



**T.C.
GEBZE TECHNICAL UNIVERSITY**

COMPUTER SCIENCE AND ENGINEERING

Procedural Terrain Generation

**Azmi Utku Sezgin
131044048**

**ADVISOR
Prof. Dr. F. Erdoğan SEVİLGİN**

**July, 2020
Gebze, KOCAELİ**

This project was approved by juries mentioned below on 04/03/2020 as Computer Science and Engineering Graduation Project

Graduation Project Jury

Advisor Name	Prof. Dr. Fatih Erdoğan SEVİLGEN	
University	Gebze Technical University	
Faculty	Computer Science and Engineering	

Jury Name	Assoc. Prof. Dr. Didem GÖZÜPEK	
University	Gebze Technical University	
Faculty	Computer Science and Engineering	

Jury Name	Assoc. Prof. Dr. Habil KALKAN	
University	Gebze Technical University	
Faculty	Computer Science and Engineering	

PREFACE

This project has been in my mind since the day I started this school along with my first projeoct Procedural Map Generation and I want to thank everyone who has contributed in this project and my sincere gratitude to Prof. Dr. F. Erdoğan SEVİLGİN for making this possible and guiding me throughout the whole semester and helping me become a better engineer.

June 2020

Azmi Utku Sezgin

TABLE OF CONTENT

PREFACE.....	IV
TABLE OF CONTEN.....	V
FIGURE LIST.....	VI
ABBREVIATION.....	VI
ABSTRACT.....	VII
1. INTRODUCTION	1
2. PROJECT DESIGN	2
2.1) GENERATING THE MESH.....	2
2.2) GENERATING THE TEXTURE.....	10
2.3) INFINITE GENERATION.....	16
3. PUTTING IT ALL TOGETHER.....	18
4. TESTS AND RESULTS.....	20
5. CONCLUSION.....	24
6. SUCCESS CRITERIA.....	24
7. REFERENCES.....	25

FIGURE LIST

Figure 2.0 4 Octaves where red is combination of blue ones.....	3
Figure 2.1 Sample Run 1.....	5
Figure 2.2 Sample Run 2.....	5
Figure 2.3 Sample Run 3.....	5
Figure 2.4 Sample Run 4.....	5
Figure 2.5 Sample Run 5.....	5
Figure 2.6 Sample Run 6.....	5
Figure 2.7 Sample Run 7.....	6
Figure 2.8 Sample Run 8.....	6
Figure 2.9 Triangulation of vertices.....	7
Figure 2.10 Animation Curve.....	8
Figure 2.11 Example run of mesh generation.....	8
Figure 2.12 Example run of mesh generation.....	9
Figure 2.13 Example run of mesh generation.....	9
Figure 2.14 Example run of mesh generation.....	10
Figure 2.15 Simplified biome chart.....	8
Fig 2.16 Whittaker Biome Diagram.....	9
Fig 2.17 Sample Moisture Map.....	12
Fig 2.18 Sample Moisture Map.....	12
Fig 2.19 Latitude's effect on temperature.....	13
Fig 2.20 Heat Map with Power 10.....	14
Fig 2.21 Heat Map with Power 5.....	15
Fig 2.22 Heat Map with Power 10.....	15

Fig 2.23 Before offset with a reference point.....	17
Fig 2.23 After offset with a reference point.....	18
Figure 3.0 Biomes in editor.....	19
Figure 3.1 After applying biomes.....	20
Figure 4.0 Parameters for the following runs with differen4 see.....	20
Figure 4.1 Sample Run	21
Figure 4.2 Sample Run	21
Figure 4.3 Sample Run	22
Figure 4.4 Sample Run	22
Figure 4.5 Sample Run	23

ABBREVIATION

RTS : Real Time Strategy

ABSTRACT

The genre of dungeon games, or first-person shooter games as they are more commonly known, has emerged over the last ten years to become one of the most popular types of computer game. At present, the levels in this type of game are generated manually, which is a very expensive and time-consuming process for games companies.

If levels could be generated automatically then this would not only reduce development costs, but allow levels to be generated at run-time, giving game players new playing experience each time a game was played and greatly improving the replayability of the game.

This project is a tool for generating random terrains for the games that are developed using Unity game engine. The main goal is to increase the replayability of the games and give the players new experience everytime they play the game and give the developers flexibility of playing with the parameters to generate terrains that are more suitable to their games.

1. INTRODUCTION

In the rapidly changing world of computer games, game players are becoming ever more demanding and new hardware technology is constantly pushing back the limits of what can be achieved in games. This has led software developers to produce games that are more complicated and with better graphics than ever before.

It would therefore be desirable if a system could be developed that can take parameters as a set of rules defining properties of the terrain that must be produced for a specific game, and automatically create levels for that game. This project provides a base line for procedurally generating terrains.

This report describes the design and implementation of generating controlled terrains with parameters.

2. PROJECT DESIGN

There are two components of terrain generation, mesh and the texture. The generation of mesh requires a height map which is a 2D array that contains numbers between 0 and 1 where each element represents the height of the mesh so the size of that array is equal to the size of the mesh. The other one is a set of vertices and triangles.

The texture requires biomes and set of rules to determine which parts of the mesh are gonna be populated by which biome. The rules are simplified as opposed to real world. The biomes are generated by utilizing heat map and moisture map. There are 4 levels of heat map(hottest, hot, cold, coldest) and 4 levels of moisture map(dryest, dry, wet, wettest) there are 16 combinations with 9 unique biomes. The heat map is generated from both distance from the middle of the map in -y axis and the height map. The moisture map is generated from the height map, the higher altitude the dryer land.

Last thing is the infinite generation of the terrain that can be used in the game in runtime that doesn't use any extra memory so that however long you want your level it can provide you that.

The main objective of this project is to create a base line for generating terrains with flexibility by exposing parameters so that the project can be integrated to games more easily.

2.1. GENERATING THE MESH

The first component that mesh generation requires is height map. Height map is usually generated by using noise algorithms.[1] I have used an adapted version of

perlin noise which is widely used in terrain generation.[4] The adapted version of perlin noise uses multiple octaves and 2 parameters called lacunarity and persistence. Lacunarity controls the increase of frequency in octaves and persistence controls the decrease in amplitude. Lacunarity is a number greater than 0 and persistence is a number between 0 and 1.

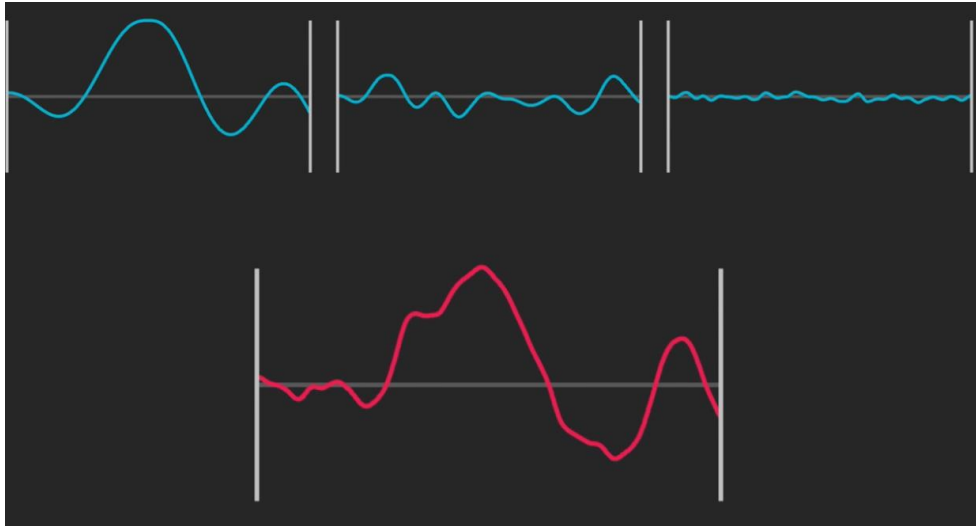


Figure 2.0 4 Octaves where red is combination of blue ones

In **Fig 2.0** there are four octaves where the red one is combination of three blue octaves. The octave on the left represents main outline like mountains whereas the octave on the right represents small things like sand or small rocks. To achieve that, frequency from left to right must increase and amplitude must decrease. The frequency in octaves is **lacunarityⁱ** and amplitude is **persistenceⁱ** where **i** is the octave number. If the lacunarity is 2, frequency from left to right is,

$$2^0 = 1,$$

$$2^1 = 2,$$

$$2^2 = 4$$

And if the persistence is 0.5, amplitude from left to right is,

$$0.5^0 = 1,$$

$$0.5^1 = 0.5,$$

$$0.5^2 = 0.25$$

These numbers are multiplied by adding the octaves. Since these octaves represent height values it also means adding different height values.[4]

The height map should contain numbers between 0 and 1 but the octaves and numbers that perlin noise generates must be between -1 and 1 so the height can increase or decrease. To normalize the values, I've used Unity's `Mathf.InverseLerp` method in order to map the values to be between 0 and 1. I've also used `Mathf.PerlinNoise` to generate the noise. The parameters for the `PerlinNoise` function are x and y coordinates for the samples. The higher frequency the further apart the sample points will be which means height values will change more rapidly. With every octave, frequency increases and persistence decreases. Here's the pseudocode for heightmap generation

Procedure `generateHeightMap`

```
For y from 0 to mapHeight begin
    For in x from 0 to mapWidth begin
        amplitude  $\leftarrow$  1
        Let frequency  $\leftarrow$  1
        Let noiseHeight  $\leftarrow$  0
        For i from 0 to numOfOctaves begin
            sampleX  $\leftarrow$  x * frequency
            sampleY  $\leftarrow$  y * frequency
            perlinValue  $\leftarrow$  PerlinNoise(sampleX, sampleY)
            noiseHeight  $\leftarrow$  perlinValue * amplitude
            amplitude  $\leftarrow$  amplitude * persistence
            frequency  $\leftarrow$  frequency * lacunarity
        end
        heightMap[x, y]  $\leftarrow$  noiseHeight
    end
end
```

Map heightmap values to [0, 1]
return heightMap

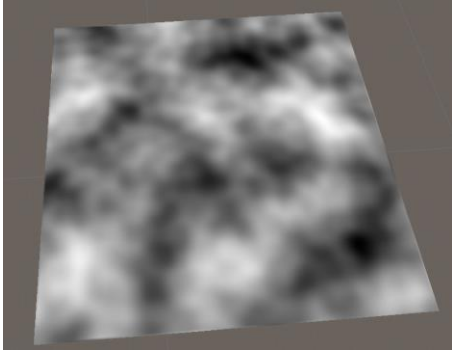


Figure 2.1 Sample Run 1

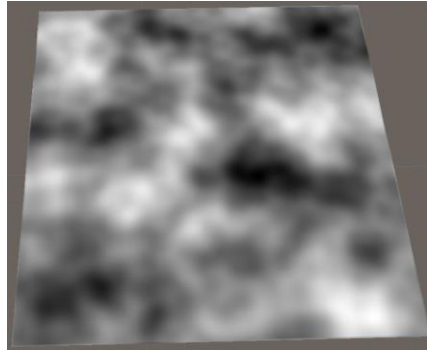


Fig 2.2 Sample Run 2

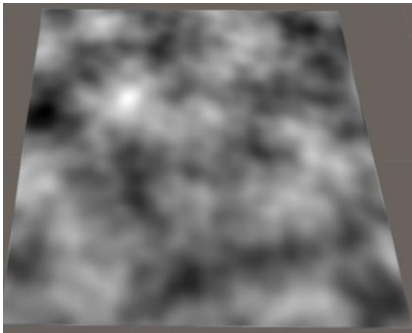


Figure 2.3 Sample Run 3

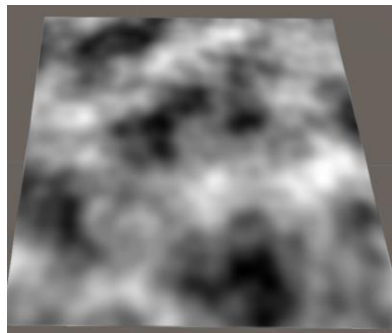


Figure 2.4 Sample Run 4

Four sample runs from **Figure 2.1-2.4** has the optimal parameters that I've found by testing which are Lacunarity = 2, Persistence = 0.5 using 3 octaves. From white to black, height goes lower.

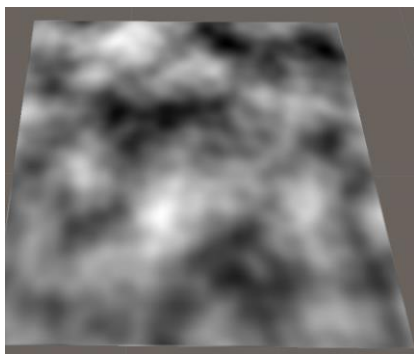


Fig 2.5 Lacunarity = 2, Persistence 0.5

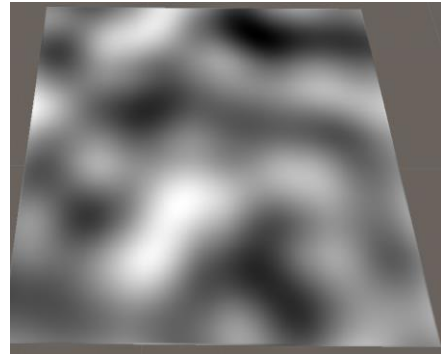


Fig 2.6 Lacunarity = 1 Persistence 0.5

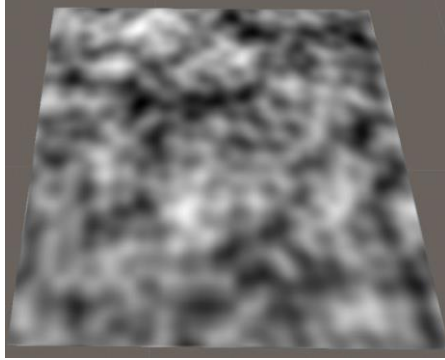


Fig 2.7 Lacunarity = 2, Persistence = 1

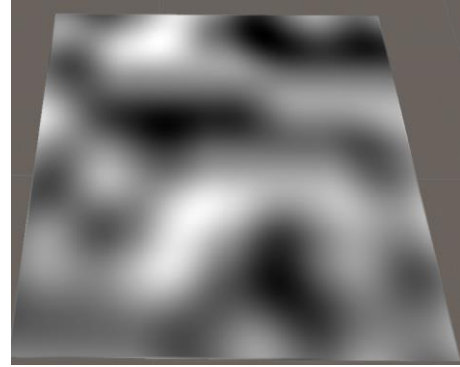


Fig 2.8 Lacunarity = 2, Persistence = 0

Here **Fig 2.5** is generated with optimal parameters, **Fig 2.6** has lacunarity of 1 and **Fig 2.7** has persistence of 0 which both have really simple noises with minimal detail which is not useful. **Fig 2.8** has persistence of 1 which is too complex too many details. These will make more sense in fully generated version of these.

The next step to generate the mesh is to have a set of vertices and triangles. There are height * width vertices in a mesh and the representation of triangles are showed in **Fig 2.9** and feeding these to a mesh object in unity and calling `RecalculateNormals()` method handles the generation of the mesh from the height map.

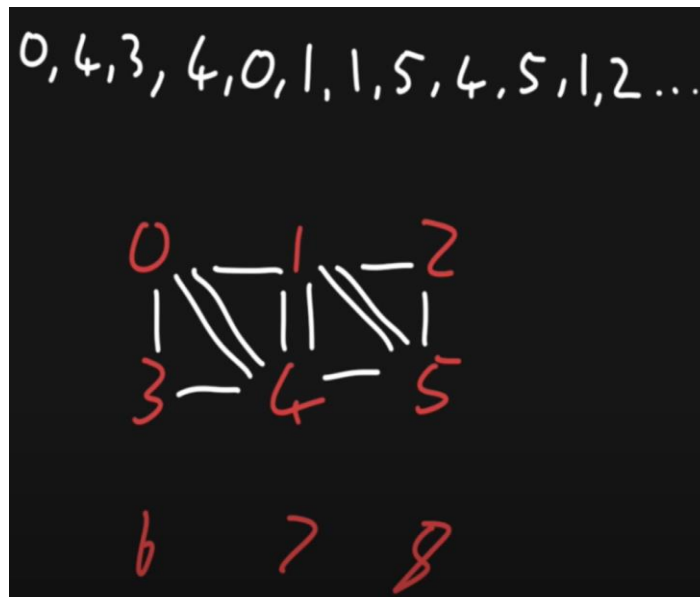


Fig 2.9 Triangulation of vertices

The numbers here represent vertices of the plane, there are **width * height** times vertices on a plane which is $100 * 100 = 10000$ as all the sample runs are generated from 100x100 planes.

Lastly, one of the problems with this approach is that anything below a certain height is considered water but the water level wouldn't be same after applying heightmap to the mesh as there are different levels of height between 0 and the specified water level. To fix that I've used Unity's **AnimationCurve** in order to set the height of each point under water level to water level. Animation Curve allows you to set numbers between two numbers to be equal to a number you want, in **Fig 2.10** -x axis represents heights of the points and -y axis represents their value. The curve is flat between 0.0 and 0.4 and they are equal to 0.4 which is the water level and after that the numbers and their values linearly increases. The linear equation for the flat part is $y = 0.4$ while the rest is $y = x$.

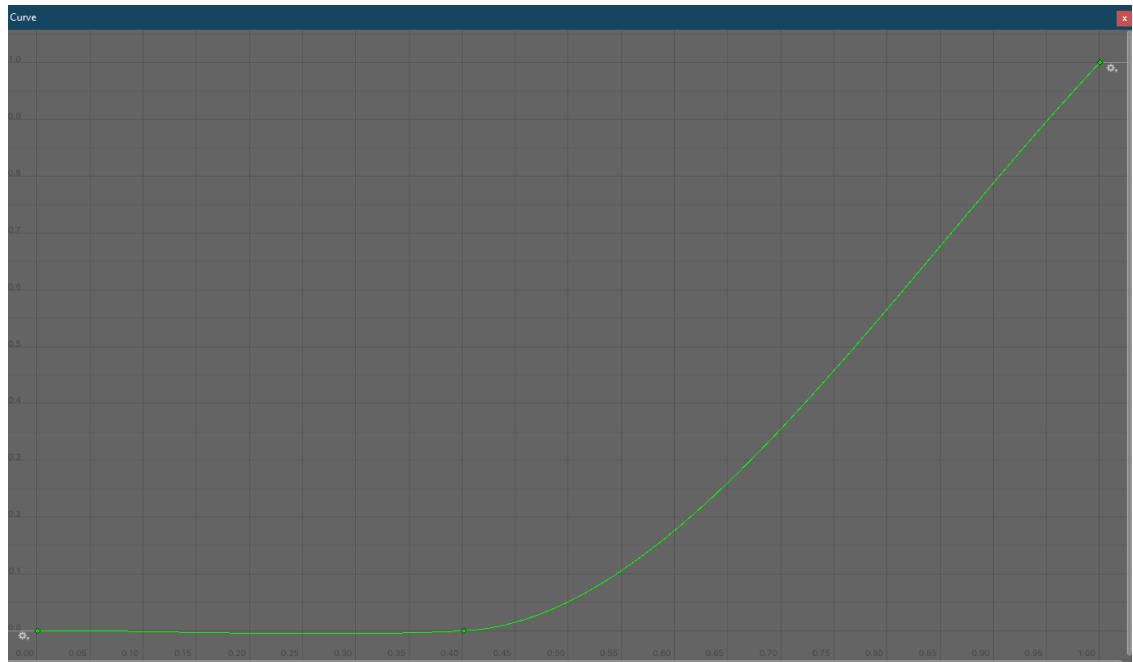


Fig 2.10 Animation Curve

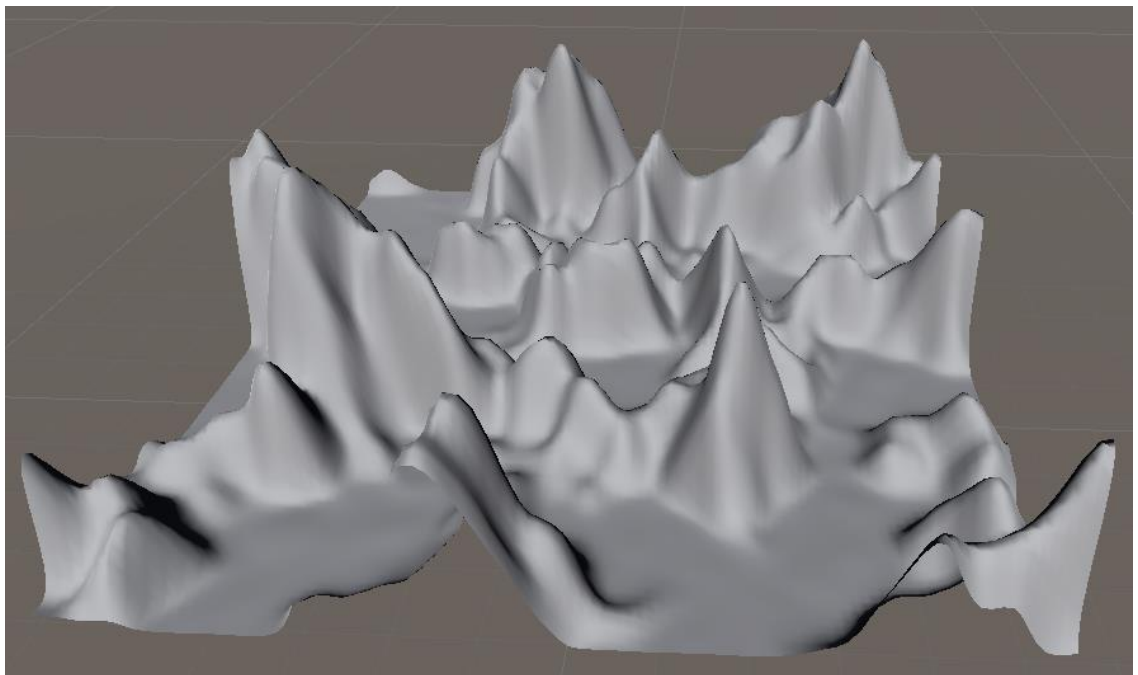


Fig 2.11 Example run of mesh generation

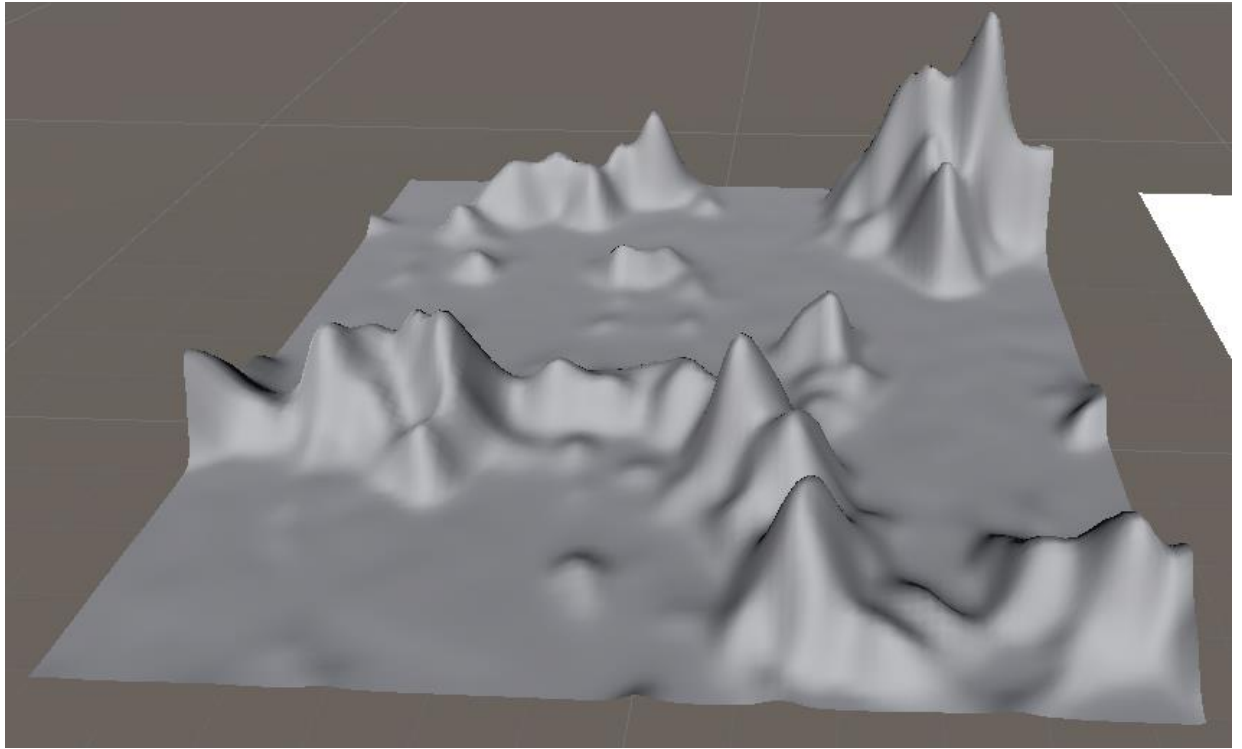


Fig 2.12 Example run of mesh generation

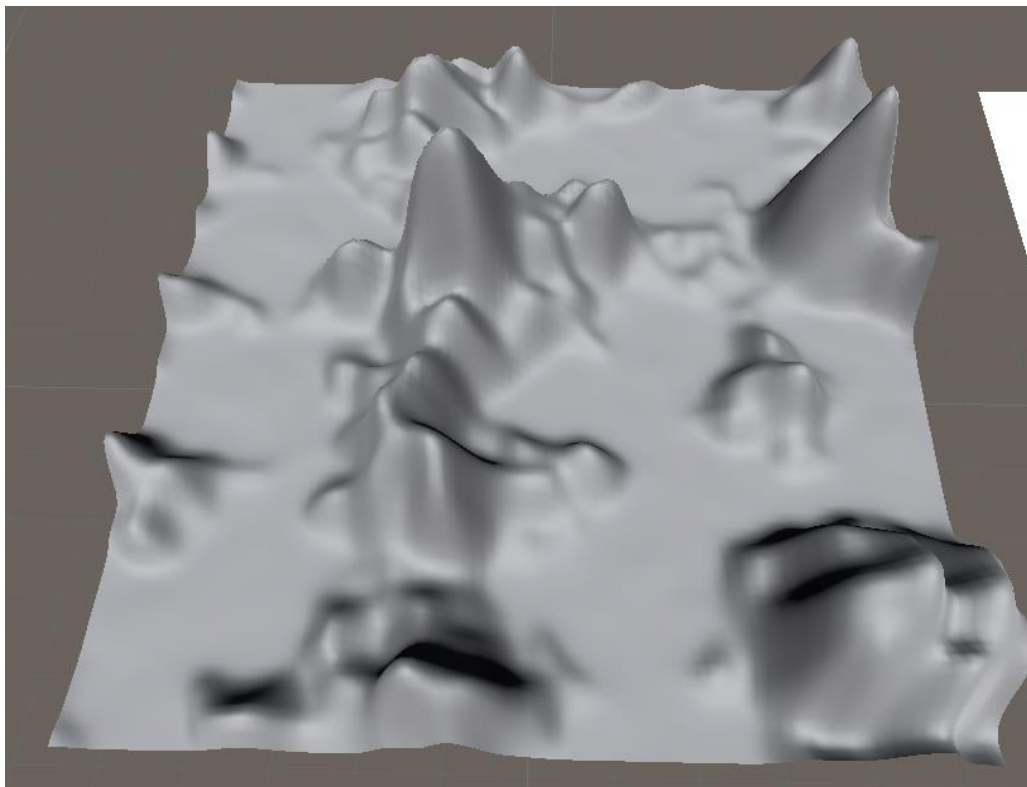


Fig 2.13 Example run of mesh generation

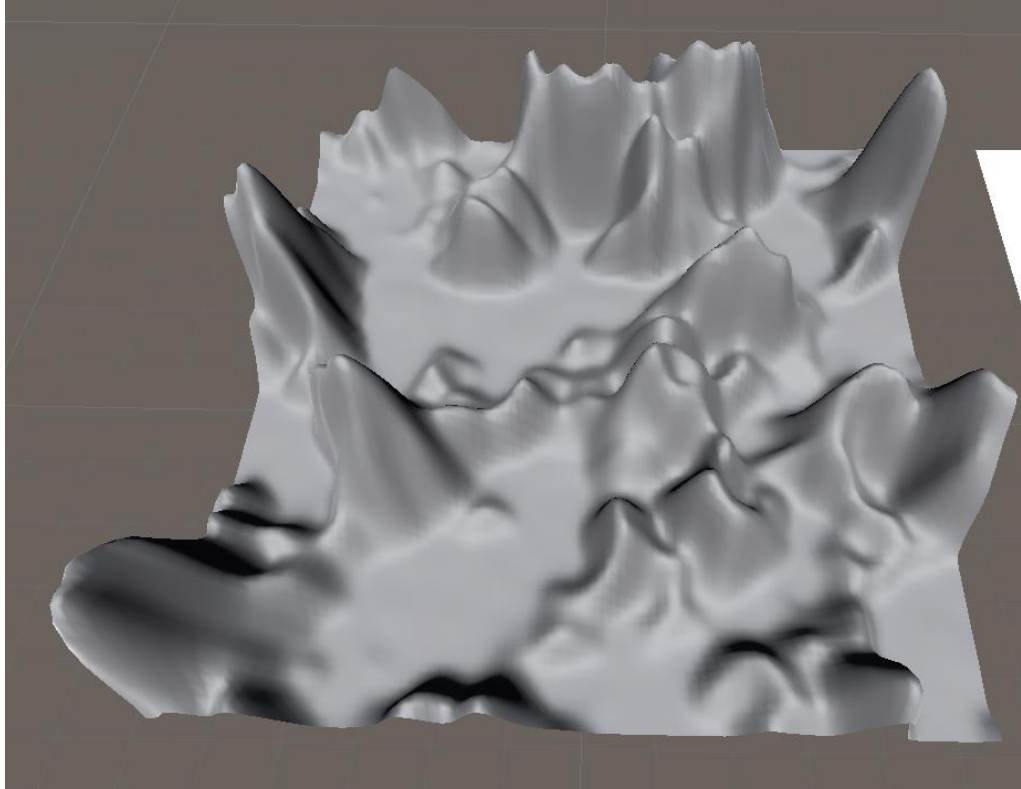


Fig 2.14 Example run of mesh generation

Fig 2.11 – Fig 2.14 are some of the example runs of mesh generation with optimal parameters used in **Fig 2.1 – Fig 2.5**.

2.2. GENERATING THE TEXTURE

Texture generation is deciding which parts of the mesh is more suited to which biome which is calculated for each point (100x100 plane has 10000 point). Each point is assigned a moisture and temperature level which both have 4 levels. Every combination of these two levels which is 16 in total corresponds to a biome as illustrated in **Fig 2.15**.

The biomes are decided by distance from the water and distance from the equator which is going to be the middle of the mesh and the temperature gets higher as you get closer to the equator in -y axis.

There are 4 different levels of temperature and 4 different levels of moisture zone and 9 different biome types in total. These biomes are represented by colors in the demo.

	hottest	hot	cold	coldest
dryest	desert	grassland	tundra	tundra
dry	savanna	savanna	boreal forest	tundra
wet	tropical rainforest	boreal forest	boreal forest	tundra
wettest	tropical rainforest	tropical rainforest	tundra	tundra

Fig 2.15 Simplified biome chart

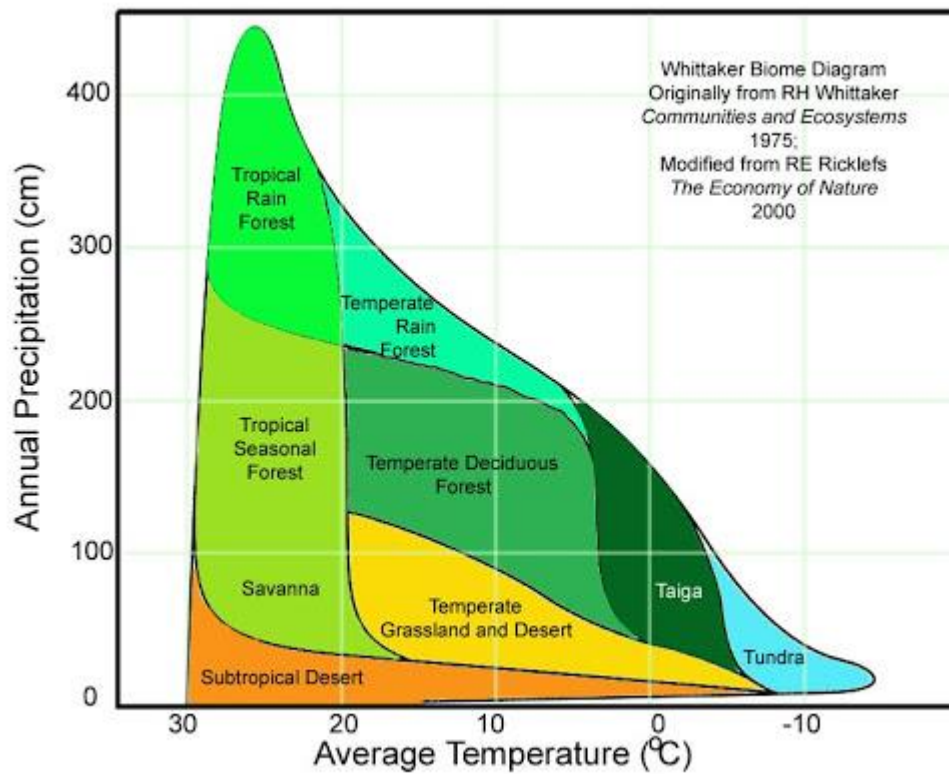


Fig 2.16 Whittaker Biome Diagram

The moisture map utilizes heightmap to decide how dry or wet an area is. **Fig 2.17** and **Fig 2.18** are examples from the moisture map, the higher the point is the dryer as red areas represent driest and cyan represents wettest. Heights that are under 0.5 are wettest as 0.4 is water level, heights between 0.5 and 0.66 are wet, heights between 0.66 and 0.8 are dry and heights that are higher than 0.8 is dryest.

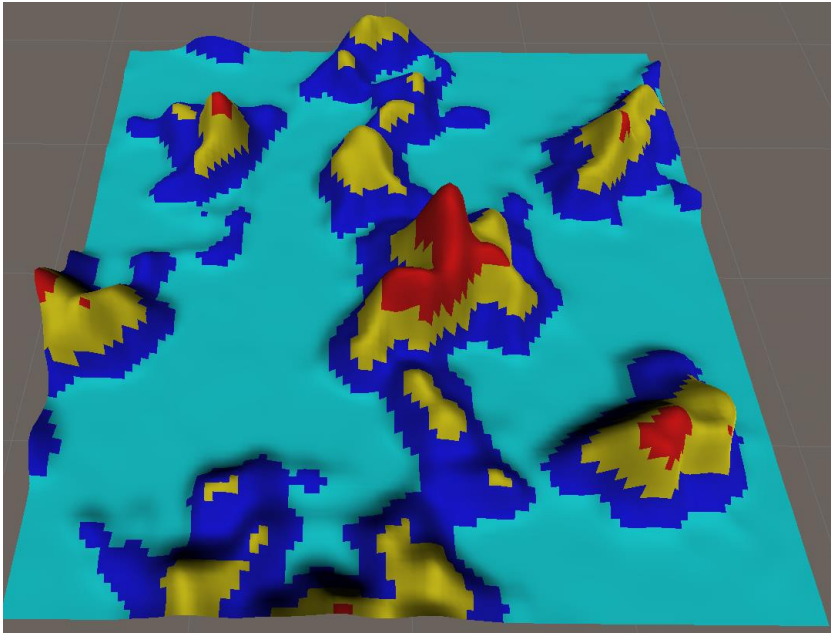


Fig 2.17 Sample Moisture Map

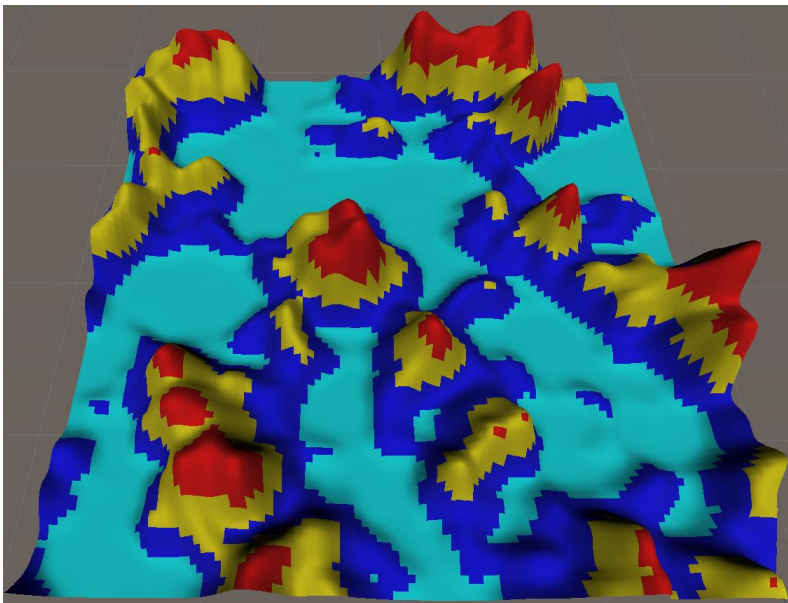


Fig 2.18 Sample Moisture Map

Heat map is a little bit more complex than the moisture map. There are two factors here first one is latitude, how close point is to the middle of the map in -y axis see **Fig2.19** for reference. The other thing is the height of the point, the higher the point is, the colder it should be.

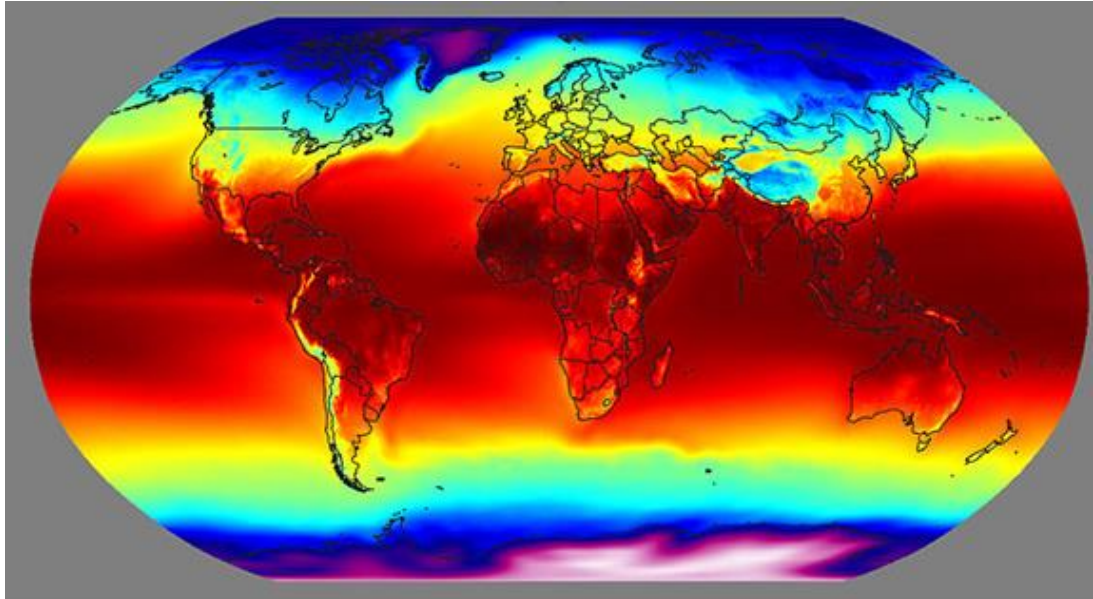


Fig 2.19 Latitude's effect on temperature

The formula of the heatmap calculation from the latitude is $\frac{mapHeight}{2} - i$ / $mapHeight$ where i is the current point in the -y axis. Since the distance can only be half of the mapHeight, the maximum value that the mapHeight can get is 0.5 in the case of i being equal to zero or mapHeight and the problem with that is heightmap values are between 0 and 1 so multiplying the heatmap values with 2 gives numbers between 0 and 1. The other thing is the values increases as you go from middle to edges which is kinda confusing since they are reverted, the colder the point is, the lower value it should have and to achieve that I've subtracted the numbers from one and the final formula became $(1 - 2 * (\text{Mathf.Abs}(mapHeight / 2 - i) / (\text{float})mapHeight))$. Since width doesn't affect the temperature, the heat value for same height is same for every width.

The next step is to involve height into the heatmap. While subtracting height value directly from the heat value may seem like the first option, since both height map and heat map values are between 0 and 1 the height map's affect on heat map is too significant that every point becomes cold or coldest. The way I decreased the impact of height map on the heat map, I used height maps value to the 10 (value 10). The number 10 is a result of trial error as the results are the most satisfying with number 10.

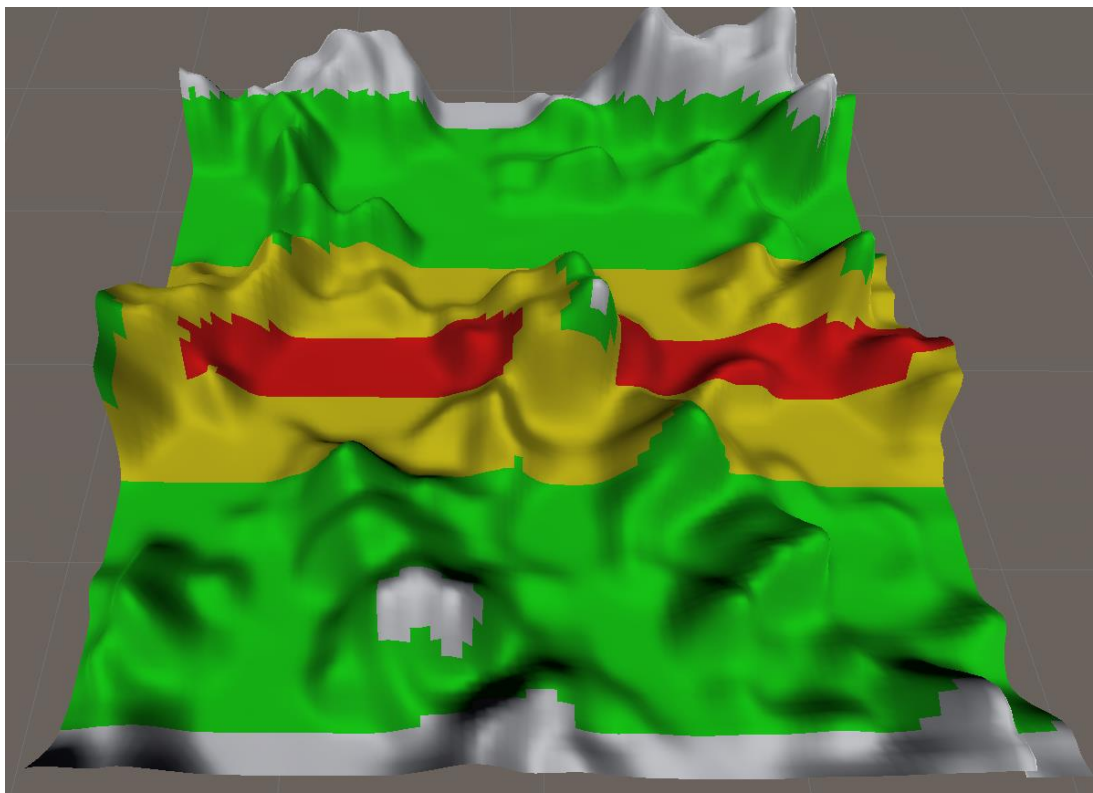


Fig 2.20 Heat Map with Power 10

Fig 2.20 is the result of using power 10 and the coldest level(white) can barely appear near equator with maximum height.

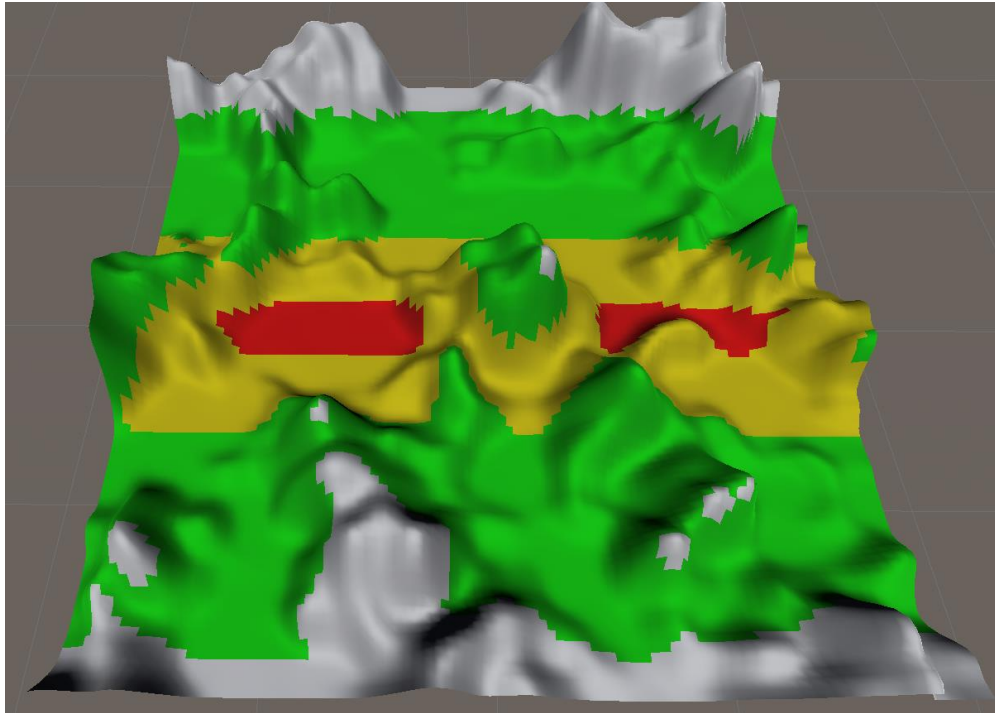


Fig 2.21 Heat Map with Power 5

Fig 2.21 is the result of using power 5 and the hottest points are far less than it should be and coldest points are too many as the heights impact is far more significant.

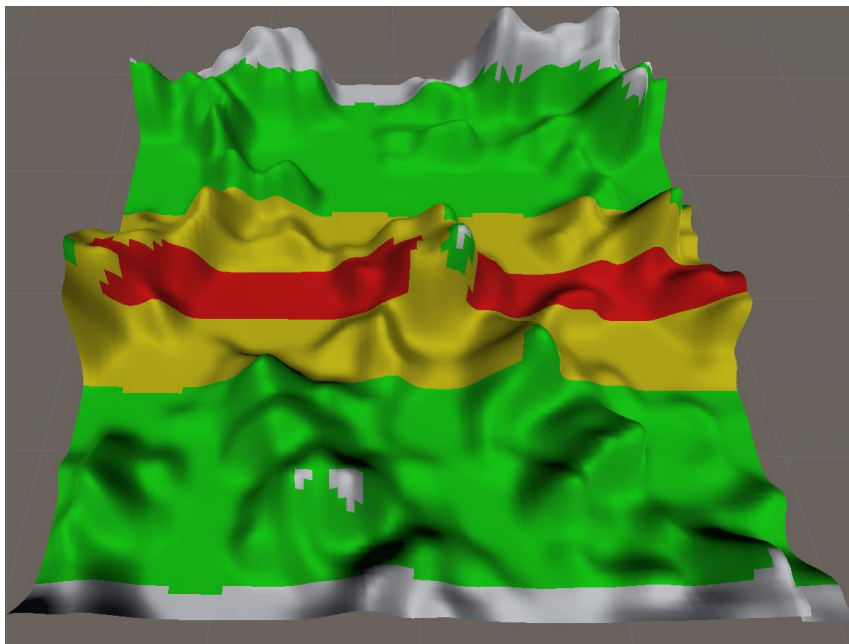


Fig 2.22 Heat Map with Power 15

Lastly, **Fig 2.22** is the result of using power 15 which has almost no impact on the map as it is far less significant that it hardly matters.

Last thing to point out is as points gets further away from equator, the heat gets lower and lower value and after some point, subtracting height from these values result in values that are less than 0 and those are set to 0.

Here's the pseudo code for calculating the heat map

Procedure calculateHeatMap

For j from 0 -> mapWidth begin

For i from 0 -> mapHeight begin

heatMap[j, i] ← (Mathf.Abs(mapHeight / 2 - i) / (float)mapHeight) - Mathf.Pow((precipitationMap[j,i]), 15f);

if heatMap[j, i] less than 0 -> heatMap[j, i] ← 0

end

end

Mathf.Abs(x) returns |x|

Mathf.Pow(f, p) returns f^p

The rest is using simple if clause to check which heat level the heatMap value corresponds to.

2.3 INFINITE GENERATION

In order to achieve infinite generation, the generator modifies the same mesh, uses same variables but with different values. To have a continuous feeling and not randomising every pixel, the generated noise had to be continuous and what I utilized is the fact that perlin noise is not stochastic which means for the same input the result will always be same.

The reason this is useful is, you can always randomize the initial input and use an offset value to navigate through the noise. To explain in a simpler manner, if the initial randomized value is lets say 100 then the middle of the mesh will have (100, 100) as it's coordinates in the noise and as you go right, all you have to do will be increasing the X value, (101, 100) and so on. Now the offset value comes in to play, instead of changing the initial randomized value 100, it adds offset to the initial value to make it look like it's moving.

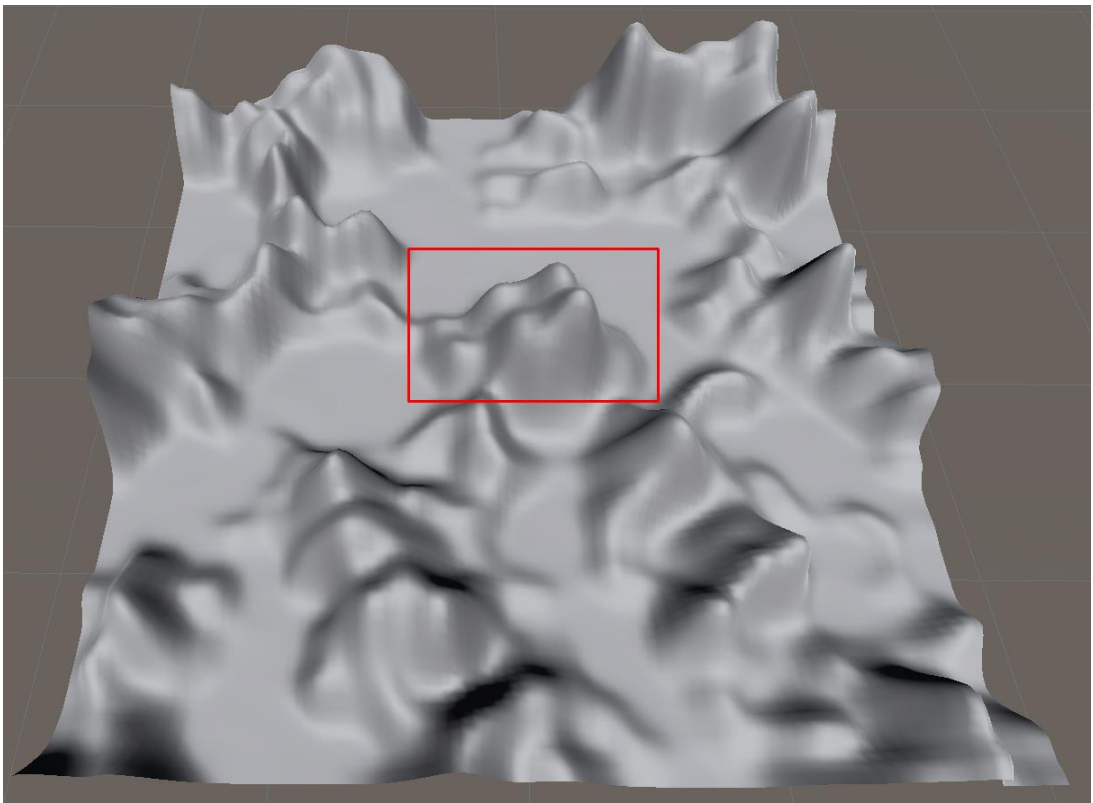


Fig 2.23 Before offset with a reference point

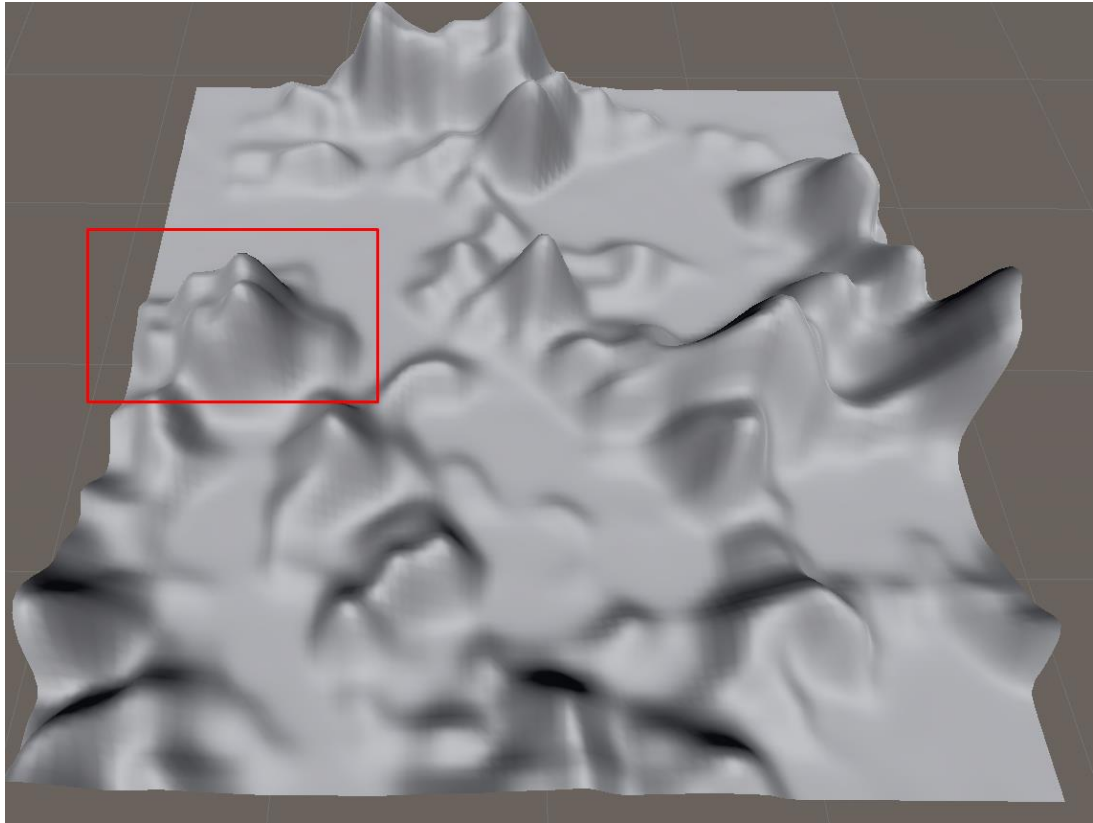


Fig 2.24 after offset with a reference point

3. PUTTING IT ALL TOGETHER

After deciding which point corresponds to which level of temperature and precipitation or moisture by utilizing heat map and precipitation map the next thing to do is deciding which combination of temperature and moisture levels corresponds to which biome which can be edited via Unity's UI **Fig 3.0** and then applying the color of the biome that corresponds to for each point in the mesh. **Fig 3.1** shows the results after applying biomes to the mesh.






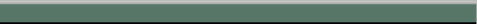



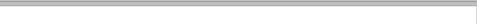
▼ Tropical Rainforest	Name	Tropical Rainforest
	Height	0.45
	Temperature	Hottest
	Precipitation	Wet
	Color	
▼ Savanna	Name	Savanna
	Height	0.55
	Temperature	Hot
	Precipitation	Dry
	Color	
▼ Savanna	Name	Savanna
	Height	0.6
	Temperature	Hottest
	Precipitation	Dry
	Color	
▼ Boreal Forest	Name	Boreal Forest
	Height	0
	Temperature	Cold
	Precipitation	Dry
	Color	
▼ Boreal Forest	Name	Boreal Forest
	Height	5
	Temperature	Cold
	Precipitation	Wet
	Color	
▼ Boreal Forest	Name	Boreal Forest
	Height	5
	Temperature	Hot
	Precipitation	Wet
	Color	
▼ Tundra	Name	Tundra
	Height	5
	Temperature	Coldest
	Precipitation	Wettest
	Color	
▼ Taiga	Name	Taiga
	Height	5
	Temperature	Coldest
	Precipitation	Wet
	Color	
▼ Tundra	Name	Tundra
	Height	5
	Temperature	Coldest
	Precipitation	Dry
	Color	
▼ Tundra	Name	Tundra
	Height	5
	Temperature	Coldest
	Precipitation	Dryest
	Color	

Fig 3.0 Biomes in editor

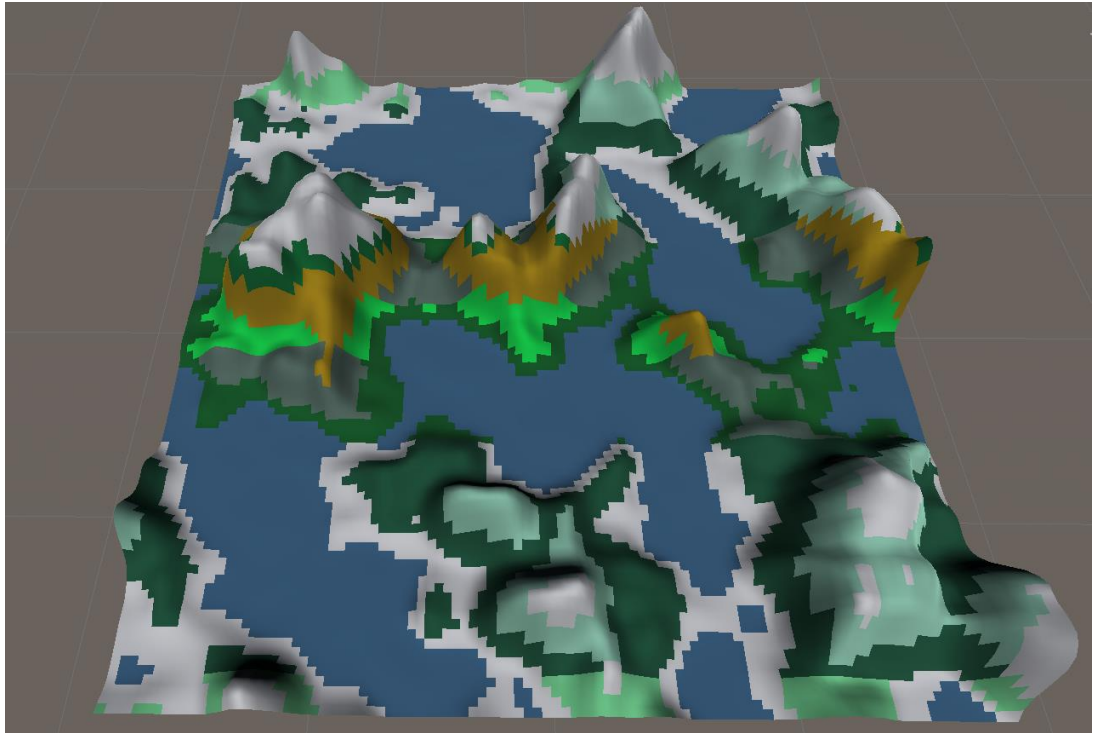


Fig 3.1 After applying biomes

4. TESTS AND RESULTS

Here are some example runs with same parameters but different seeds

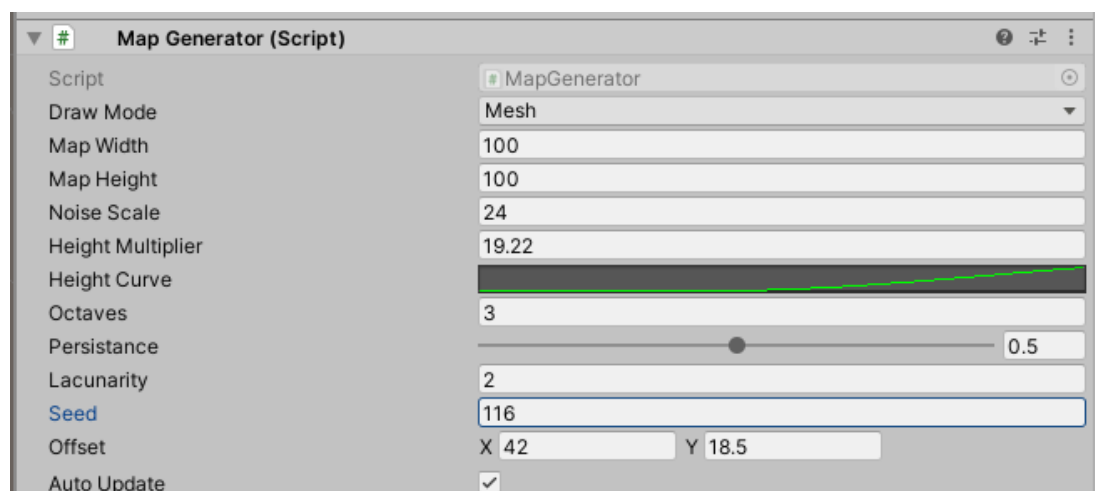


Fig 4.0 Parameters for the following runs with different seeds

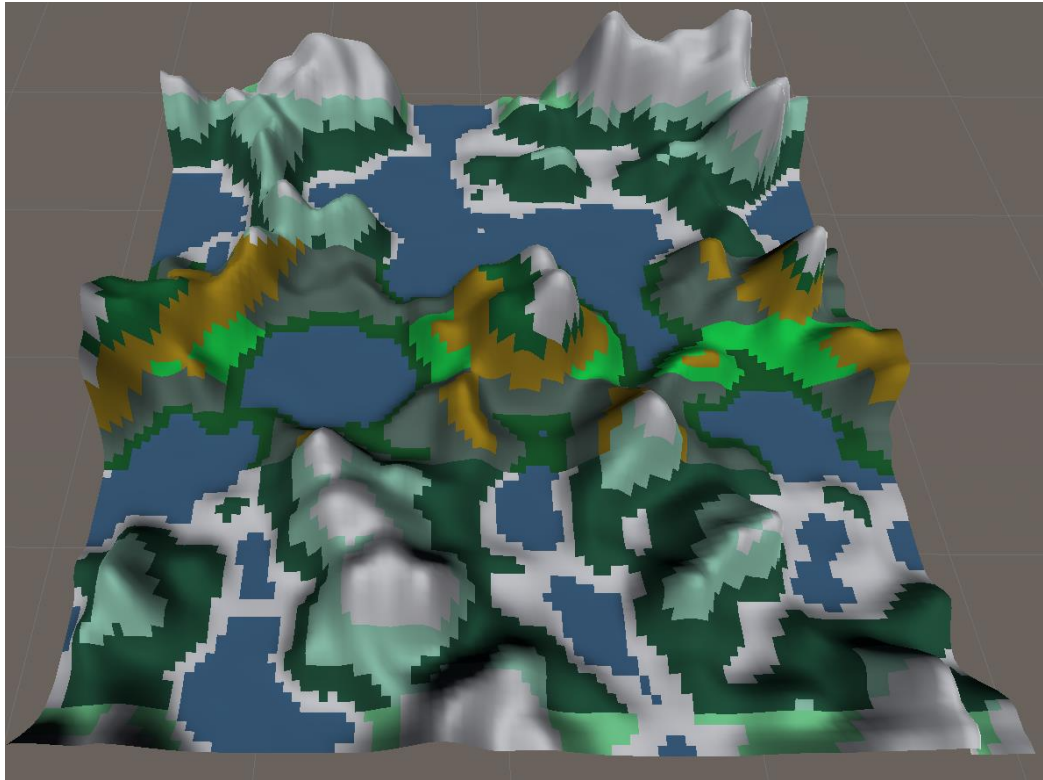


Fig 4.1 Sample Run

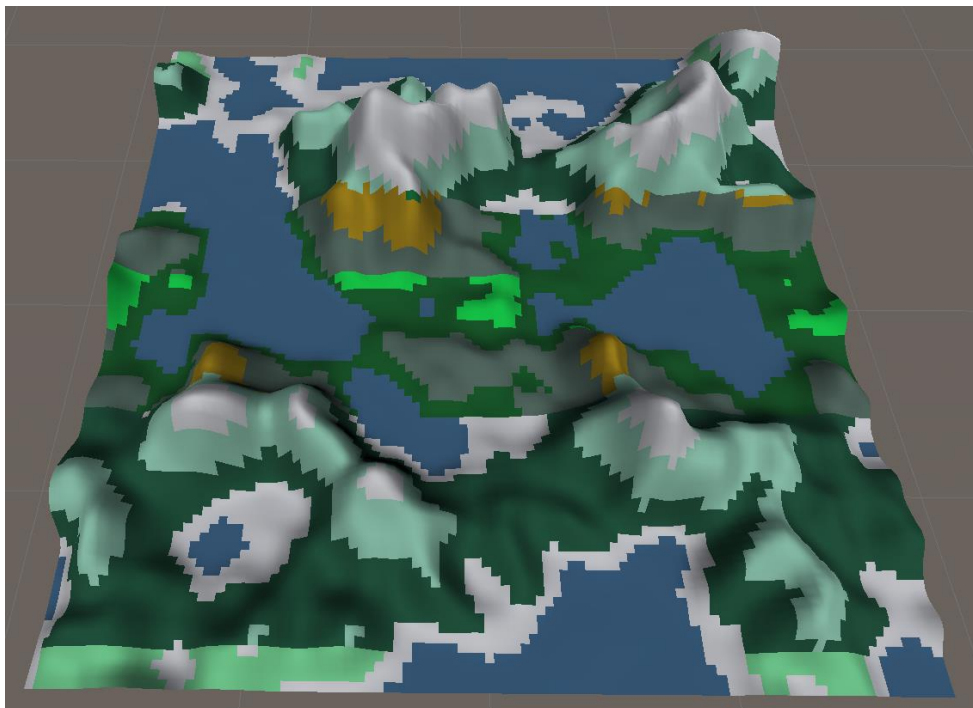


Fig 4.2 Sample Run

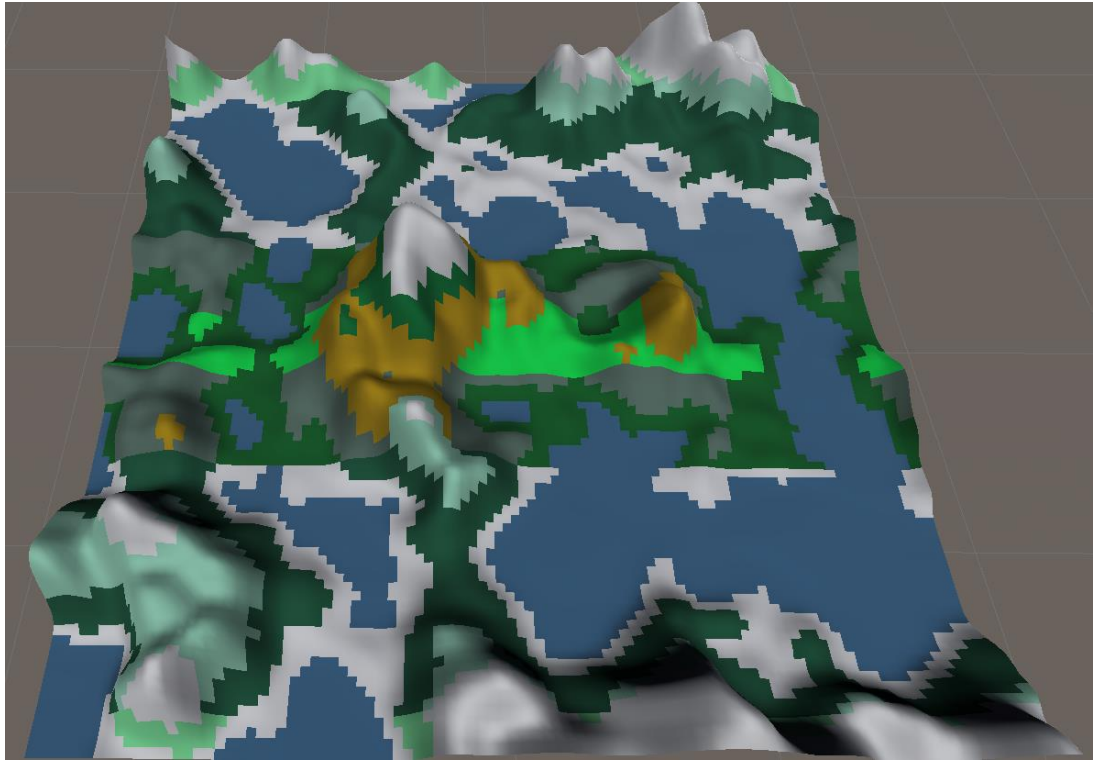


Fig 4.3 Sample Run

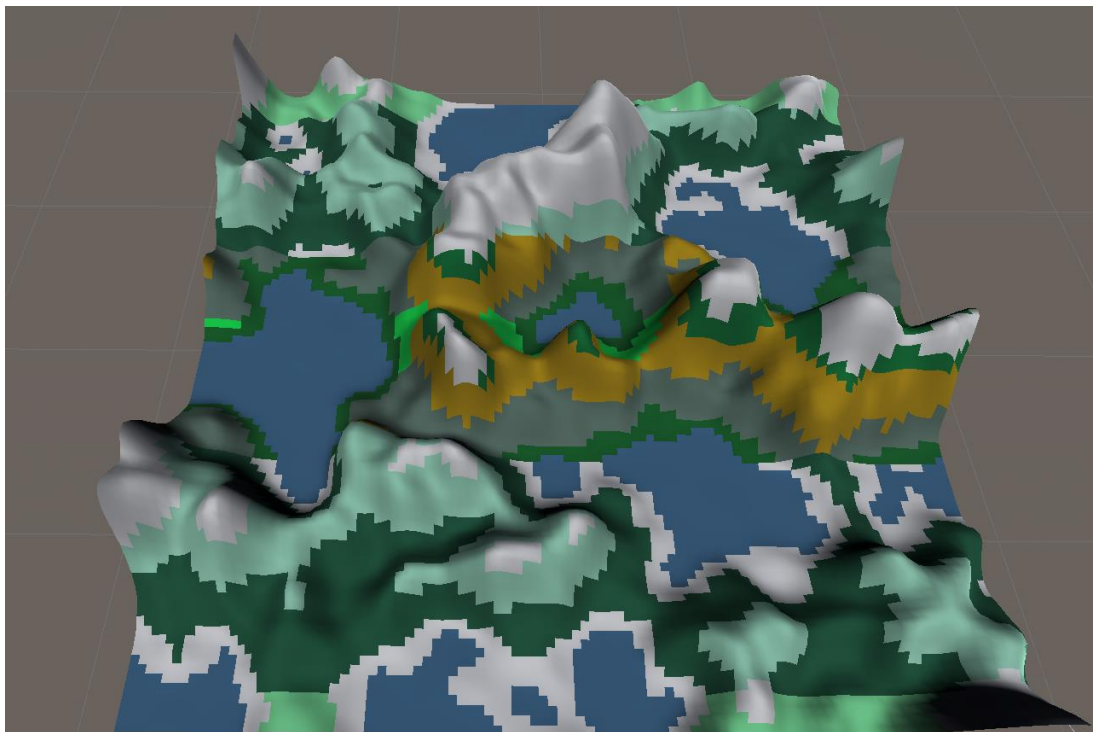


Fig 4.4 Sample Run

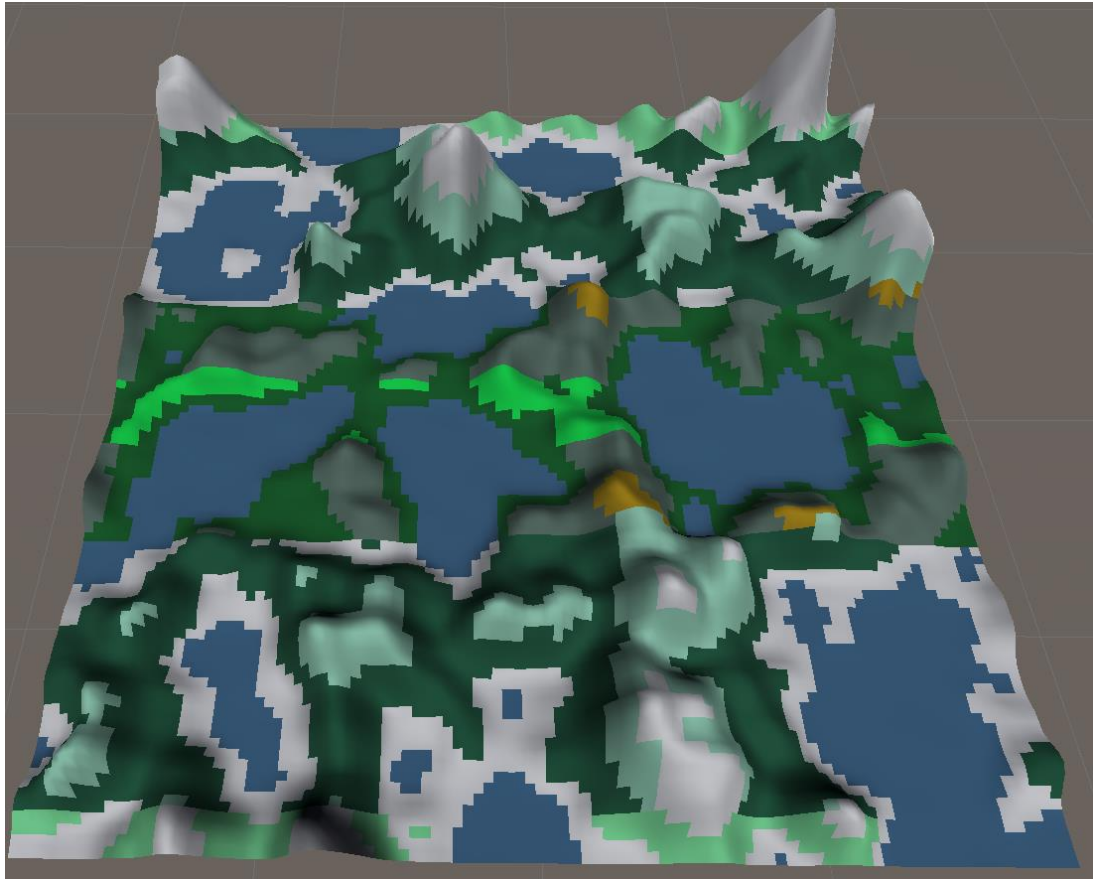


Fig 4.5 Sample Run

Some of the noticeable problems with the generations are the transitions between biomes are not smooth, there are noticeable lines that separate biomes and it feels like there's too much going on, so many different biomes clamped together but part of the reason behind that is the size of the map. With a bigger scale map and smoother transitions between biomes and with using textures instead of colors, it would generate much more natural and beautiful looking terrains that can be used in games.

Overall, I'm happy with the results as this is a good base line for procedural terrain generation that I will be improving the mentioned problems and use it on an actual game.

5. CONCLUSION

This project offers up a solution to one of the most common problems in game industry, replayability. It is a tool that developers can integrate into their games and modify it to their need. It can generate a terrain from 100x100 plane.

While the generator has basic functionality, there are a lot of improvements that can be done. The biggest problem to me is that the biomes are only represented by paints. They can be replaced by textures or even models for the trees and have another parameter to control the tree frequency. Another improvement would be having more smooth transitions between biomes. Also, the plane size cannot be higher than 100x100 because of the limit of array size in Unity C# arrays, this is particularly important because there are a lot of biomes and on a smaller scale, they look to clamped together which would look better on a larger scale.

6. SUCCESS CRITERIA

- 1- Infinite terrain generation that extends by 100x100 every second
Computer Specs: GTX 960, Ryzen 2700 @3.7GHz, 16GB RAM
- 2- Generating terrain with 2 water sources and at least 5 different from 100x100 plane under one second
- 3- Maximum 5% memory usage difference between static generation and infinite generation from 100x100 plane with same parameters.

While criteria 1 and 3 were passed, criteria 2 failed as I couldn't implement the water sources to the generation.

7. REFERENCES

- [1] Rose, T. J., & Bakaoukas, A. G. (2016). Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques. 2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES).
- [2] Kamal, K. R., & Uddin, Y. S. (2007). Parametrically controlled terrain generation. Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia - GRAPHITE '07.
- [3] Togelius, J., Preuss, M., & Yannakakis, G. N. (2010). . PCGames '10 Article No. 3 *Towards Multiobjective Procedural Map Generation*
- [4] Designer Worlds: Procedural Generation of Infinite Terrain from Real-World Elevation Data by Ian Parberry Journal of Computer Graphics Techniques Vol. 3, No. 1, 2014
- [5] Polygonal Map Generation for Games,
<http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/> [20 June 2020].