT Sheet contains instructions on how to navigate the software, so that the user can test the code effectively. Next it contains a table to record the name of the tester and date of testing. Underneath are the success criteria with check boxes for each. This is followed by another table so that the user can provide more details if some criteria were not met. Finally, there is a box to report any errors or bugs.

From the 5 user tests, I received an overall positive response. The testers marked that I have met all my criteria, but they did leave some suggestions. The UAT sheets can be found in the appendix.

User 1 stated that the game window and scene window was inverted so I rotated the plane by 180 so that they were facing the same way. Secondly, the border wasn't positioned properly, meaning that the user could see outside the map. I fixed that my shrinking the border to the size of the plane. Lastly, he stated that the simulation showed a disproportionate amount of sand, compared to the other biomes. I solved this by decreasing the hight range for sand and increasing the range of water. But apart from that, all success criteria were achieved.

User 2 stated that the location of the generate button wasn't well explained, so I changed that in the explanation of the UAT. Secondly the user found that other variables such as octaves weren't well

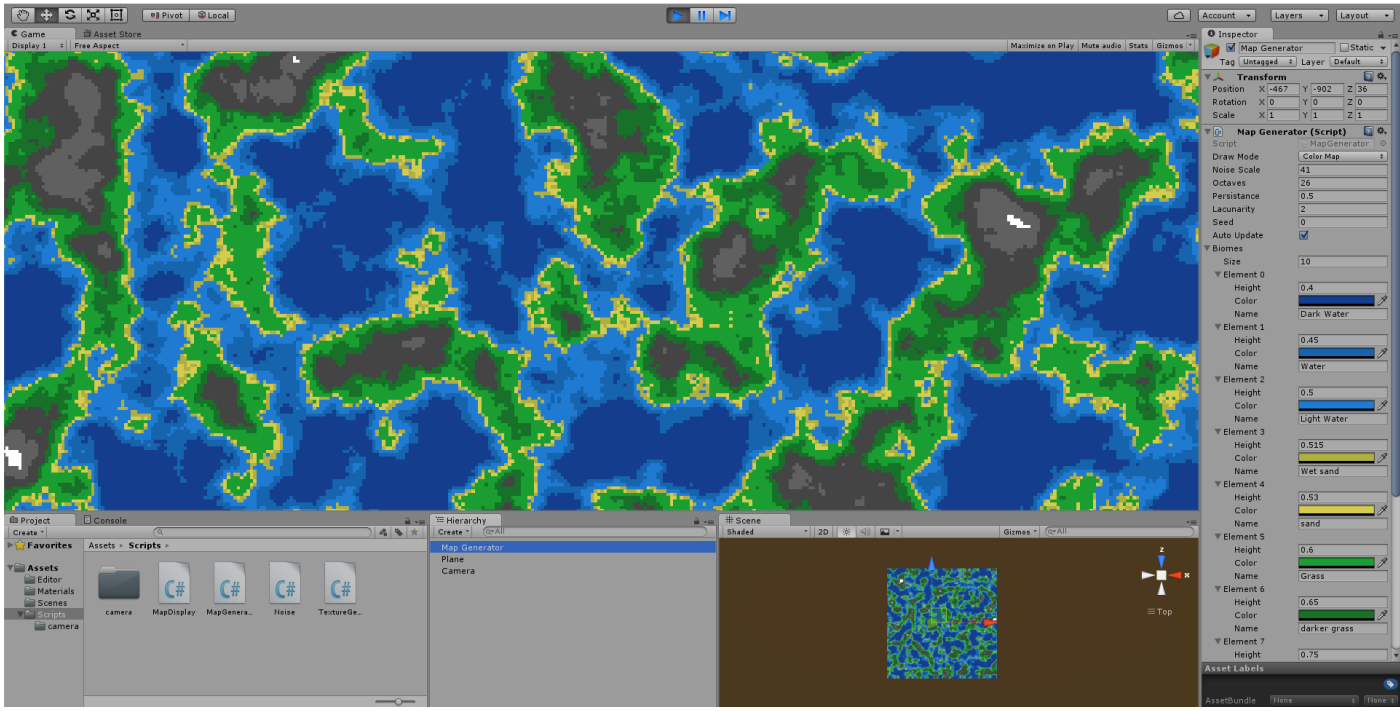
explained so I added an explanation for them. Despite this suggestion, once again all the success criteria were accomplished.

User 3 found no issues and stated that 'The game worked as intended'. They also ticked off all the success criteria

User 4 found a bug where the zoom level resets when the changing the seed. I fixed this by assigning the zoom level to a value and keeping it consistent when there is a change to the map. Apart from this, the user agrees that I have accomplished everything in my success criteria.

Lastly, user 5 found no errors and ticked off all my success criteria.

Conclusion



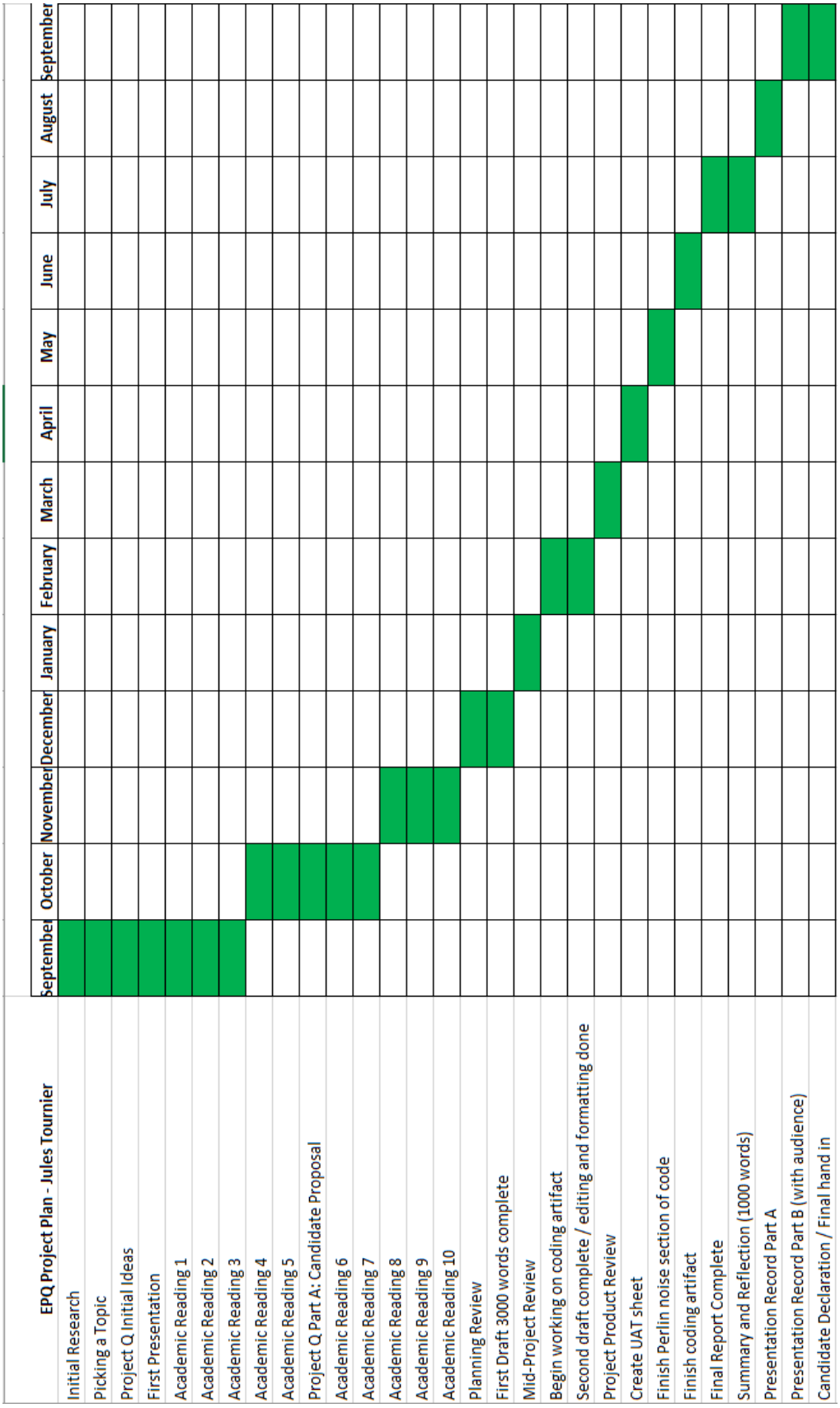
The image above shows my final project. This includes the changes made from the suggestions given to me by the user testers.

Having read over the UAT reviews I received; I concluded that I had succeeded in meeting my success criteria. However, I did have some issues, regarding the user interface and clarity for the tester. But in the end, the overall response was positive, with all agreeing that I had accomplished all the criteria I set out for myself. This was a very challenging project for me to complete, as I not only had to learn about Perlin noise and procedural generation, but I also had to learn the Unity game engine and C#. One of my personal objectives was to learn this new coding language and applying it to this project helped me to learn and improve my knowledge and familiarity of it.

If I were to re-do this project, based on the responses I got and my overall opinion, I would have allocated some time to create a more user-friendly interface for my simulation. Furthermore, I would have wanted to make a 3D simulation instead of 2D as well as make it fully infinite instead of looping back to the beginning. These weren't achieved due to the time constraints of the EPQ, as well as my lack of skill with C# at the time of writing.

I believe my simulation can be used to develop an app/software to teach geography to students. Overall, I am very pleased with the work I have achieved, and I believe that I have answered the question that it is possible to create a procedurally generated terrain simulation using Perlin Noise in Unity.

Appendix 1: Planning Tool (Gant Chart)



Appendix 2: Coding log

Bellow is the log I made, recording my progress during the

Part 1: Creating base Perlin Noise

Basic Perlin Noise Script

This screenshot shows the noise script I have made, allowing me to calculate the numbers (0-1) from a Perlin noise function *(line 8)*. This will then be stored in a 2D array *(line 9)* to be plotted and displayed on a map.

- Map scale = **`(width [x] , height [y])`**
- **`float perlinValue = Mathf.PerlinNoise(sampleX, sampleY);`**
    - Creates the Perlin values, according to the dimensions of the map *(line 17)*

```
1  using UnityEngine;
2  using System.Collections;
3  // not monobehaviour as it wont be applied to an object in the scene
4  // static to make sure it doesn't create multiple instances
   1 reference
5  public static class Noise {
6
7      // create a grid of value from 0-1 from generating a noise map
       1 reference
8      public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight) { // public static method that returns a 2D list of float variable (not whole numbers), as well as creating value for height and width
9          float[,] noiseMap = new float[mapWidth,mapHeight]; // 2D float array = 2D float aray of width by height
10
11         for (int y = 0; y < mapHeight; y++)
12         {
13             for (int x = 0; x < mapWidth; x++) {
14                 float sampleX = x; //figures out which point will be sampling out height and width value from (sample coordinates)
15                 float sampleY = y;
16
17                 float perlinValue = Mathf.PerlinNoise (sampleX, sampleY); // this is the perlin function that will calculate the perlin values
18                 noiseMap [x, y] = perlinValue;
19             }
20         }
21
22         return noiseMap;
23     }
24
25 }
```

However, the value that was returned for x and y was always the same, so I added the scale function to instead get a different value every time. But this created another issue, as whenever the value for scale was 0, it would create a math error as it would be trying to divide the value by 0. To prevent this, I added an if function which would change the value of scale to 0.0001 if it were ever equal to 0 *(line 12-13)*

```
1  using UnityEngine;
2  using System.Collections;
3  // not monobehaviour as it wont be applied to an object in the scene
4  // static to make sure it doesn't create multiple instances
   1 reference
5  public static class Noise {
6
7      // create a grid of value from 0-1 from generating a noise map
       1 reference
8      public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, float scale) { // public static method that returns a 2D list of float variable (not whole numbers), as well as creating value for height and width
9          float[,] noiseMap = new float[mapWidth,mapHeight]; // 2D float array = 2D float aray of width by height
10
11         // makes sure the scale is never 0 so we don't get an error
12         if (scale <= 0) {
13             scale = 0.0001f;
14         }
15
16
17         for (int y = 0; y < mapHeight; y++)
18         {
19             for (int x = 0; x < mapWidth; x++) {
20                 float sampleX = x / scale; //figures out which point will be sampling out height and width value from (sample coordinates)
21                 float sampleY = y / scale; // scale error fixer
22
23                 float perlinValue = Mathf.PerlinNoise (sampleX, sampleY); // this is the perlin function that will calculate the perlin values
24                 noiseMap [x, y] = perlinValue;
25             }
26         }
27
28         return noiseMap;
29     }
30
31 }
```

Map Generator

(This script was applied to an empty object in the scene)

Firstly, this script defines the values of the map's width and height, as well as the Perlin noise scale *(line 7-9)*. Then it fetches the data from the 2D noise map/array from the noise script *(line 12)* and uses the Display function to call the map display from the other script *(line15)*.

```
1   using UnityEngine;
2   using System.Collections;
3
    Unity Script | 3 references
4   public class MapGenerator : MonoBehaviour {
5
6       // values defining the map (lacanarity and persistence later)
7       public int mapWidth;
8       public int mapHeight;
9       public float noiseScale;
10
        2 references
11      public void GenerateMap() {
12          float[,] noiseMap = Noise.GenerateNoiseMap (mapWidth, mapHeight, noiseScale); //fetches the 2D noise map from the Noise script
13
14
15          MapDisplay display = FindObjectOfType<MapDisplay> (); //calls the mapdisplay with the noise map
16          display.DrawNoiseMap (noiseMap);
17      }
18
19  }
```

Map Display

(Script was applied to a plane in the scene).

Firstly, the script uses the rendered plane from the Map Generator script to take in the 2D float array of the noise map *(line 6 & 8)*. Then it calculates the height and width of the noise map and creates a 2D texture with that information *(line 9-12)*. Each pixel is registered in an index and assigned a colour between black and white based on its value between 0-1 *(line 14-17)* and is assigned to the texture *(line 20-21)*. Finally, the texture is applied to the renderer *(line 23)*.

I could have used the texture. **`SetPixel()`**, however it was quicker to make an array of all the colours for all the pixels and set them all at once *(line 14-17)*. Secondly, instead of using textureRender.mat, I used shared material to be able to generate the texture without entering and exiting game mode *(line23)*. Then, I added a line that made sure the size of the map was equal to the size of the texture, as when this was not implemented, it created some visual bugs *(line 25)*.

Finally, the scripts where not displaying anything. The issue was that the texture could not be applied directly to a plane without a material also applied to the plane, so a new material was created, and applied to the plane, so that the texture could be applied to it.

```
1   using UnityEngine;
2   using System.Collections;
3
    Unity Script | 2 references
4   public class MapDisplay : MonoBehaviour {
5
6       public Renderer textureRender; // references the redered of that plane to be able to set its texture
7
        1 reference
8       public void DrawNoiseMap(float[,] noiseMap) { //takes in 2D float arrar of noise map
9           int width = noiseMap.GetLength (0); // figures out the width and height of the noise map
10          int height = noiseMap.GetLength (1);
11
12          Texture2D texture = new Texture2D(width, height); // creates 2D texture with a width and height
13
14          Color[] colourMap = new Color[width * height]; // set color of each pixel in the texture with an array
15          for (int y = 0; y < height; y++) {
16              for (int x = 0; x < width; x++) {
17                  colourMap [y * width + x] = Color.Lerp (Color.black, Color.white, noiseMap [x, y]); // creates the index for the map, and then sets the values between 1-0 with a color between black and white.
18              }
19          }
20          texture.SetPixels (colourMap); //applies the colors to the texture
21          texture.Apply ();
22
23          textureRender.sharedMaterial.mainTexture = texture; // applies the texture to the texture renderer, without entering gamemore
24          textureRender.transform.localScale = new Vector3 (width, 1, height); // set the size of the plane to the size of the map
25      }
26  }
```

Editor

This editor creates a GUI (Graphical User Interface) to make changing values in the script easier. Firstly, I tell unity that this is an editor *(line 6).* Then I reference the map generator, so the script knows what it will be editing *(line 8).* Next, I added a button to the GUI, so that whenever it is clicked, the Map is displayed onto the plane.

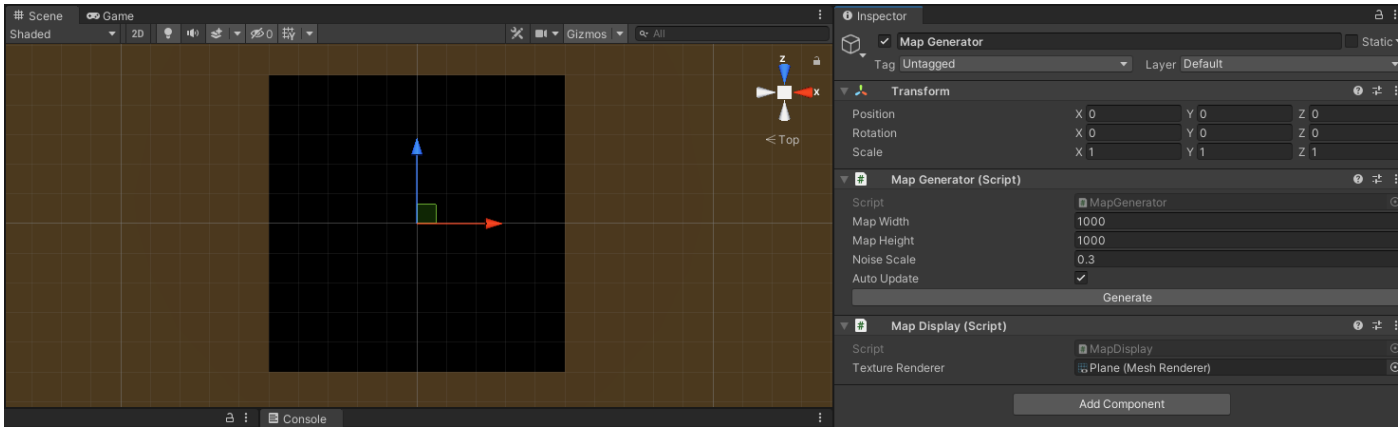
I made an error where I did not add the line **`[CustomEditor (typeof (MapGenerator))]`**, which confused unity as I did not tell unity that this was a custom editor, so the GUI didn't appear on the screen. *(Line 5)*

Secondly, I added a function that allows you to change the values from the GUI without having to click 'Generate' every time *(Line 12-16).* To do this, I also had to define a new variable (**`public bool autoUpdate;`**) in the map Generator.

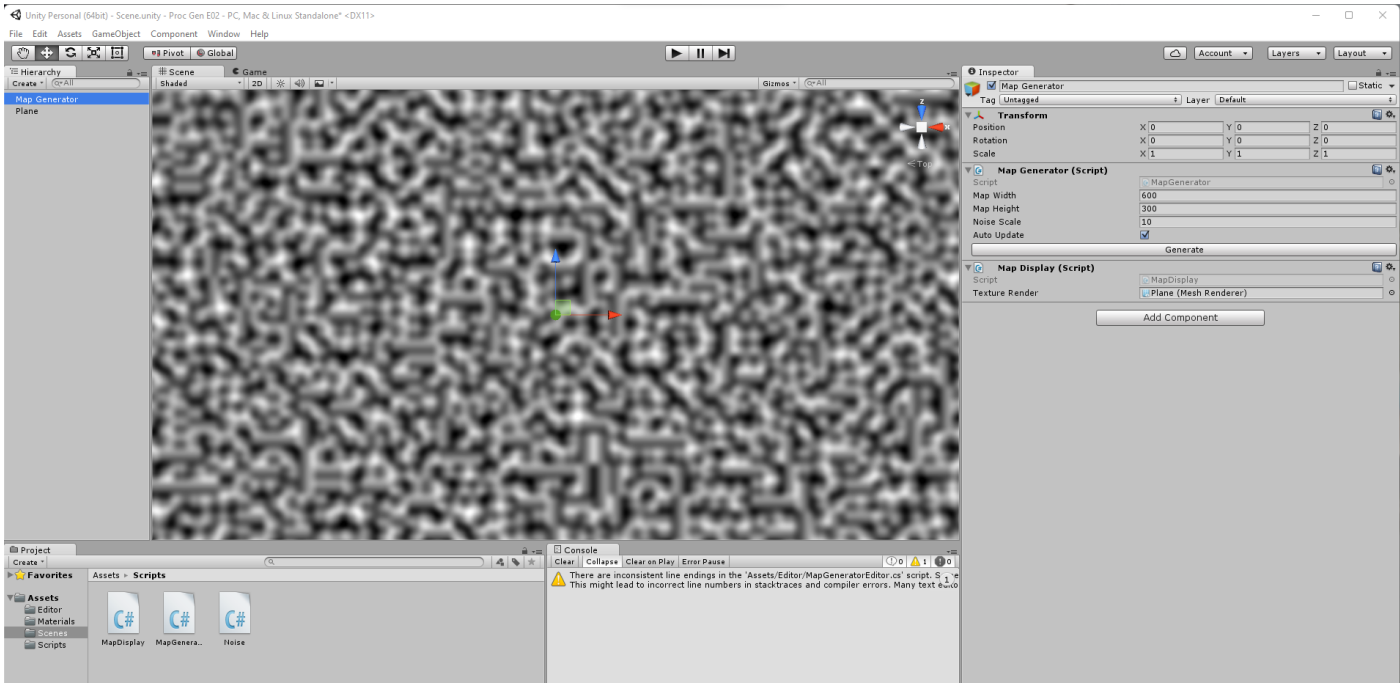
```
1   using UnityEngine;
2   using System.Collections;
3   using UnityEditor;
4
5   [CustomEditor (typeof (MapGenerator))] //says that it is a custom editor for the Map Generator
        Unity Script | 0 references
6   public class MapGeneratorEditor : Editor {
7
        0 references
8       public override void OnInspectorGUI() {
9           MapGenerator mapGen = (MapGenerator)target; // getting a reference to map generator (target, the object the customer editor is editing)
10
11          //DrawDefaultInspector ();
12          if (DrawDefaultInspector ()) { //if any value was changed, it will auto update the map (so when the scale was changed so will the pam without having to repress generate
13              if (mapGen.autoUpdate) {
14                  mapGen.GenerateMap ();
15              }
16          }
17
18          if (GUILayout.Button ("Generate")) { //adds a button, if pressed generates the map
19              mapGen.GenerateMap ();
20          }
21      }
22  }
```

Black Screen Error:

A major error in my project so far was that the Perlin noise function **`(float sample = Mathf.PerlinNoise(xCoord, yCoord);)`** does not work with the code I had typed. One fix was to re-read the program to fit the 2020.3 version. However, after a lot of re-reading and debugging, a quicker and easier solution was to simply switch to the version where this function was not changed. At first, I was hesitant to do this, as I thought an older version of Unity would lead to worse performance. But, after some research other software and game developers also prefer older versions of unity. As even though the new version has some benefits, the older versions have some utility that don't work the same in the new version (such as the function mentioned above).



Part 1 Final Product



The purpose of this step was to create and display the basic noise map, as well as add an easy-to-use GUI to manage this project further on. My next step will be to assign the pixel values to colours other than zero to get a more accurate render.

On the left is the objects in the scene. So far, I have implemented a plane, with the map generator pasted on top. In the centre, is the scene where I am displaying the Perlin noise. Currently the scale is at ten. A higher scale blends the pixels together to create large blurry shapes, whereas lowering it spreads them out and creates a visual remarkably similar of static on a TV. Going forward, I will use a higher scale. Next, on the right is the GUI, where I can edit the map's width and height as well as the scale. I can also tick the 'auto update' box to turn on the function I implemented earlier. Finally, the bottom shows the folders where I hold the main scripts, the editor script as well as the material used to display the noise map.

Part 2: Creating Fractal Perlin noise

True Perlin Noise

So far, I have coded a simple X noise generator. This is a simplified version of Perlin noise without values such as Octaves, lacunarity and persistence. The next step is to code those values in to turn my noise generator into a Perlin noise generator.

Changes to the Noise script

```
public static class Noise {

    // defines new octave, persistance, lacunarity and seed values
    public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, float scale, int octaves, float persistance, float lacunarity, int seed, Vector2 offset) {
        float[,] noiseMap = new float[mapWidth,mapHeight];

        // creates a pseudorandom number which is = to the seed
        System.Random prng = new System.Random(seed);
        Vector2[] octaveOffsets = new Vector2[octaves]; //each octave is sampled from a different location from an array
        for (int i = 0; i < octaves; i++)
        {
            float offsetX = prng.Next(-100000, 100000) + offset.x; //can't be too high or it will return the same value
            float offsetY = prng.Next(-100000, 100000) + offset.y;
            octaveOffsets[i] = new Vector2(offsetX, offsetY);
        }

        if (scale <= 0) {
            scale = 0.0001f;
        }

        // This normalises the noise map (by keep its values in the range of 0-1 by keeping track of the values)
        float maxNoiseHeight = float.MinValue;
        float minNoiseHeight = float.MaxValue;

        for (int y = 0; y < mapHeight; y++) {
            for (int x = 0; x < mapWidth; x++) {

                //Frequency and amplitude variables
                float amplitude = 1;
                float frequency = 1;
                float noiseHeight = 0;  // keeps track of the current hight value

                // Loops through the octaves
                for (int i = 0; i < octaves; i++)
                {
                    float sampleX = x / scale * frequency + octaveOffsets[i].x; //the higher the frequency = furether aprt the sample points.
                    float sampleY = y / scale * frequency + octaveOffsets[i].y;  //This means that hight values will change quicker

                    float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;    // This allows us to get better noise values as they can no be -1 - 1 instead of 0 - 1
                    noiseHeight += perlinValue * amplitude; //instead of noise map = perlinvalue, i will increase the noise hight by the perlin value of each ovtave

                    amplitude *= persistance; //amplitude increases each octave
                    frequency *= lacunarity;    // frequency increase each octave
                }
                //if current noise height > then current max noise, the we will update the max noise height to be = to current noise height (same with Noise height < min noise height)
                if (noiseHeight > maxNoiseHeight)
                {
                    maxNoiseHeight = noiseHeight;
                } else if (noiseHeight < minNoiseHeight)
                {
                    minNoiseHeight = noiseHeight;
                }

                // applies noise height to noise map
                noiseMap[x, y] = noiseHeight;

            }
        }
        // Once the range of noise is calculated, we loop through all the values
        // InverseLerp will give us a value between 0 and 1 ("Loops through inverselerp function which is a ... for example in the context of our code, it would...)
        for (int y = 0; y < mapHeight; y++) {
            for (int x = 0; x < mapWidth; x++) {
                noiseMap[x, y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight, noiseMap[x, y]);
            }
        }

        return noiseMap;
    }

}
```

The first step to creating true Perlin noise was to introduce the values that differentiate this to simple X or regular noise, such as octaves, persistence and lacunarity. Later, I also introduced the variable for the seed, but I will come back to that later. The next step was to define the amplitude, frequency, and noise height value. For the octaves, I used a method to sample it at various positions, using a random number generator, between the values of -100,000 to 100,000. This is because while testing out different numbers, a number too small caused the Perlin noise to look repetitive and a number too large, caused errors. The number I found was a good middle ground. Using the frequency, I then coded two lines that

change the height values quicker when the code is being run, creating a better-looking height map and lowering any lag. I then ensured that the height values could never be lower than 0 (line 42), giving us a better range of noise height values. After calculating the range of values, the nest step was to loop through them (line 64). The 'Mathf.InverseLerp' determines where a value lies between two points, in our context between 0 and 1. Lastly to avoid any errors I ensured that the current noise height could never be higher than the max noise height (Line 48).

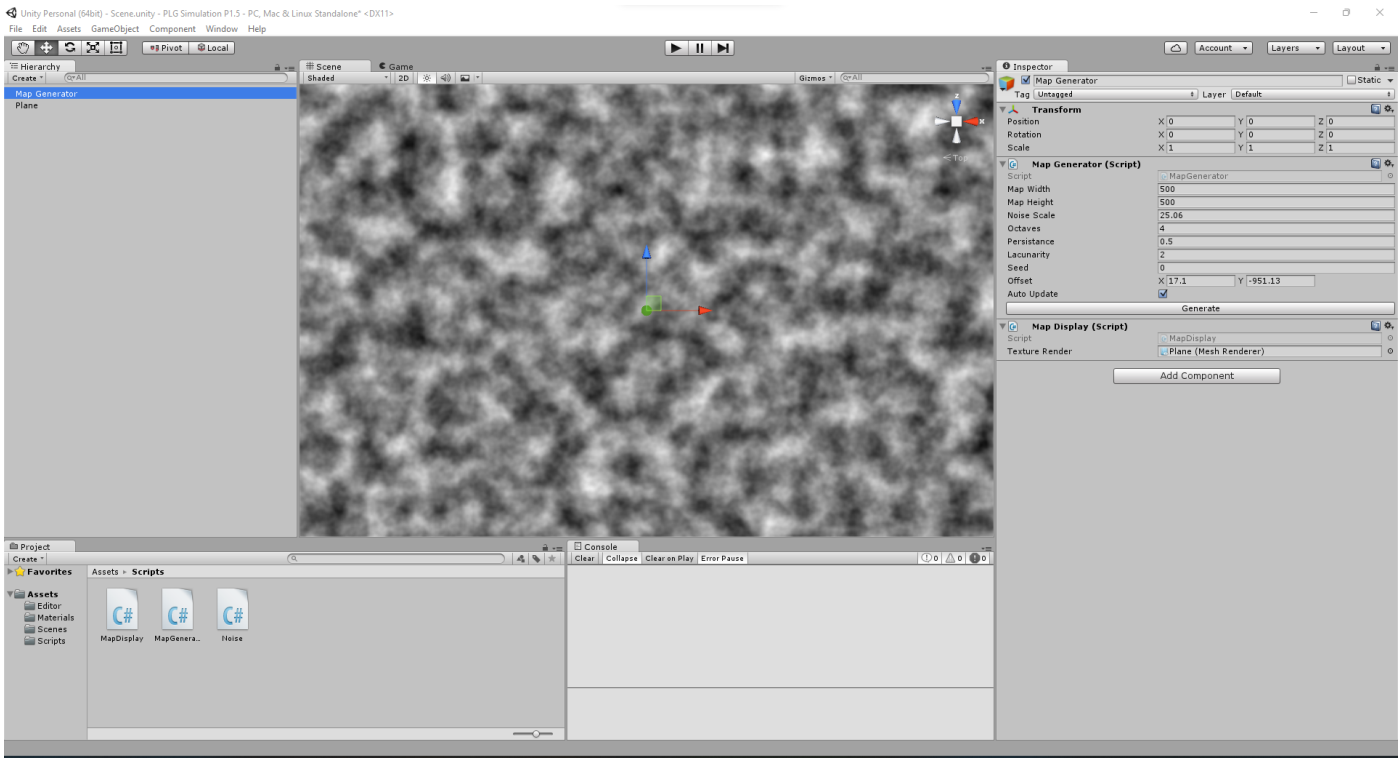
The seed is a set of numbers that is associated with a specific formation of procedurally generated Perlin noise. The purpose of this number is to be able to re-enter it to get the same formation or change it to get a completely random new one. Here I created a pseudo random number generator, using System.Random PRNG (line 9). This is a pseudo random number as true randomness is not possible on ordinary hardware and requires very advanced machinery for it such as quantum computers.

Finally, I created an array for each octave to be sampled from various locations (Line 10)

Changes to the map generator script

```
1   using UnityEngine;
2   using System.Collections;
3
    Unity Script | 3 references
4   public class MapGenerator : MonoBehaviour
5   {
6
7       // values defining the map (now including octaves, persistance, lacunarity)
8       public int mapWidth;
9       public int mapHeight;
10      public float noiseScale;
11
12      public int octaves;
13      public float persistance;
14      public int lacunarity;
15
16      public int seed;
17      public Vector2 offset;
18
19      public bool autoUpdate;
20
        2 references
21      public void GenerateMap()
22      {
23          //added the new values to this function so that they can be fetched from the Noise script)
24          float[,] noiseMap = Noise.GenerateNoiseMap(mapWidth, mapHeight, noiseScale, octaves, persistance, lacunarity, seed, offset);
25
26          MapDisplay display = FindObjectOfType<MapDisplay>();
27          display.DrawNoiseMap(noiseMap);
28      }
29
30  }
```

In this code I implemented the values stated above as well as creating a vector 2 variable for the offset. This is because vector 2 is for variables using 2 axes, in this case the X and Y.



This is what I've accomplished from those changes, The new values have been implemented into the editor, as well as the offset allowing us to travel across the noise, but most importantly, we have transformed out Simple X noise into Perlin noise.

Part 3: Creating a colour map

Implementing colours into my simulator

In this step, I will be implementing colours into my simulation, finally turning it into a realistic 2D terrain simulation. The way I will tackle this is by assigning the height map values to different colours based on the height of it geographically. For example, Height value of 0 will represent the deep ocean, whereas height value of 1 will be snow at the peak of a mountain. The design will be a pixeled/blocky texture, as that is not only easier to work with but will make the colours easier to differentiate as there won't be any gradient in between the pixels.

More changes to the Map Generator script

```
using UnityEngine;
using System.Collections;

Unity Script | 3 references
public class MapGenerator : MonoBehaviour
{
    3 references
    public enum DrawMode { NoiseMap, ColorMap};
    public DrawMode drawMode; // create a new toggleable variable for the draw mode to swap between the color and height map without overwriting the code

    [HideInInspector] public int mapWidth;
    [HideInInspector] public int mapHeight;
    public float noiseScale;

    public int octaves;
    public float persistance;
    public int lacunarity;

    public int seed;
    [HideInInspector] public Vector2 offset;

    public bool autoUpdate;

    public TerrainTypes[] Biomes; //adds the biome, color and heightmap it represents in the editor

    2 references
    public void GenerateMap()
    {
        float[,] noiseMap = Noise.GenerateNoiseMap(mapWidth, mapHeight, noiseScale, octaves, persistance, lacunarity, seed, offset);

        Color[] colorMap = new Color[mapHeight * mapHeight]; // creates a new map called color map

        for (int y=0; y<mapHeight; y++) // this loops through the noise to find the height value
        {
            for (int x=0; x<mapWidth; x++)
            {
                float height = noiseMap[x, y];  // once the height value has been found it associates to the biome
                for (int i=0; i < Biomes.Length; i++)
                {
                    if (height <= Biomes [i].height)
                    {
                        colorMap[y * mapWidth + x] = Biomes [i].color; //saves the colour for that point
                        break;
                    }
                }
            }
        }
        MapDisplay display = FindObjectOfType<MapDisplay>(); //toggles the draw mode
        if (drawMode == DrawMode.NoiseMap)
        {
            display.DrawTexture(TextureGenerator.TextureFromHeightMap(noiseMap)); //draws the differnt types of maps
        }
        else if (drawMode == DrawMode.ColorMap)
        {
            display.DrawTexture(TextureGenerator.TextureFromColorMap(colorMap, mapWidth, mapHeight));
        }
    }
}

[System.Serializable]
1 reference
public struct TerrainTypes  // assign 0-1 values to colors
{
    public float height;    // float for heihgt value
    public Color color;     // float for color value
    public string name;     // Type of terrain
}
```

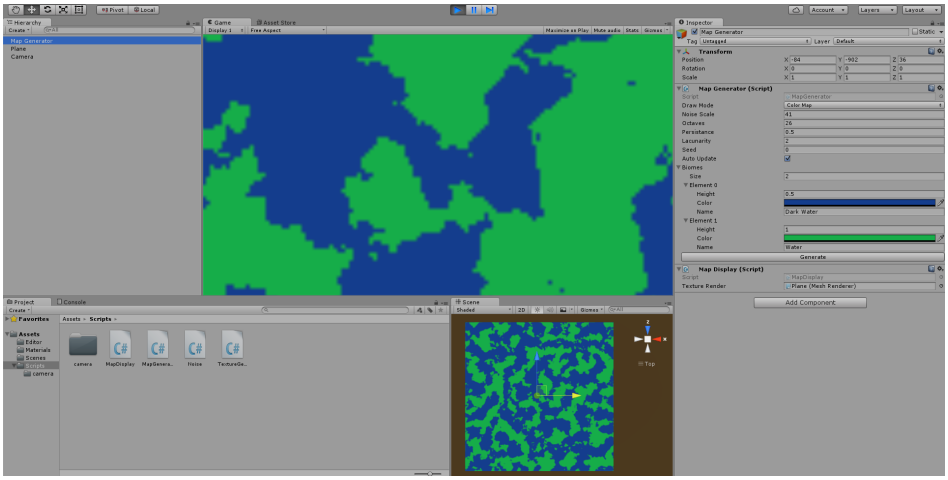
I decided to start by creating a public structure called 'Terrain Types', where the height level will be assigned a colour and a name, which is then added in the editor. However, this doesn't change the colour yet. To do this I first need to separate the

colour map and the noise map, as I don't want to overwrite what I did previously, so I created a toggleable variable called draw mode to switch between the height map and the colour map. For the colour map, I created a loop that gets all the values from the heights map and assigns them to the biomes with the corresponding height value and colour (Line 34). Finally, it saves the colour at those points and goes for the next biome.

Texture Generator script

```
using UnityEngine;
using System.Collections;

2 references
public static class TextureGenerator //seperated the texture generation code to make it cleaner and easier to work with
{

    2 references
    public static Texture2D TextureFromColorMap(Color[] colourMap, int width, int height) //creates a texture from the color map
    {
        Texture2D texture = new Texture2D(width, height);
        texture.SetPixels(colourMap);
        texture.Apply();
        return texture;
    }


    1 reference
    public static Texture2D TextureFromHeightMap(float[,] heightMap) //this method gets a texture from a 2D heihght map
    {
        int width = heightMap.GetLength(0); // this is the code used to create the height map before
        int height = heightMap.GetLength(1);

        Color[] colourMap = new Color[width * height];
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                colourMap[y * width + x] = Color.Lerp(Color.black, Color.white, heightMap[x, y]);
            }
        }

        return TextureFromColorMap(colourMap, width, height); //instead of creating the texture above, we return the results from the methods above, this is quicker
    }

}
```

Here I moved the code used to create the noise map into this script, so that I can make it work better with the colour map and to avoid confusion. I first have a 2D class for the colour map, this simply created a new texture using the colour map function from before. Then the Height map is the same code from the map generator, however instead of creating a new texture every time there is a change, the results are returned, speeding up the process and hopefully improving load times.

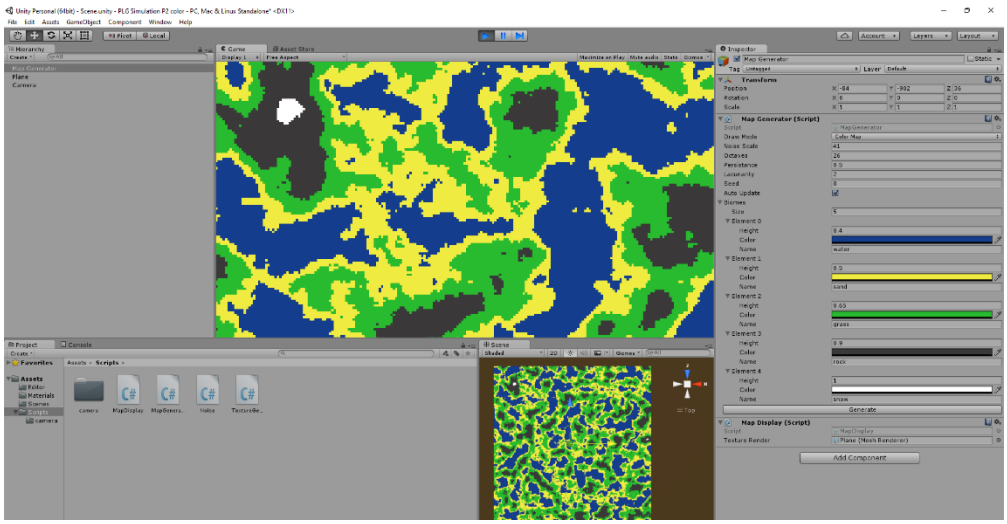


Now that we have the colours working, there are 2 issues. One of them being the blurry and blotchy appearance of the colour map. To fix this, I simply needed to write some code in the texture generator

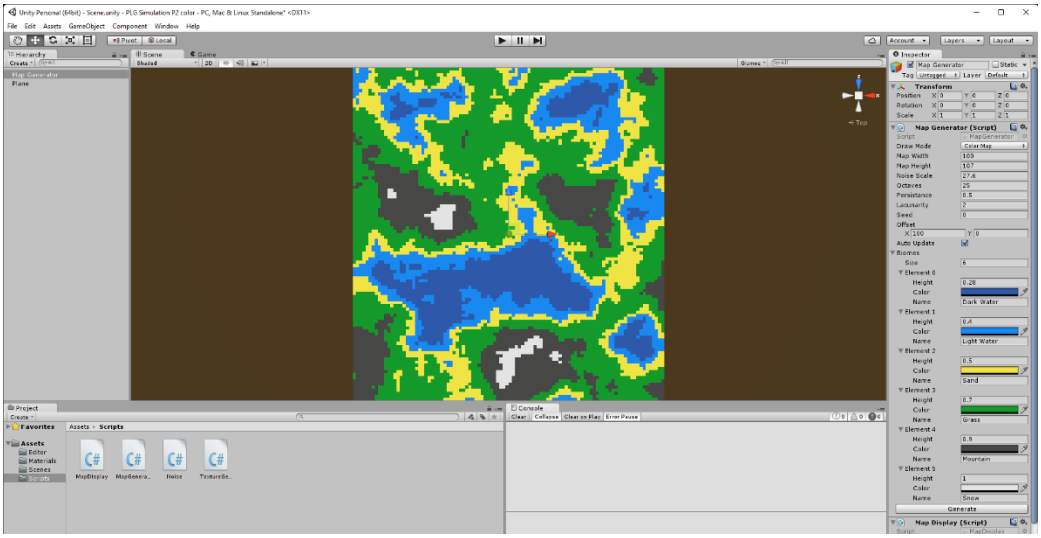
```
texture.filterMode = FilterMode.Point; //creates the pixelated texture
texture.wrapMode = TextureWrapMode.Clamp; //removes bluriness
```

One of these clumps the pixels together and the other one gives them more definition.

The next step was to get better colour gradients. And I went through many iterations for this. The first issue is that I only have 2 colours, so I added sand, mountains, and snow



However, this still did not yet look suitable due to the disproportionate amount of sand. Secondly, the water was too bland and needed more gradient. This led me to my third iteration.



I am quite happy with this as the ocean has a colour gradient and the sand is proportional to the rest of the terrain.

Part 4: Making the program user friendly

Moving around the scene

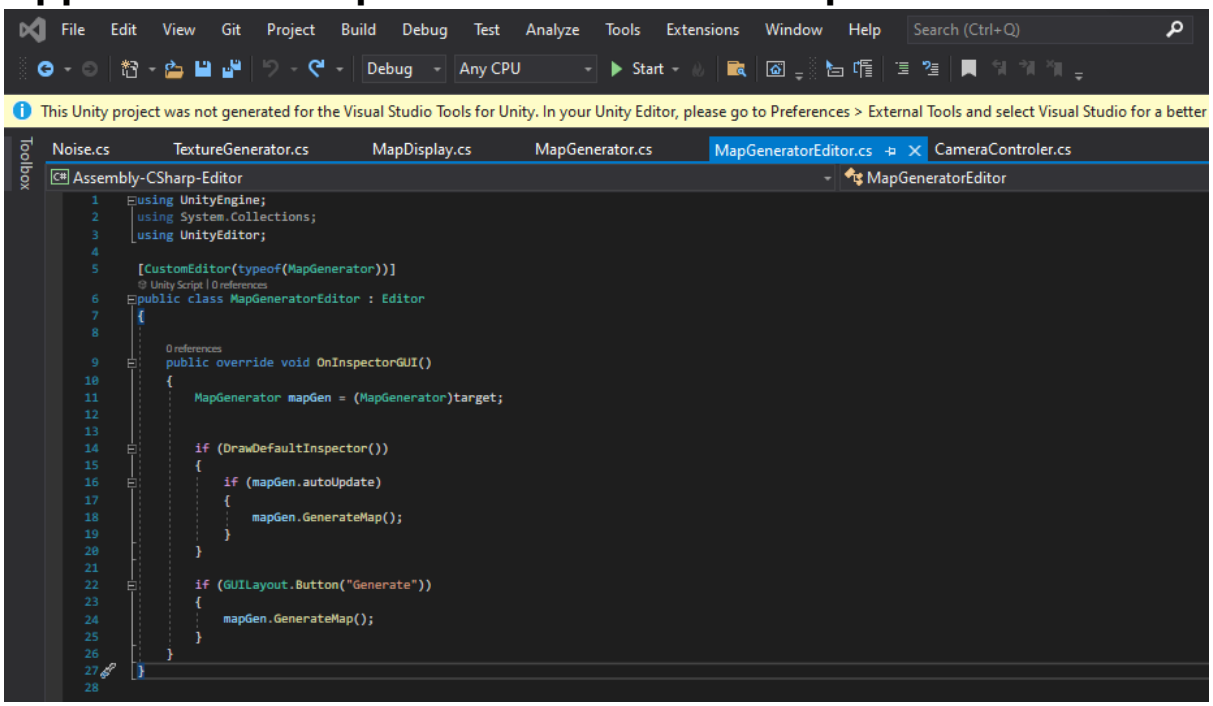
Now that the main code itself is complete, I need to create a script to control the camera movement. I went with a top-down view control, or a bird's eye view as that worked best with the 2D map. Secondly, as this simulation isn't infinite, I combined 2 techniques to create the feeling that it is infinite by making sure the camera can't see outside the terrain. The first method is to create a border where the camera cannot travel further, limiting its view so it can only see the terrain and not the empty scene around it. I will also make it so that when you reach the end of the border, the camera is instantly transported back to the centre. Secondly, I will attempt to increase the size of the terrain to the maximum without causing issues with my computer limited hardware.

```
using UnityEngine;

Unity Script | 0 references
public class CameraControler : MonoBehaviour {

    public float CameraSpeed = 20f;
    public float panBoarderThickness = 10f;
    public Vector2 panLimit;

    Unity Message | 0 references
    void Update () {

        Vector3 pos = transform.position;

        if (Input.GetKey(KeyCode.UpArrow))
        {
            pos.z += CameraSpeed * Time.deltaTime;
        }
        if (Input.GetKey(KeyCode.DownArrow))
        {
            pos.z -= CameraSpeed * Time.deltaTime;
        }
        if (Input.GetKey(KeyCode.RightArrow))
        {
            pos.x += CameraSpeed * Time.deltaTime;
        }
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            pos.x -= CameraSpeed * Time.deltaTime;
        }

        if (pos.x == panLimit.x)
        {
            pos.x = 0;
        }
        if (pos.x == -panLimit.x)
        {
            pos.x = 0;
        }
        if (pos.z == panLimit.y)
        {
            pos.z = 0;
        }
        if (pos.z == -panLimit.x)
        {
            pos.z = 0;
        }

        pos.x = Mathf.Clamp(pos.x, -panLimit.x, panLimit.x);
        pos.z = Mathf.Clamp(pos.z, -panLimit.y, panLimit.y);

        transform.position = pos;


    }
}
```

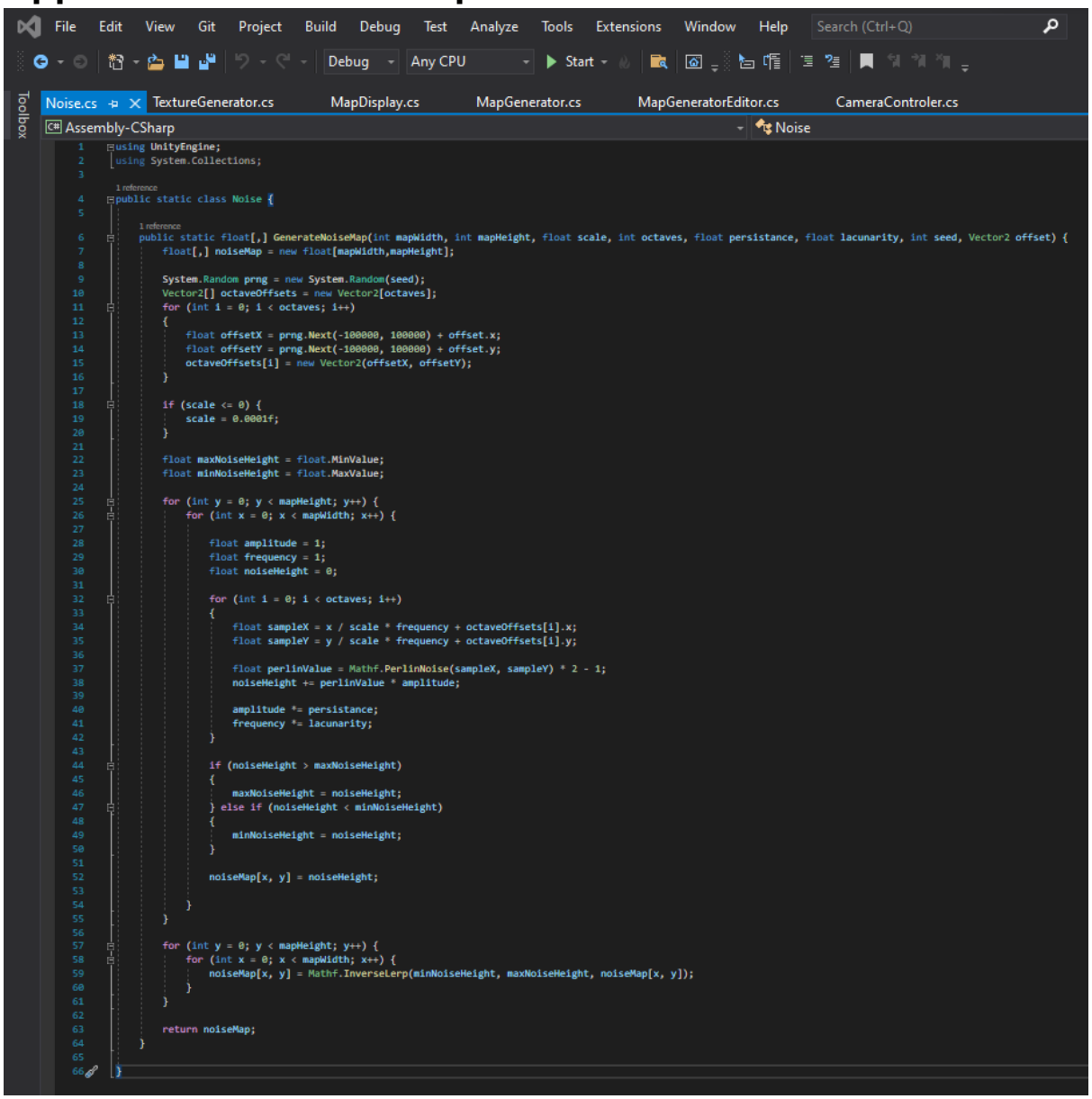
The first step was to create some variables for different aspects of the camera control. The first one being the camera speed. I later did some changes to this to get the perfect value, so that you don't travel so fast you miss everything but quick enough, so the speed isn't tedious. I then created a loop to change the camera position using the arrow keys. This was done with the input.Getkey function. For the camera border, I created the variable for the boarder thickness, as well as the 2D vector for the X and Y axis of the border size. I then create 4 'if' loops so that if the X or Y camera position = to the X or Y border position, then it would displace the camera back to the centre.

Appendix 3: Full C# Code

Appendix 2.1: Map Generator Editor Script

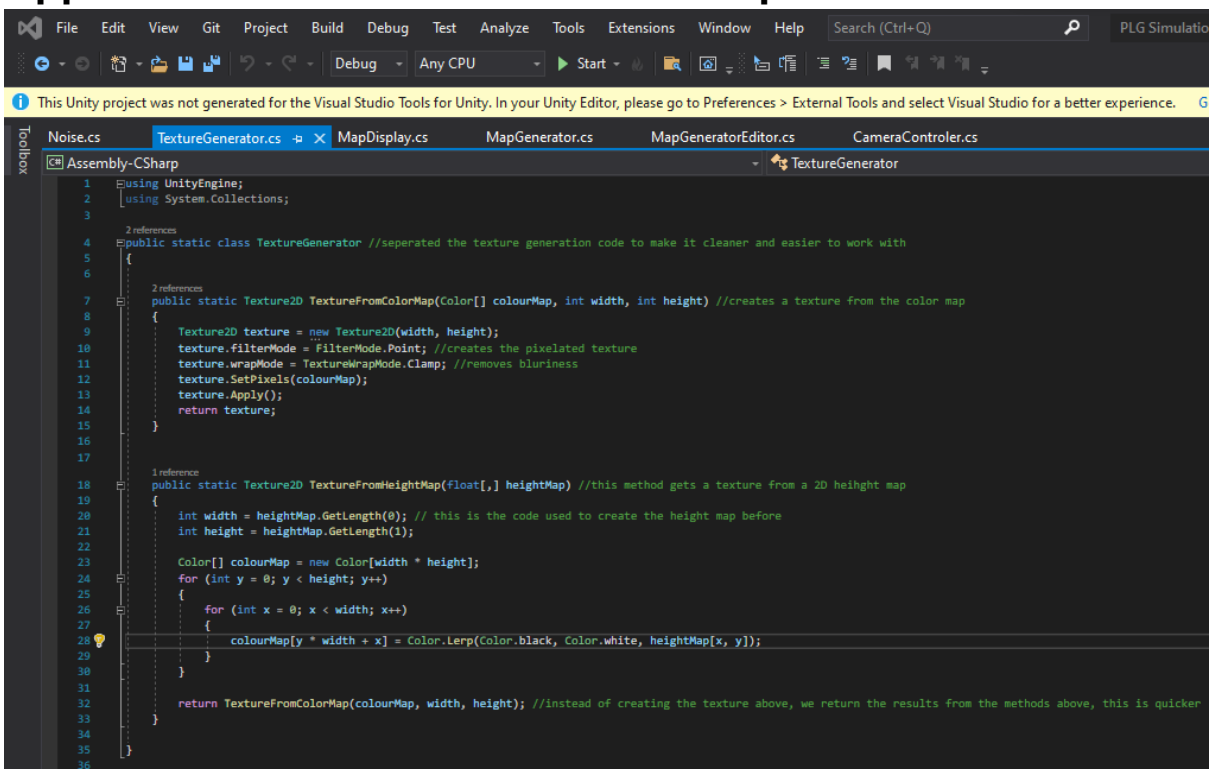


Appendix 2.2: Noise Script



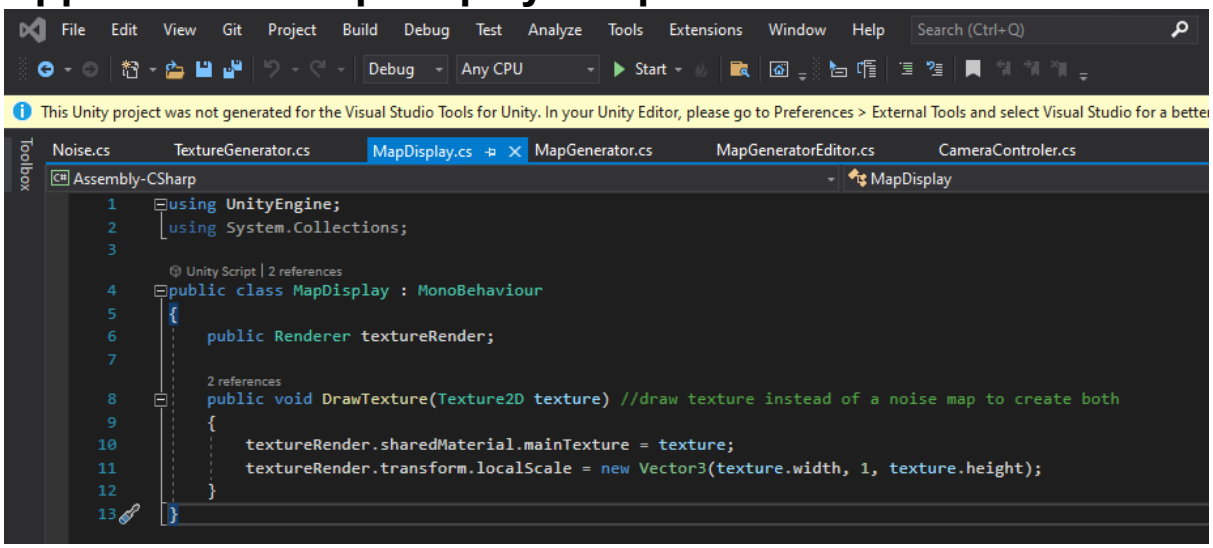

```
using UnityEngine;
using System.Collections;

1 reference
public static class Noise {

    1 reference
    public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, float scale, int octaves, float persistance, float lacunarity, int seed, Vector2 offset) {
        float[,] noiseMap = new float[mapWidth,mapHeight];

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

        if (scale <= 0) {
            scale = 0.0001f;
        }

        float maxNoiseHeight = float.MinValue;
        float minNoiseHeight = float.MaxValue;

        for (int y = 0; y < mapHeight; y++) {
            for (int x = 0; x < mapWidth; x++) {

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

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

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

                    amplitude *= persistance;
                    frequency *= lacunarity;
                }

                if (noiseHeight > maxNoiseHeight)
                {
                    maxNoiseHeight = noiseHeight;
                } else if (noiseHeight < minNoiseHeight)
                {
                    minNoiseHeight = noiseHeight;
                }

                noiseMap[x, y] = noiseHeight;

            }
        }

        for (int y = 0; y < mapHeight; y++) {
            for (int x = 0; x < mapWidth; x++) {
                noiseMap[x, y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight, noiseMap[x, y]);
            }
        }

        return noiseMap;
    }
}
```

Appendix 2.3: Texture Generator Script



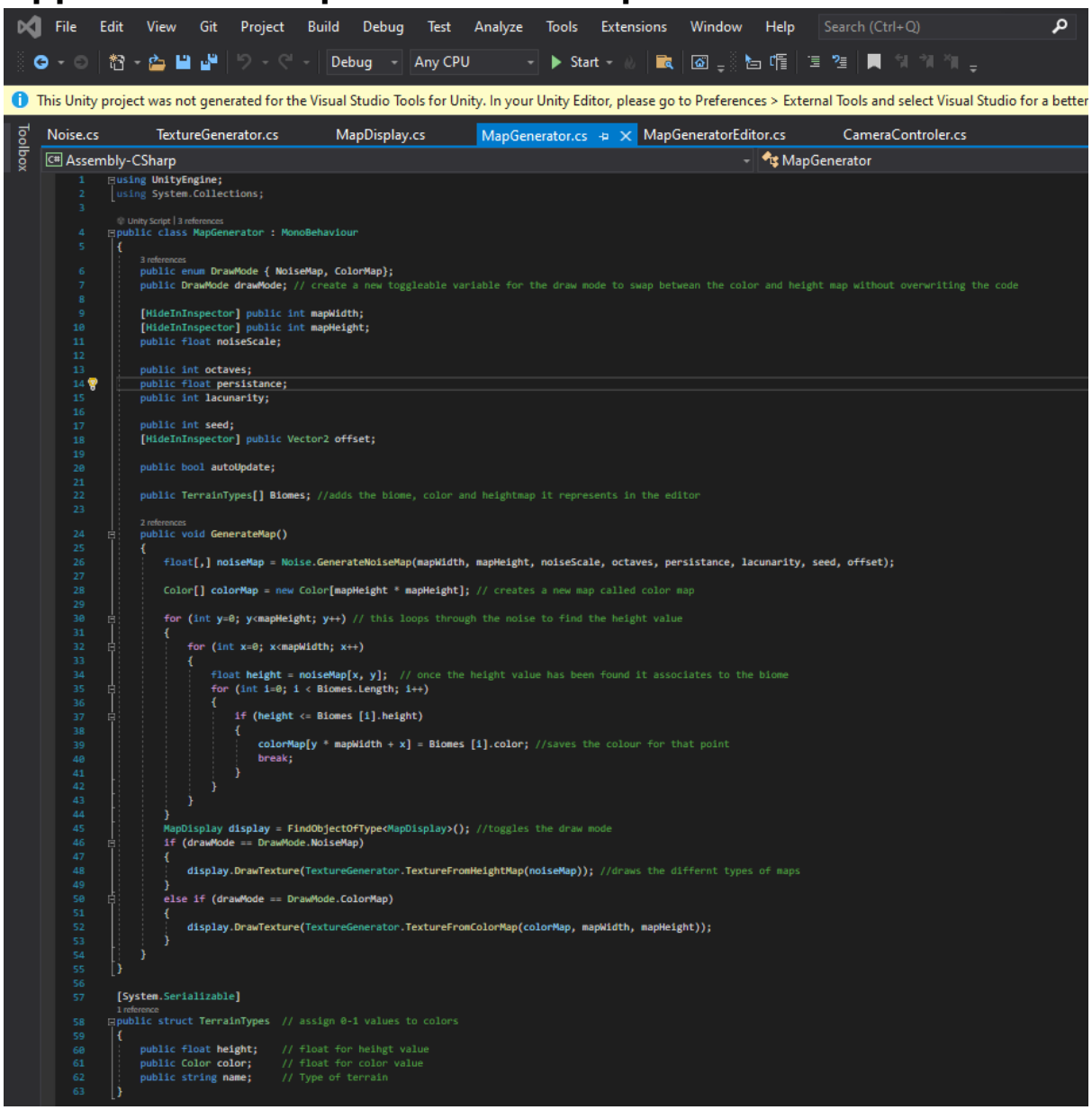
```
using UnityEngine;
using System.Collections;

2 references
public static class TextureGenerator //seperated the texture generation code to make it cleaner and easier to work with
{

    2 references
    public static Texture2D TextureFromColorMap(Color[] colourMap, int width, int height) //creates a texture from the color map
    {
        Texture2D texture = new Texture2D(width, height);
        texture.filterMode = FilterMode.Point; //creates the pixelated texture
        texture.wrapMode = TextureWrapMode.Clamp; //removes bluriness
        texture.SetPixels(colourMap);
        texture.Apply();
        return texture;
    }


    1 reference
    public static Texture2D TextureFromHeightMap(float[,] heightMap) //this method gets a texture from a 2D heihght map
    {
        int width = heightMap.GetLength(0); // this is the code used to create the height map before
        int height = heightMap.GetLength(1);

        Color[] colourMap = new Color[width * height];
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                colourMap[y * width + x] = Color.Lerp(Color.black, Color.white, heightMap[x, y]);
            }
        }

        return TextureFromColorMap(colourMap, width, height); //instead of creating the texture above, we return the results from the methods above, this is quicker
    }
}
```

Appendix 2.4: Map Display Script

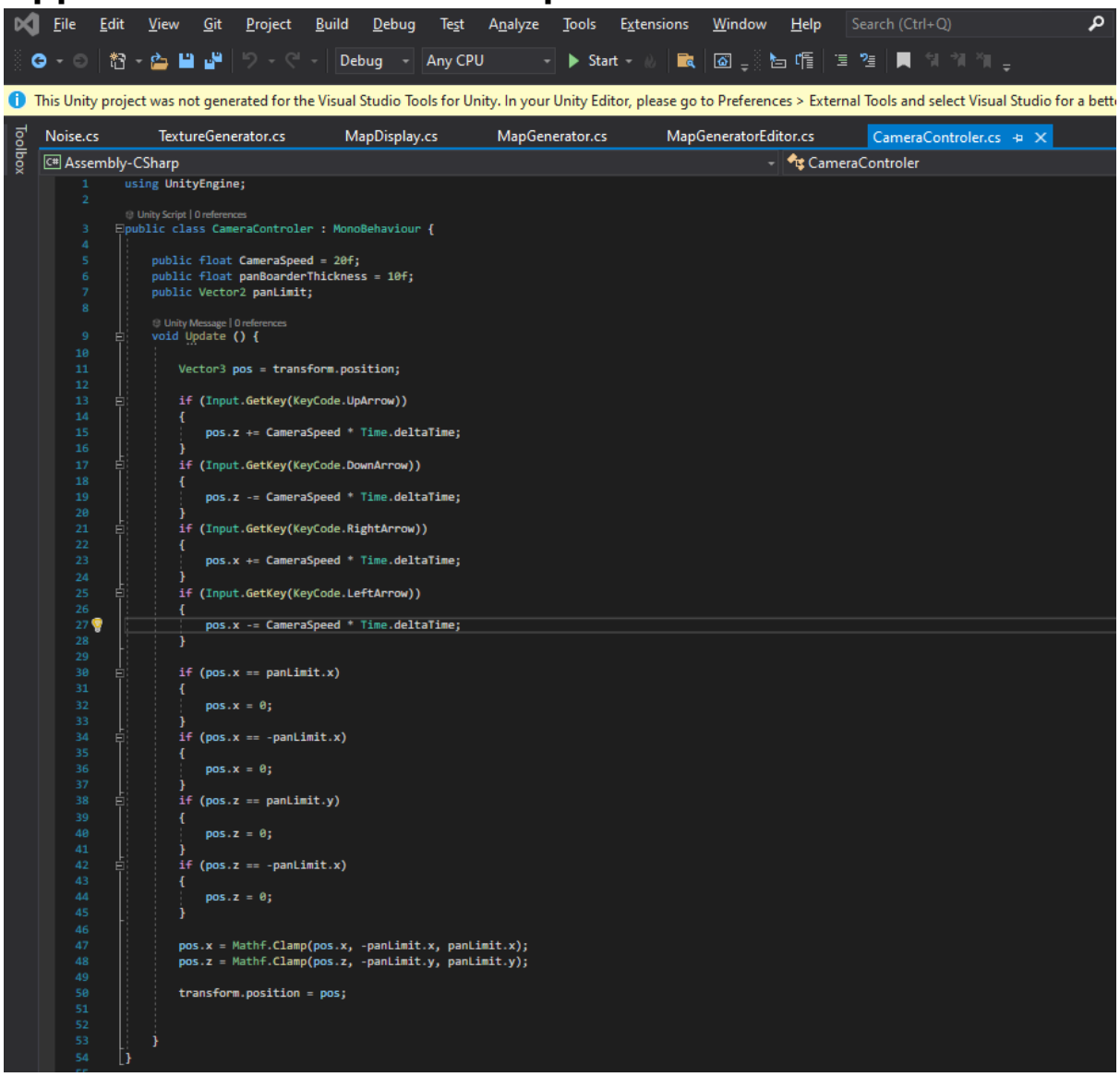


```
using UnityEngine;
using System.Collections;

Unity Script | 2 references
public class MapDisplay : MonoBehaviour
{
    public Renderer textureRender;


    2 references
    public void DrawTexture(Texture2D texture) //draw texture instead of a noise map to create both
    {
        textureRender.sharedMaterial.mainTexture = texture;
        textureRender.transform.localScale = new Vector3(texture.width, 1, texture.height);
    }
}
```

Appendix 2.5: Map Generator Script



Appendix 2.6: Controller script



Appendix 3: User Acceptance Testing Sheets

User Acceptance Testing

User ID	Tested by	Date tested
001	F. TOURNIER	30/07/22

Success criteria:

- ☒ Does the code generate terrain and does it appear to be random/procedurally generated? Yes
- ☒ Does the generated terrain show geological forms like changes in elevation and bodies of water? Yes
- ☒ Do different seeds equal different terrains, and do the same seeds generate the same terrain? Yes
- ☒ Do the terrains design/texture change depending on height level to match its real-world representation (e.g., tall, generated mountains will have snow at peaks, and deep oceans will have a darker colour)? Yes - with one comment
- ☒ Is the terrain open to explore? Yes

If any criteria were not met, please state why. If they were met, then leave blank.

Criteria number	Reason for unsuccess
1)	
2)	
3)	
4)	See recommendation #3 below for further improvement and a more realistic rendering
5)	

Bugs/Errors:

If you notice any bugs or errors in your testing that are unrelated to the criteria, please comment them bellow (if possible, please leave screen shots)

1) • During Testing, the game window and the scene window seem to be inverted → North is South, and South is North
2) • At the boundaries of the map, terrain is greyed out, whereas it should loop back to the beginning of generated terrain.
3) • The preset values for biome are rendering a disproportionate amount of sand/rocks compared to an environment like the earth (where there are alot of oceans).
I would suggest to study the height repartition on earth (from Marians Trench - 10km to Mt Everest +8km), and the corresponding biome ranges. Use the resulting values for a more realistic rendering (according to earthly standards).

User Acceptance Testing

User ID	Tested by	Date tested
002	EVE TOURNIER	31st July 2022

Success criteria:

- ☑ Does the code generate terrain, and ~~does it is~~ is it random/procedurally generated?
- ☑ Does the generated terrain show geological forms like changes in elevation and bodies of water?
- ☑ Do different seeds equal different terrains, and do the same seeds generate the same terrain?
- ☑ Do the terrains design/texture change depending on height level to match its real-world representation (e.g., tall, generated mountains will have snow at peaks, and deep oceans will have a darker colour)?
- ☑ Is the terrain open to explore?

If any criteria were not met, please state why. If they were met, then leave blank.

Criteria number	Reason for unsuccess
1)	
2)	
3)	
4)	
5)	

Bugs/Errors:

If you notice any bugs or errors in your testing that are unrelated to the criteria, please comment them bellow (if possible, please leave screen shots)

- Everything worked really well and 'how to use it' was well explained.
- Only slight feedback is
  - it was difficult to find the 'generate button', indicate its location (bottom right corner) for the next user.
  - it was not obvious to me what the different variables did, but reading the report I will better understand it.

User Acceptance Testing

User ID	Tested by	Date tested
003	Ramtin Lakani	21/10/2022

Success criteria:

- ☑ Does the code generate terrain, and does it is random/procedurally generated?
- ☑ Does the generated terrain show geological forms like changes in elevation and bodies of water?
- ☑ Do different seeds equal different terrains, and do the same seeds generate the same terrain?
- ☑ Do the terrains design/texture change depending on height level to match its real-world representation (e.g., tall, generated mountains will have snow at peaks, and deep oceans will have a darker colour)?
- ☑ Is the terrain open to explore?

If any criteria were not met, please state why. If they were met, then leave blank.

Criteria number	Reason for unsuccess
1)	
2)	
3)	
4)	
5)	

Bugs/Errors:

If you notice any bugs or errors in your testing that are unrelated to the criteria, please comment them bellow (if possible, please leave screen shots)

- No issues game worked as intended

User Acceptance Testing

User ID	Tested by	Date tested
005	Shaan	18/11/22

Success criteria:

- ☑ Does the code generate terrain, and does it is random/procedurally generated?
- ☑ Does the generated terrain show geological forms like changes in elevation and bodies of water?
- ☑ Do different seeds equal different terrains, and do the same seeds generate the same terrain?
- ☑ Do the terrains design/texture change depending on height level to match its real-world representation (e.g., tall, generated mountains will have snow at peaks, and deep oceans will have a darker colour)?
- ☑ Is the terrain open to explore?

If any criteria were not met, please state why. If they were met, then leave blank.

Criteria number	Reason for unsuccess
1)	
2)	
3)	
4)	
5)	

Bugs/Errors:

If you notice any bugs or errors in your testing that are unrelated to the criteria, please comment them bellow (if possible, please leave screen shots)

No errors found

User Acceptance Testing

User ID	Tested by	Date tested
004	MR A Routine	Nov 4, 2022

Success criteria:

- [x] Does the code generate terrain, and does it is random/procedurally generated?
- [x] Does the generated terrain show geological forms like changes in elevation and bodies of water?
- [x] Do different seeds equal different terrains, and do the same seeds generate the same terrain?
- [x] Do the terrains design/texture change depending on height level to match its real-world representation (e.g., tall, generated mountains will have snow at peaks, and deep oceans will have a darker colour)?
- [x] Is the terrain open to explore?

If any criteria were not met, please state why. If they were met, then leave blank.

Criteria number	Reason for unsuccess
1)	
2)	
3)	
4)	
5)	

Bugs/Errors:

If you notice any bugs or errors in your testing that are unrelated to the criteria, please comment them bellow (if possible, please leave screen shots)

On regenerating of the map, the zoom level is reset, it would make more sense if it did not,

Bibliography

Anderson, S. (2015, May 3). *Building an Infinite Procedurally-Generated World*. Retrieved from Atomic object: https://spin.atomicobject.com/2015/05/03/infinite-procedurally-generated-world/

Benjamin Mark, T. B. (2020, July 10). *Procedural Generation of 3D Caves for Games on the GPU.* Retrieved from ITU Copenhagen, Lund University and NY University: https://portal.research.lu.se/portal/files/6067634/5464988.pdf

Chen, J. V. (2020, April 17). *Procedural Generation: creating 3D worlds with deep learning*. Retrieved from Massachusetts Institute of Technology: http://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html

Ferrone, H. (2021). *Learning C# by Developing Games with Unity 2021.* Packt.

Generating Complex Procedural Terrains Using the GPU. (2020). In R. Geiss, *GPU Gems 3* (p. Chapter 1). Nvidia.

Price, H. (2017, October 25). *Procedural Generation – Not Just For Games*. Retrieved from Red River: https://river.red/procedural-generation-not-just-games/

Vandrake. (2020, Febuary 24). *Procedural Generation in Game Development*. Retrieved from davidepesce.com: https://www.davidepesce.com/2020/02/24/procedural-generation-in-game-development/

Werner Gaisbauer, H. H. (2020, Febuary 15). *Procedural Generation for Populated Virtual Cities: A Survey.* Retrieved from university of Vienna : https://pdfs.semanticscholar.org/652d/04eea5dee15ff9f453ab0730c61bc0698d3a.pdf

Yi Xu, R. S. (2021, June 16). *Procedural generation of problems for elementary math education.* Retrieved from Delft University of Technology, The Netherlands & Futurewhiz, Amsterdam, The Netherlands: https://journal.seriousgamessociety.org/index.php/IJSG/article/view/396/421