Poppe Arno

An approach to create Landmass

Graduation work 2021-2022

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT

Terrain generation is always a bit tricky to get right for both consummate, professionals and game enthusiasts. Landmass does a great job in solving that in Unreal, we are going to approach it in Unity. By implementing the tool into Unity, we are able to modify the terrain as we wish using parameters and techniques chosen for procedural terrain generation. More specifically we will make use of Perlin noise as a random generator, Bezier curves to make splines and octaves to generate finer terrain detail.

The research question is as follows: How to make a similar terrain editor to Unreal's Landmass tool in Unity?
The application itself is written entirely in C# and a custom algorithm was used to get the distance field.
The main findings are that the implementation is a good start but needs further development.

INTRODUCTION

When I started my journey into game design there was always one topic that I never managed to tackle properly. How to sculpt terrain in unity in a correct fashion. Whenever I tried to make some parts of the terrain look nice, I would ruin other parts and go over the undo limit.

The industry is facing similar issues, they must spend a lot of time and resources on making the environment where the game will take place.

Smaller teams have it also very difficult as they often don't have the time or resources to do so.
The aim of this paper is to implement a terrain tool that is similar to Unreals Landmass Terrain Tool in usage. Using the same principals as Landmass, relying on certain parameters to influence a landscape and change the shape of a spline to move a mountain.

Since the goal is to implement a tool similar to Unreals Landmass plugin into Unity, are there some topics that need to be addressed first.

As of writing this, the tool is still experimental in Unreal Engine 4.27.
Meaning the current version of Landmass might be different from the one described further in this document.

RELATED WORK

Procedural generation has been a subject of research for decades. It was experimented with and iterated upon since it started in the 1980s. Noise-based methods are mostly used and well-known. The most used ones are Perlin noise by Perlin and his improved variant Simplex. This is a rather simple and efficient method to use in terrain generation. [1]

1. UNREAL'S LANDMASS TOOL

1.1. UNREAL'S LANDMASS TOOL

The Landmass Blueprint plugin from Unreal is a new way of procedurally and non-destructively modifying the landscape. This feature alongside with landscape layers has made editing large maps significantly easier. Before, any changes made to the landscape were permanent. But now it's a simple case of clicking and dragging to literally move mountains.
If there are any changes that might ruin the landscape it is now just a matter of disabling a layer or removing a spline. [2]

It is designed in such a way that designers can iterate quickly over changes that they have made in the editor and test whether the terrain works for the game. Things such as step height and slope can be tested and iterated upon if need be. The difference in the following two pictures was accomplished by changing a single parameter, changing the slope angle so that it becomes walkable for the player. [2]

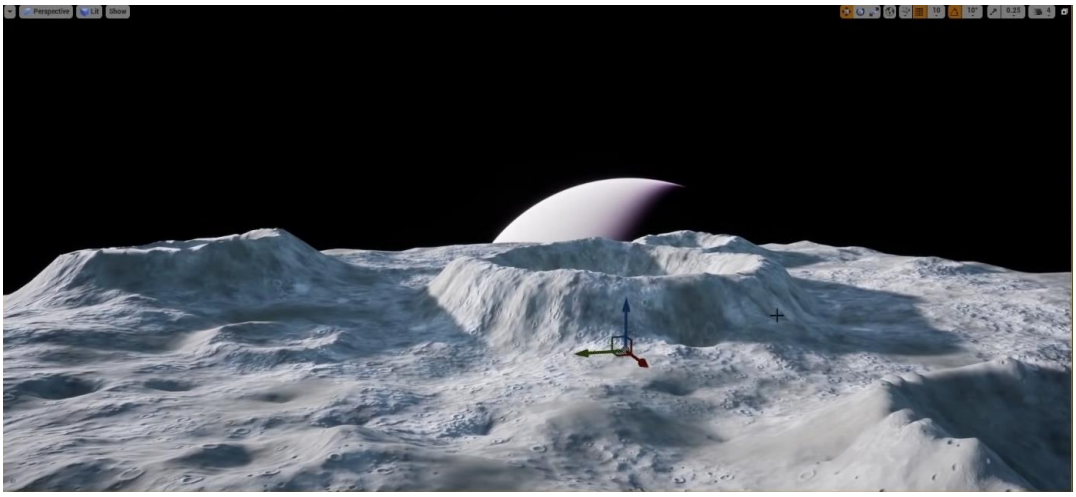


Figure 1 - steep slope on a crater [2]

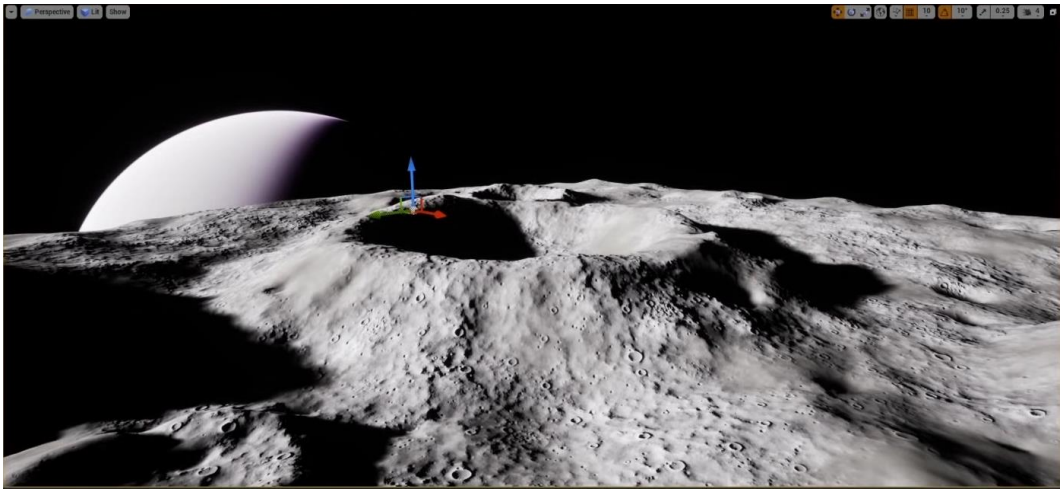


Figure 2 - lesser steep slope on the crater [2]

1.2. HOW IT WORKS

The landmass tool from Unreal Engine uses a combination of techniques to manipulate the terrain.

The first one is effectively using a layer with a random generator, mixing it with the existing visible layers and adjusting the parameters. [2]

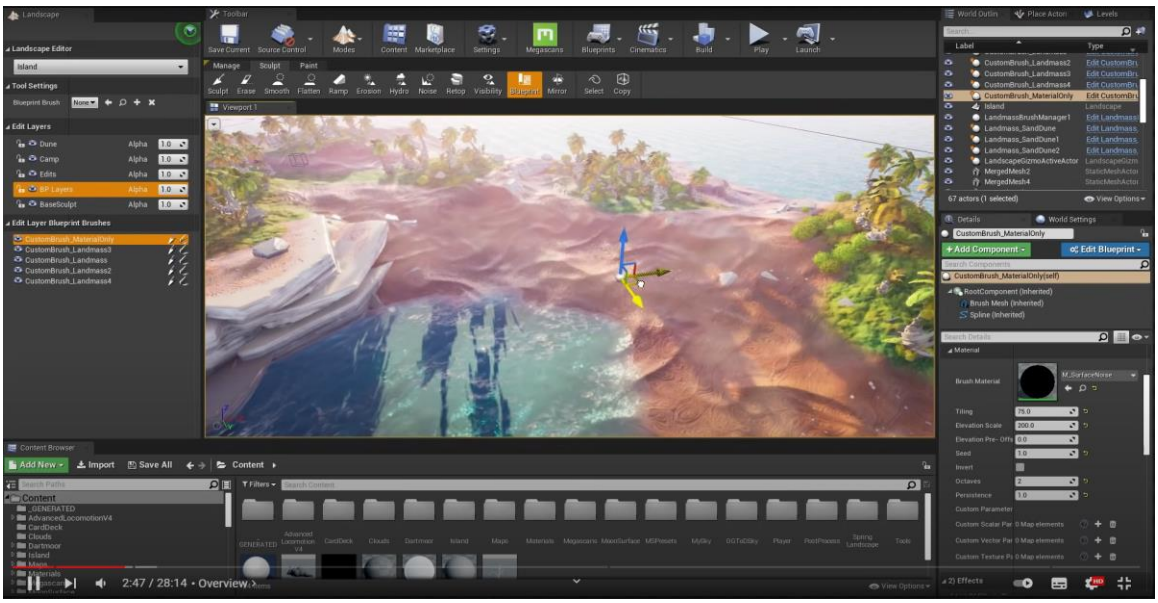


Figure 3 - Landmass tool [2]

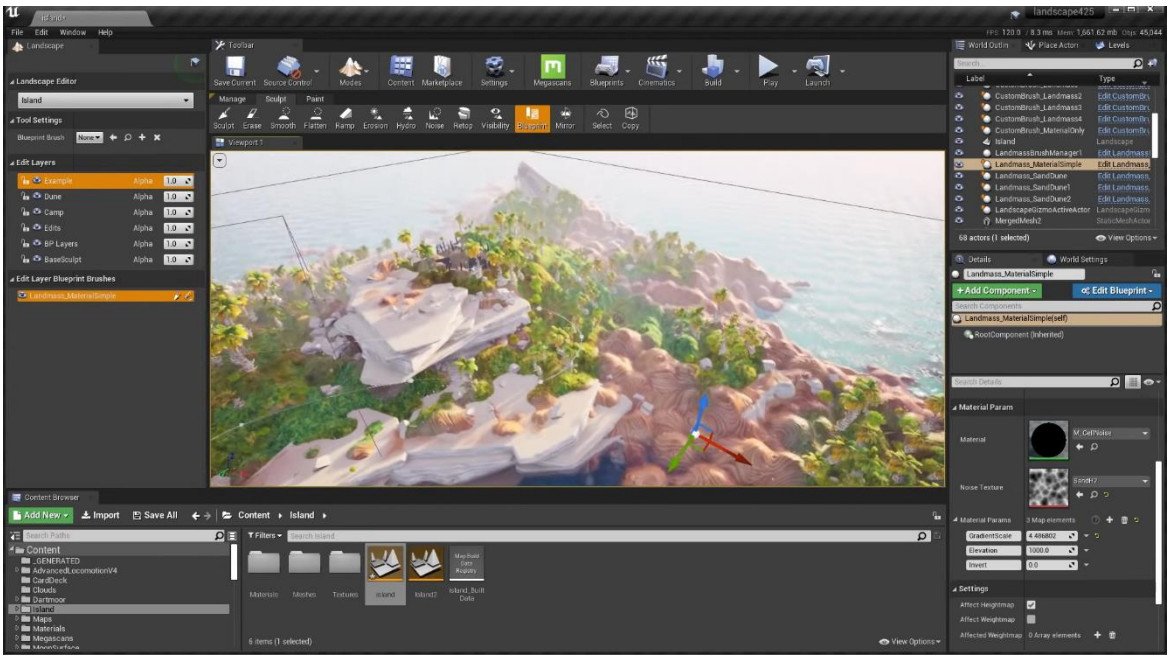


Figure 4 - Landmass tool with different parameters [2]

The layers themselves are modified individually using a height texture.
The elevation also plays a big role here as that will affect the height difference greatly.

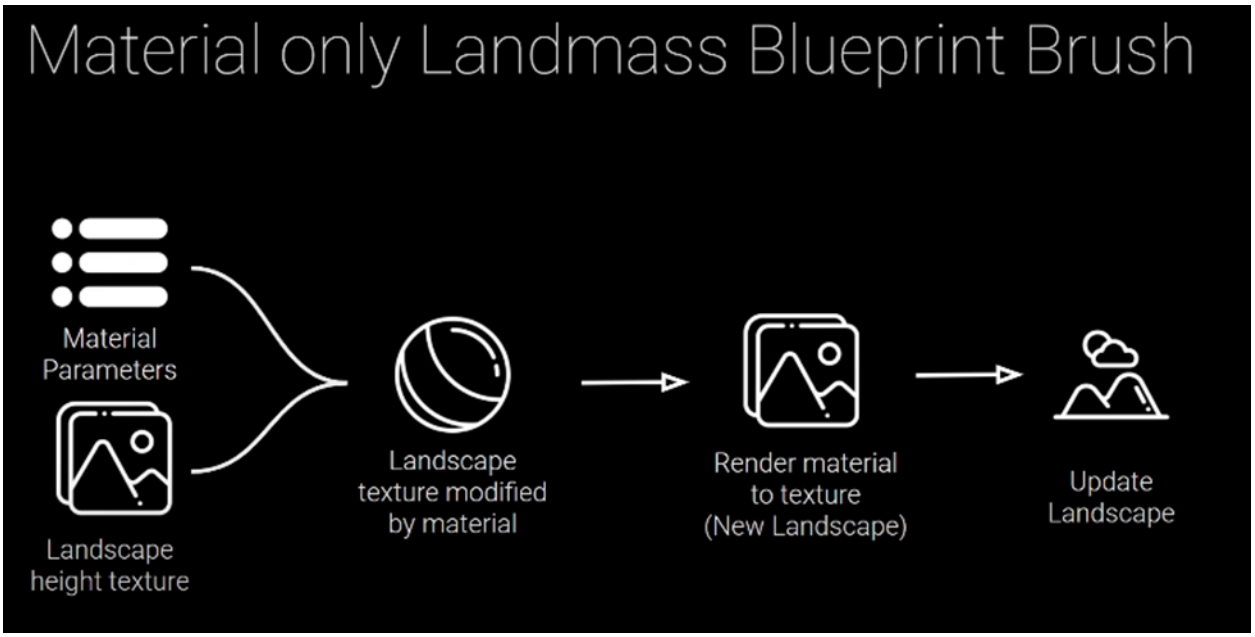


Figure 5 - explanation material only Landmass blueprint brush [2]

As shown by the graph used by Arran Langmead. Landmass' blueprint brush uses the material parameters to combine it with the heightmap into a landscape that is modified by the material. Landmass itself is primarily focused on the height texture. It will then render the material to texture and afterwards it will be updated. [2]

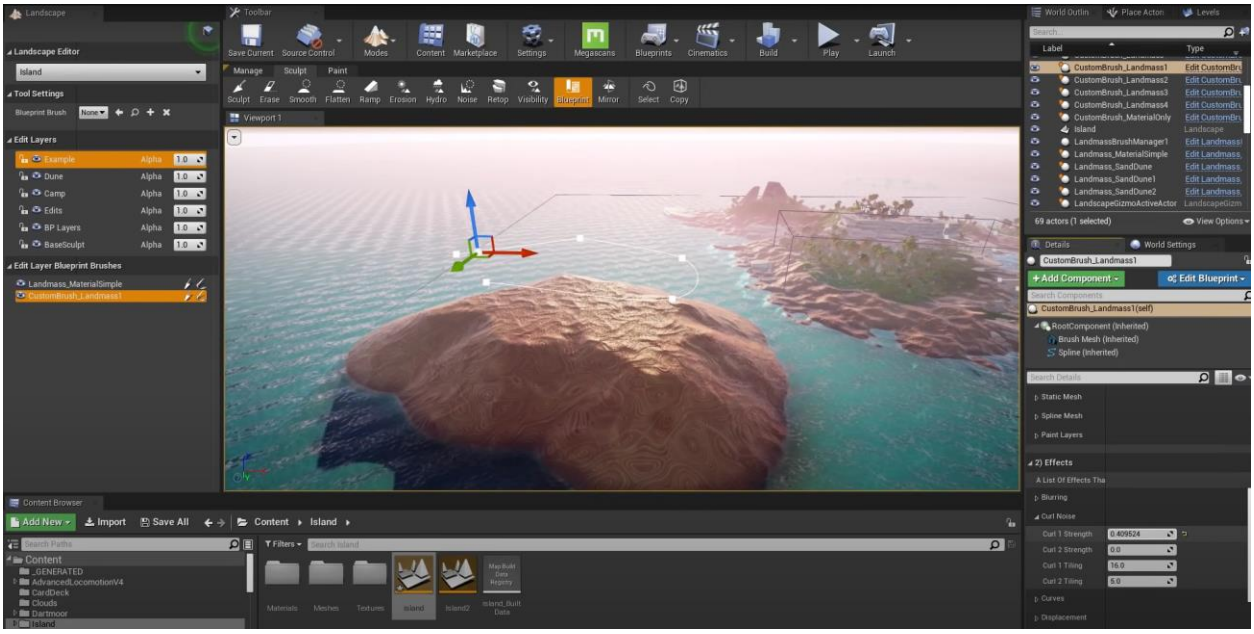
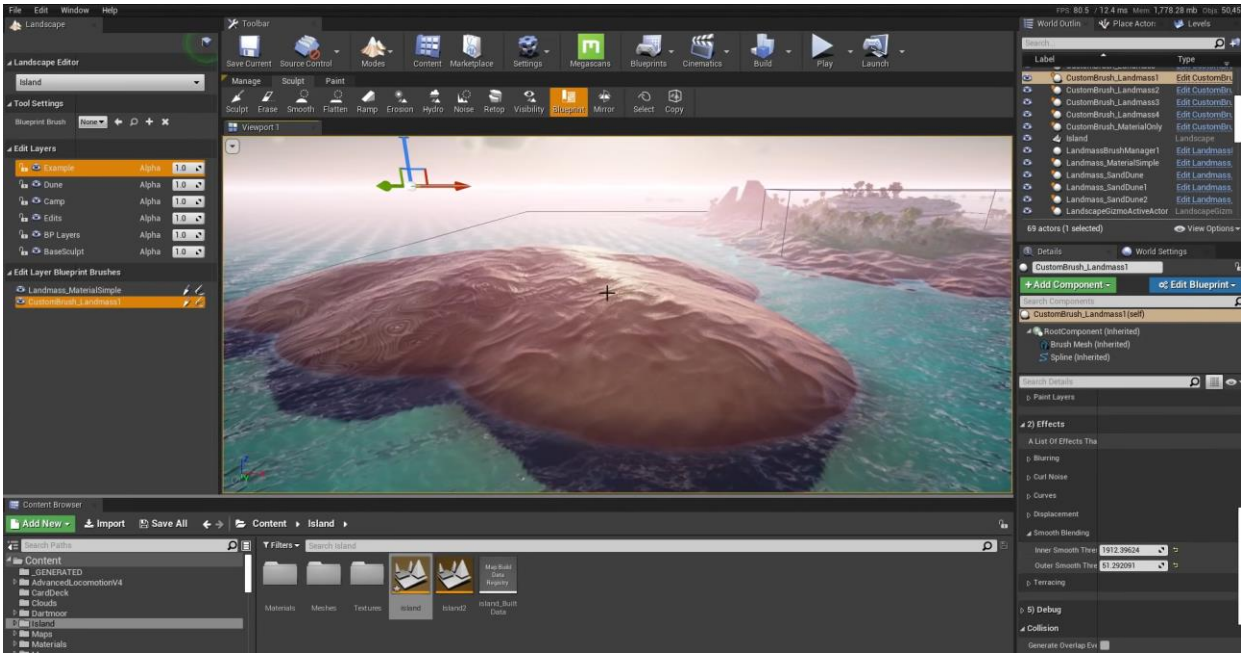


Figure 6 - spline modification layer demonstration [2]

The second part is spline-controlled terrain modification with the help of a series of parameters.
Allowing you to modify a specific part of the landscape. This still redraws the entire landscape as it must update its heightmap.
They warp the shape of the selected terrain giving you the option to blur, curl, curve, displace, blend, and enable terracing.



` **Figure 7 - Spline but with more smoothing [2]**

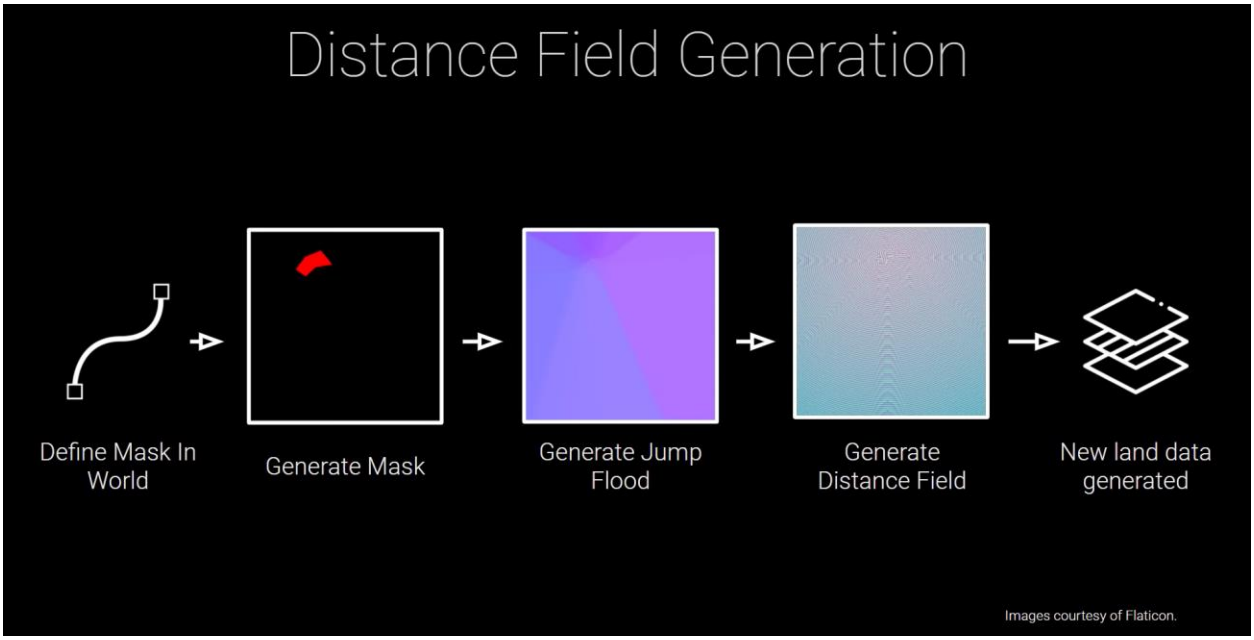


Figure 8 - From spline to generated land [2]

To get the splines into the heightmap, it needs to generate a mask around the spline first, then make a jump flood out of that mask and then generate a distance field based off the jump flood. Before taking elevation into account and making it part of a normal layer. [2]

A jump flood is an intermediate step that we could use to create Voronoi diagrams. We are going to use it to create a distance field. This is needed to create and manipulate the slope of the shape. [3]

2. RANDOM GENERATORS

2.1. PERLIN NOISE AND SIMPLEX NOISE

Perlin noise is a procedural texture primitive, a type of gradient noise used by visual effects artists to increase the appearance of realism in computer graphics. [4] The function has a pseudo-random appearance, yet all its visual details are the same size. This property allows it to be readily controllable; multiple scaled copies of Perlin noise can be inserted into mathematical expressions to create a great variety of procedural textures. Synthetic textures using Perlin noise are often used in CGI to make computer-generated visual elements – such as object surfaces, fire, smoke, or clouds – appear more natural, by imitating the controlled random appearance of textures in nature. [4]

The variant Simplex is Perlin noise but with fewer artefacts on the edges and, in higher dimensions a lower computational overhead. It was also invented by Ken Perlin in 2001.

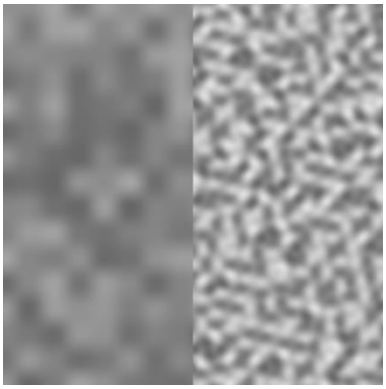


Figure 9 - Perlin noise (left) vs Simplex noise (right)

2.2. CUBIC NOISE

Cubic noise is essentially scaled up white noise with cubic interpolation. Cubic interpolation ensures the result is smoothed enough to remove grid-like artifacts. The values are based on coordinates, every pixel in this example represents a set of coordinates. These coordinate values can be tiled to make a tiling noise texture. All values produced by the cubic noise algorithm lie within the range of 0 and 1. [5]

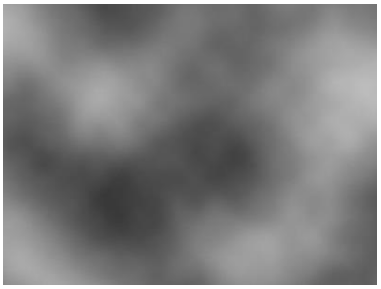


Figure 10 - Cubic noise

3. OCTAVES

To give terrain more detail there are techniques that make use of common noise functions overlapping. multiple octaves of noise maps, as this is well suited to generate a general fractal terrain. Fractal in the sense that it is the same generated value but has certain values manipulating the octaves. This method is fast, scalable, not too complex regarding usability and sufficient as a coherent coarse basis. The noise parameters as well as the number octaves can be set by the user. Changing it is a rather easy modification. [1]

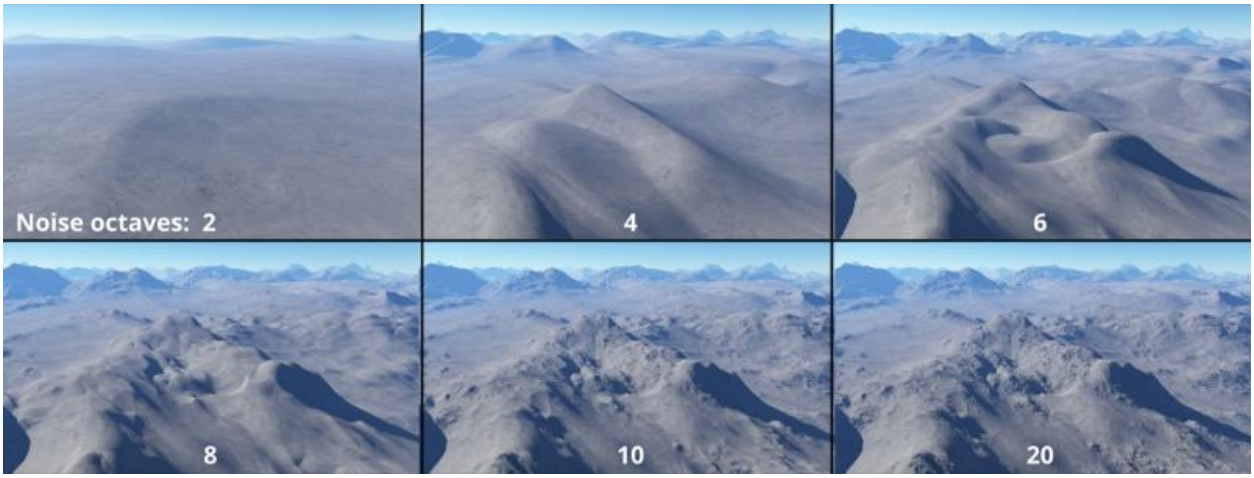


Figure 11 - Different octaves

As shown above octaves and increasing the number of octaves can make a big difference in how the landscape gets transformed. This should be a good enough start for our base layer. [1]

4. BEZIER CURVES

4.1. WHAT ARE BEZIER CURVES

A Bezier curve is a curve that defines a path defined using 3 points.
Where the first point is the begin point and the last one is the end point, while a third point decides how the interpolation between the two works.

Both blue points are the same percentage on their respective line. In between those two another point is placed which has the same percentage as the other two segment points. [6]

A collection of all the points between the begin point and end point will make a quadratic Bezier curve.

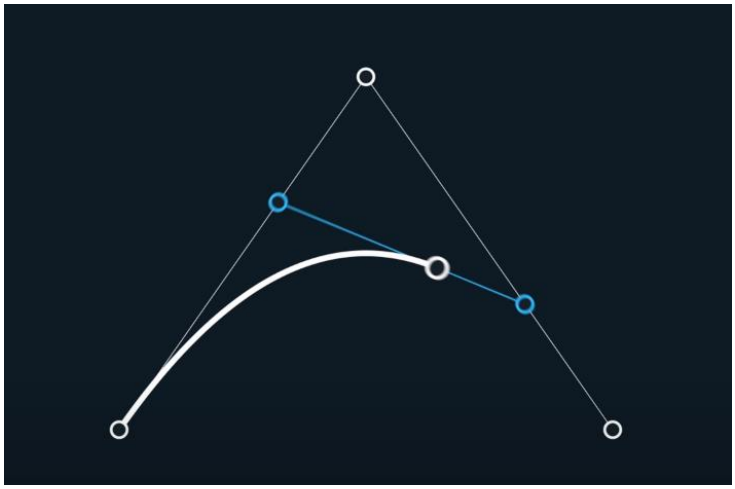


Figure 12 - Quadratic Bezier curve [6]

To have more control over the curve, simply add another control point.
This is called a Cubic Bezier curve and will be the kind of Bezier curve that will be used in this paper.

Some of the segment points will be differently calculated now since there is another control point to be considered. Between the old control point and the new control point and the new control point and the end point will the segment points be calculated. Between the 3 segment points shall another segment be drawn where the final segment point can be calculated.

Everything is using the same percentage to calculate the interpolation. [6]

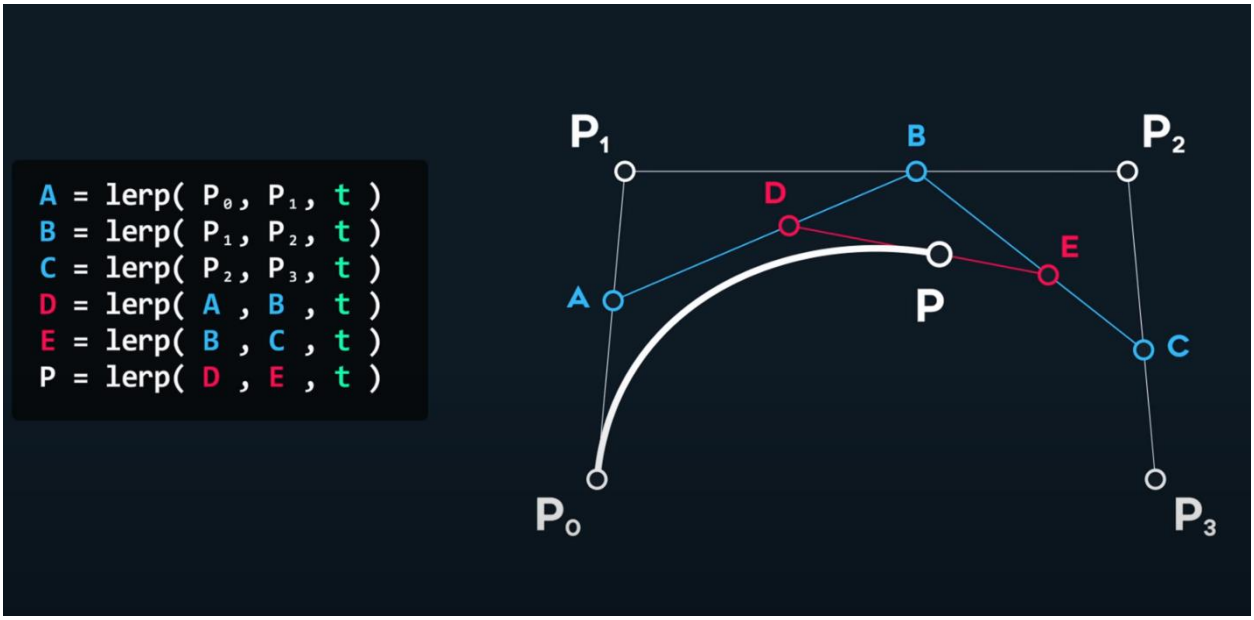


Figure 13 - "De Casteljau's Algorithm" used to create a cubic Bezier Curve [6]

In the picture above P0 is the begin point and P3 is the end point.
P1 and P2 are the control points where P1 has more influence over the first half of the Bezier curve and P2 has more influence over the second half of the Bezier curve.

P is a point that is using t as percentage, a value within the range of 0 and 1, to decide where its position is. It is the intention that, using a small enough interval, it will form this smooth curve.

The points P0 and P3 will be part of a continuous spline, each point having their personal P1 and P2 to form a continuous motion. This will be helpful in the making of the spline-based terrain features. [6]

4.2. BERNSTEIN POLYNOMIAL FORM

The Bernstein Polynomial Form is a way to simplify the formula as all 4 points have a certain weight on the equation depending on what the value of “t” is.

Since the last image shows a clean overview on how to calculate P.

This can be rewritten as the following:

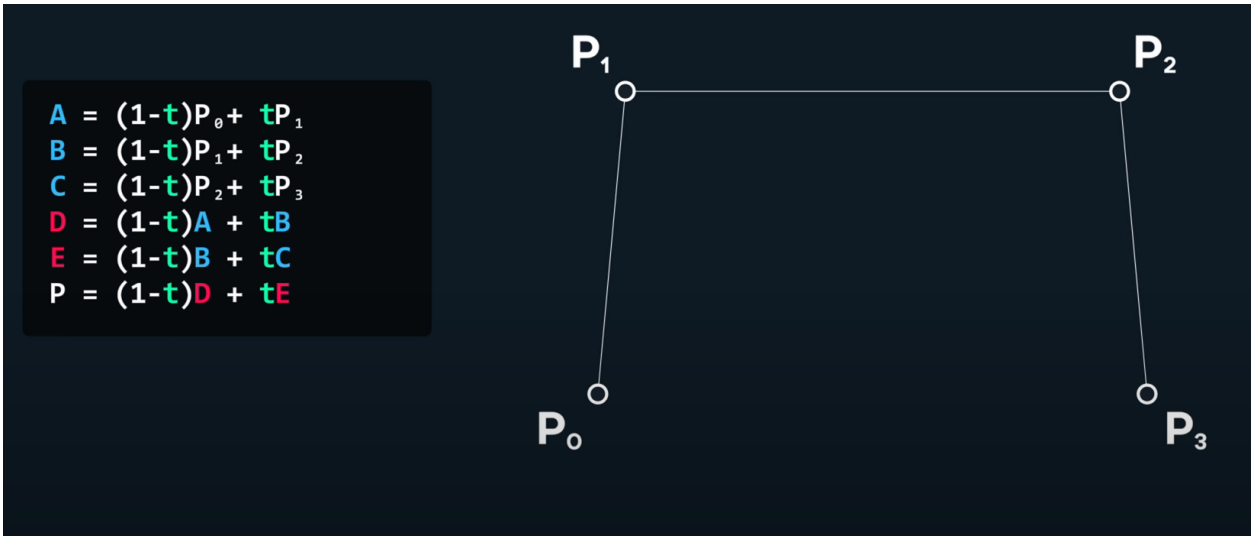


Figure 14 - Writing the lerps functions as math [6]

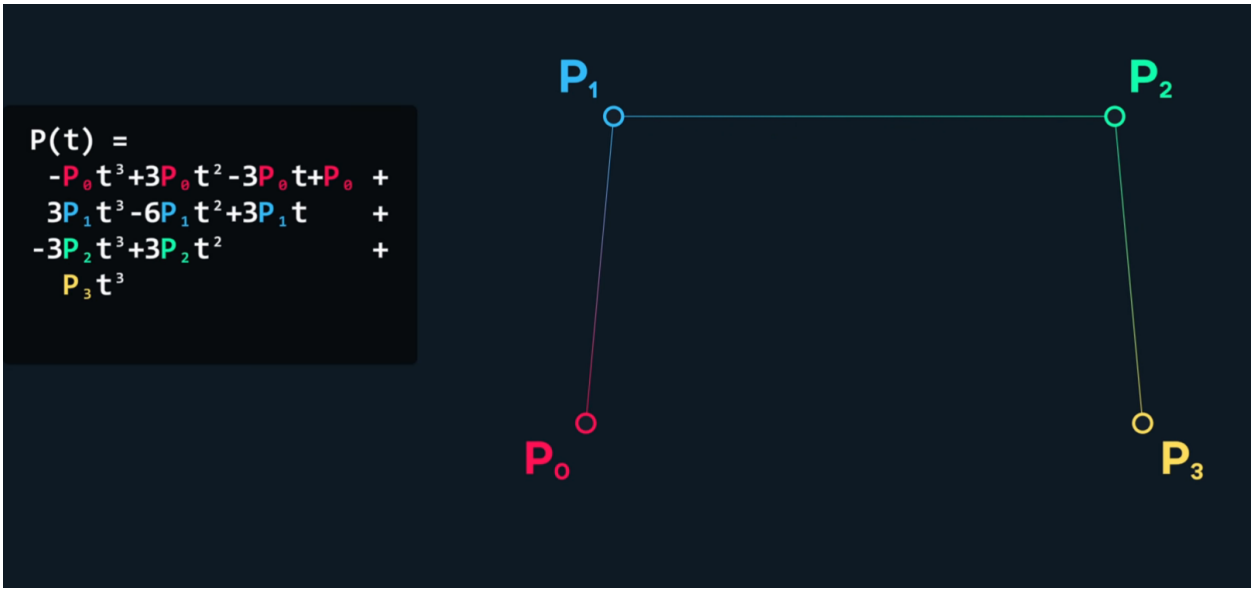


Figure 15 - Filling in all the data into the function of "P(t) =" [6]

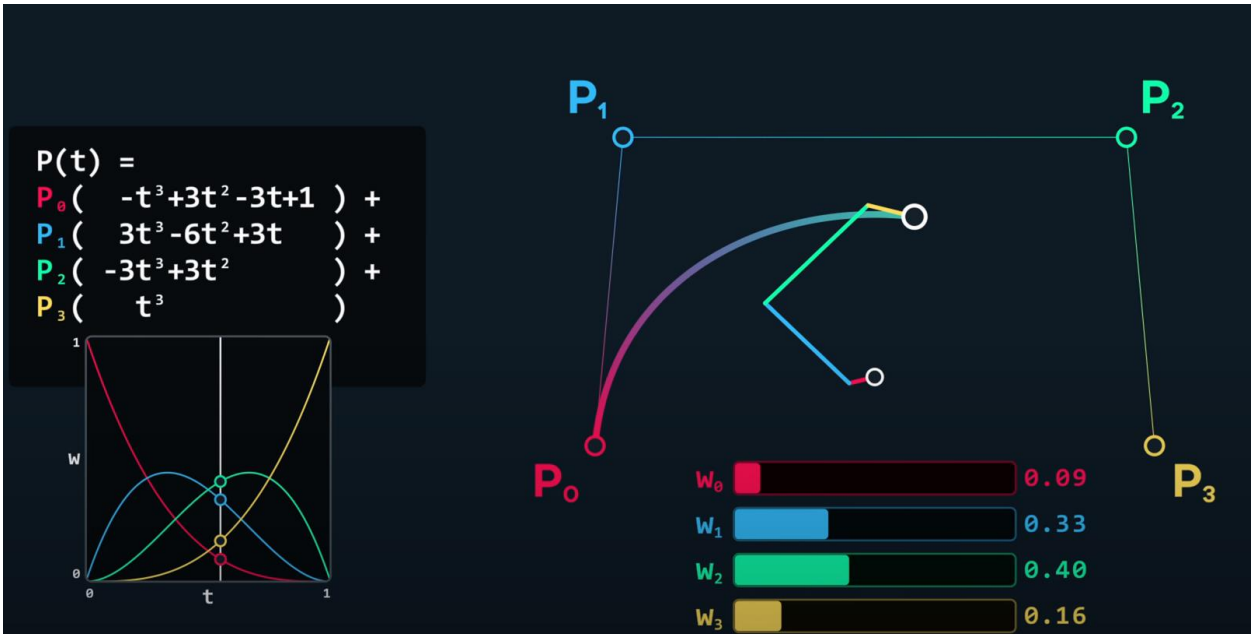


Figure 16 - Bernstein Polynomial Form [6]

The Bernstein Polynomial form will be the one used in the project. [6]

4.3. TANGENTS AND NORMALS

The final data that is required is the tangent direction and normal vector.
Calculating first derivative of the Bezier curve is the same as calculating the velocity or tangent direction, and the normal is the tangent direction rotated 90 degrees. [6]

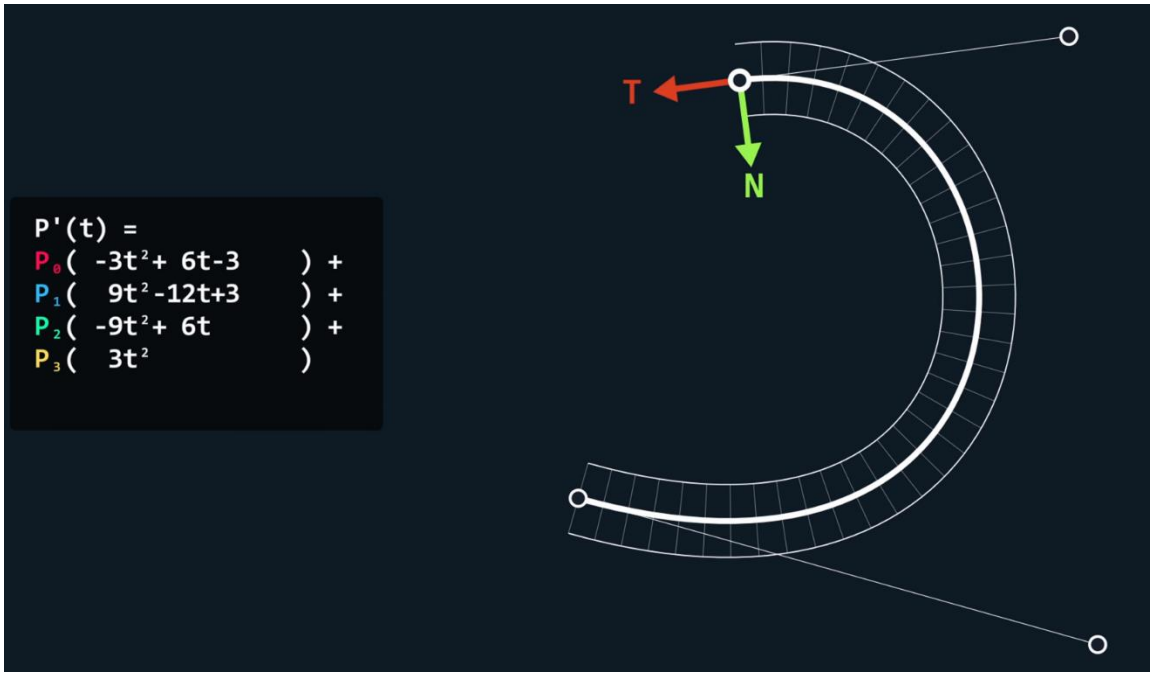


Figure 17 - First derivative (velocity or tangent direction) [6]

CASE STUDY

In this section, we shall go a little more into depth about the execution of the application.
Both going over the implementation and how I think they have done it.

1. COMPARISON WITH LANDMASS

The project itself was made in Unity 2021.2.5f1 using the 3D Core template.
Landmass on the other hand is, as of writing, still an experimental tool in Unreal Engine 4.27.

Landmass uses a system where the terrain is changed based on the heightmap of the terrain, but they do it entirely inside of a material blueprint. Making the project inside of Unity's Universal Render Pipeline was an option that was considered, but just making use of the heightmap of Unity's internal terrain component seemed better.

Unity's Universal Render Pipeline does make the graphical side look more like the Unreal Engine since it is a node based graphical representation of coding materials. However, in Unity there are certain nodes that don't have the same functionality as the ones in the Unreal Engine. I choose to do the generation of the terrain on the CPU side, as the tool needs to be fully accessible in the engine and initial trials indicated that handling data was easier in that way. Thus, it was treated more like an engine tool rather than a shader.

Making a mesh out of the terrain was considered as it would have some interesting properties.
But since it needs to be somewhat performance-friendly, as it needs to be able to change when the parameter does. And because Unity's internal terrain functionality gives access to the heights and texture of the terrain. We shall make use of the internal terrain functionality to make it a bit more performant. It won't be as performant as it would have been on the GPU, but it is better than recreating a mesh every time it updates.

2. A SINGLE TERRAIN LAYER USING PERLIN NOISE

The first part of the tool was to create a basic terrain generator. The generator needs to be able to create realistic looking terrain to a certain degree. This is necessary as we at least have something to fall back on.

The basic generator is making use of a noise generator, in this case it is the Perlin Noise version as Unity already provides that function for us. [7]
In figure 18 you can see only Perlin Noise being used to influence the terrain height.

It uses a seed for the random generator of the noise generation, which makes it so the randomness of values can be somewhat controlled since the same seed and offset will give the same value every time.

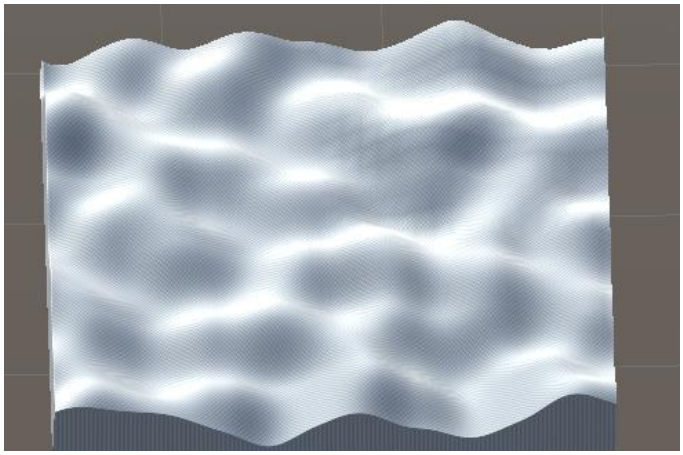


Figure 18 - Unchanged Perlin noise generated terrain

To gain even more control over the random generator you can add a detail scale which allows us to make the differences in the texture larger or smaller.

The 6 images in figure 19 are all generated from the same seed using the same Perlin noise algorithm.
These are however manipulated slightly using the terrain detail scale and the offset of the random generator.
Image A, has a terrain detail scale of 8, the same as image B and image C. The only difference in B and C is the seed offset. Because of the seed offset the generator starts reading the texture from a different coordinate instead of the default (0,0).

The terrain detail scale gives us the feeling as if we are zooming in or out. The greater the terrain detail scale is the finer the details are. The images A, D, E, F have all the same seed offset being the default (0,0). But image A has a terrain detail scale of 8, image D has the scale of 2, E has the scale of 16 and F has the scale of 27.

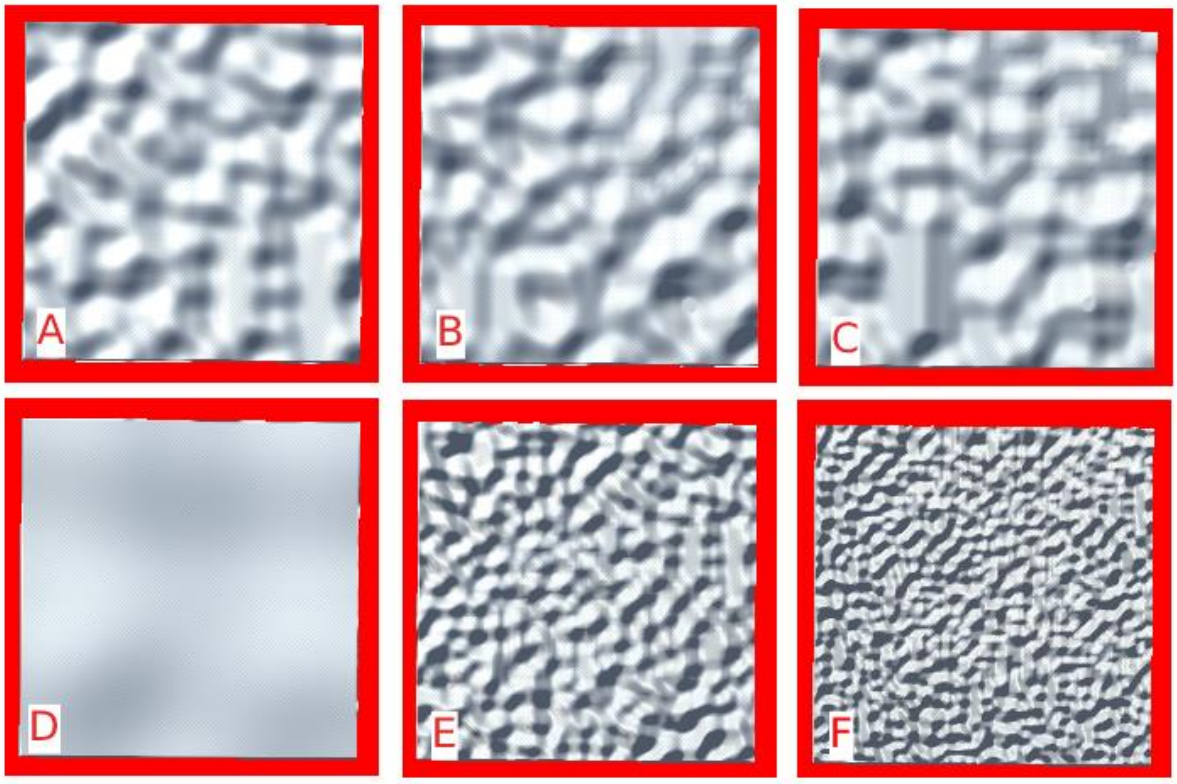


Figure 19 - The differences of Perlin noise generation

The code snippet (figure 20), shown on the next page, shows my method on how to sample a pixel from the terrain with x and z being the coordinates. The result will be used as the y coordinate of that pixel.
The variables called offsetX and offsetZ refer to the seed offset, and the width and the height refer to the width and height of the terrain. [7]

```
float CalculateHeight(int x, int z)
{
    float xCoord = (float)x / _width * _terrainDetailScale + _offsetX;
    float zCoord = (float)z / _height * _terrainDetailScale + _offsetZ;

    return Mathf.PerlinNoise(xCoord, zCoord);
}
```

Figure 20 - Code snippet of Perlin noise sampling [7]

If done with every pixel of the terrain, then we have our first layer of Perlin noise.
Which can be more refined by using octaves. In figure 21 the same terrain is being used with the same offset and seed, the only difference is that image A uses 6 octaves, image B uses 12, image C uses 4 and image D uses 20.

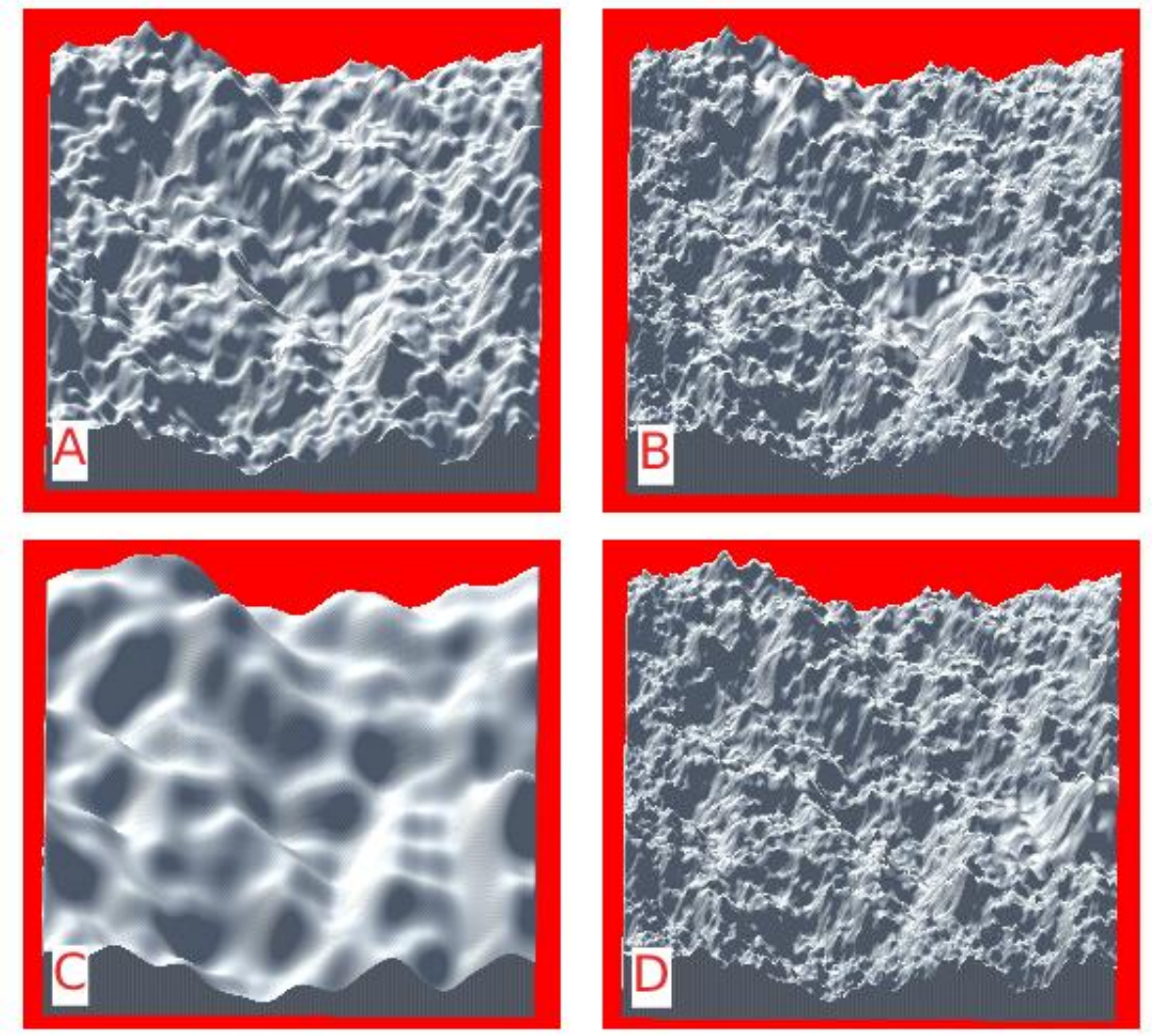


Figure 21 - Different amounts of octaves in the terrain

The differences seen in the use of octaves are increasingly less noticeable, because every smaller octave uses persistence and lacunarity at a lower frequency.

The persistence determines how much influence the layers lower have over the end result.
The lacunarity determines the scale of the lower layers as they keep being multiplied.

Figure 22 shows how the frequency is the same as lacunarity to the power of the current octave.
The amplitude, also known as the height difference, gets smaller and smaller because persistence is a value between 0 and 1.

```
virtual public float[,] GenerateHeights()
{
    float[,] heights = new float[_width, _height];
    for (int x = 0; x < _width; x++)
    {
        for (int z = 0; z < _height; z++)
        {
            float amplitude = 1f;
            float frequency = 1f;
            float noiseHeight = 0f;

            for (int o = 0; o < _octaves; o++)
            {
                float xCoord = ((float)x / _width) * frequency * _terrainDetailScale + _offsetX;
                float zCoord = ((float)z / _height) * frequency * _terrainDetailScale + _offsetZ;

                float perlinValue = Mathf.PerlinNoise(xCoord, zCoord) * 2 - 1;

                noiseHeight += perlinValue * amplitude;
                amplitude *= _persistence;
                frequency *= _lacunarity;
            }

            heights[x, z] = noiseHeight;
        }
    }
    return heights;
}
```

Figure 22 – Code snippet of generating heights using octaves [7]

In figure 23 the persistence is the only difference between the four terrains.
They all use 6 octaves, have a terrain detail scale of 1.02 and use 2.1 for lacunarity.
With image A having a value of 0.397, image B having 0.595, image C having 0.701 and image D having 1.
In the application persistence is a slider between 0 and 1.

The closer the persistence is to 0 the flatter the terrain is, the higher the persistence is the more chaotic and rockier it becomes.

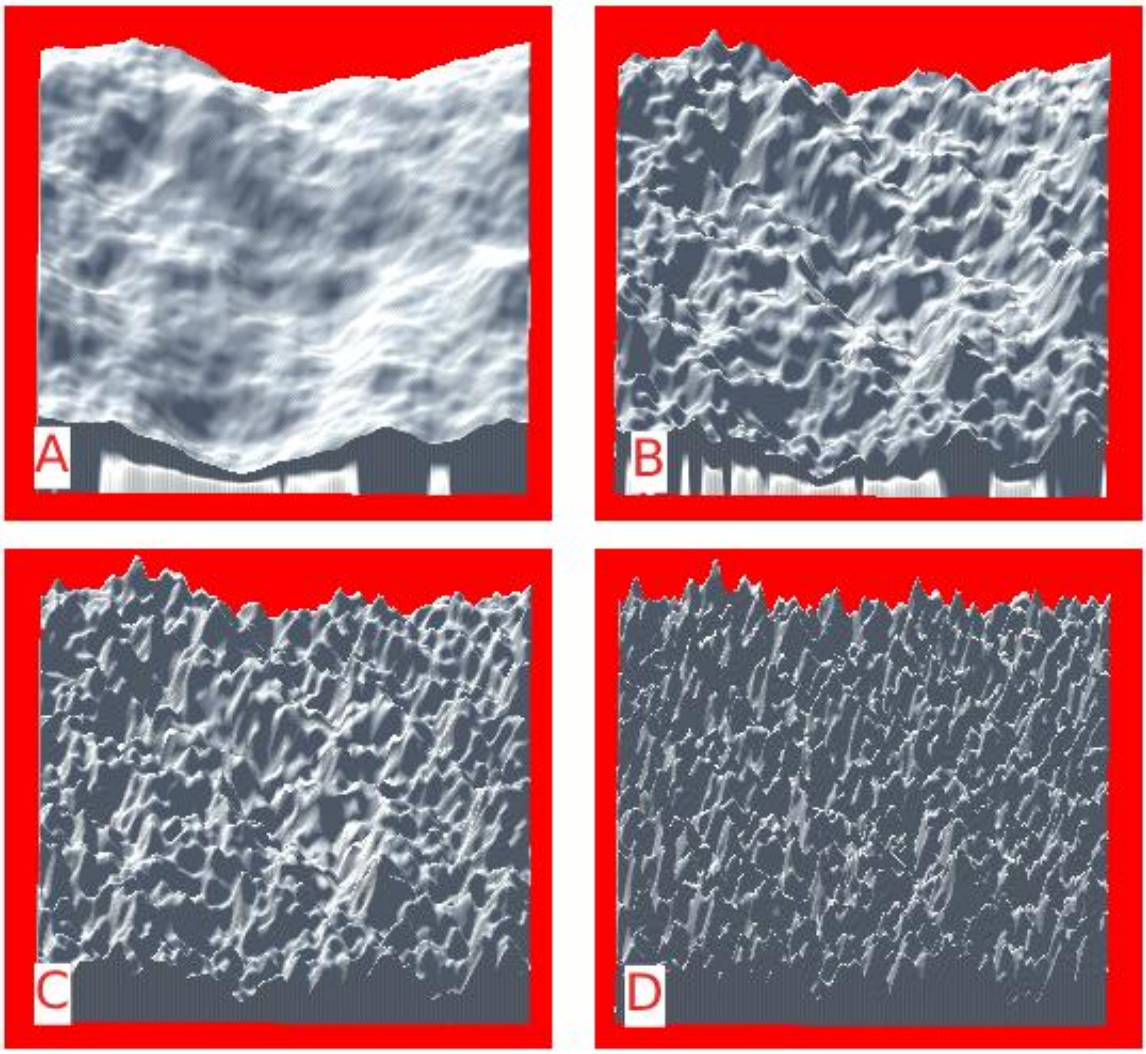


Figure 23 - Differences in persistence

The lacunarity refers to a gap or pool, the higher the lacunarity the smaller the gaps become.
In figure 24, the terrains have the same parameters except for the differences in lacunarity.
They all use six octaves, have a terrain detail scale of 1.02 and use 0.5 for persistence.
The lacunarity in image A is 2, image B has 2.3 lacunarity, image C has 3 lacunarity and image D has 5 lacunarity.

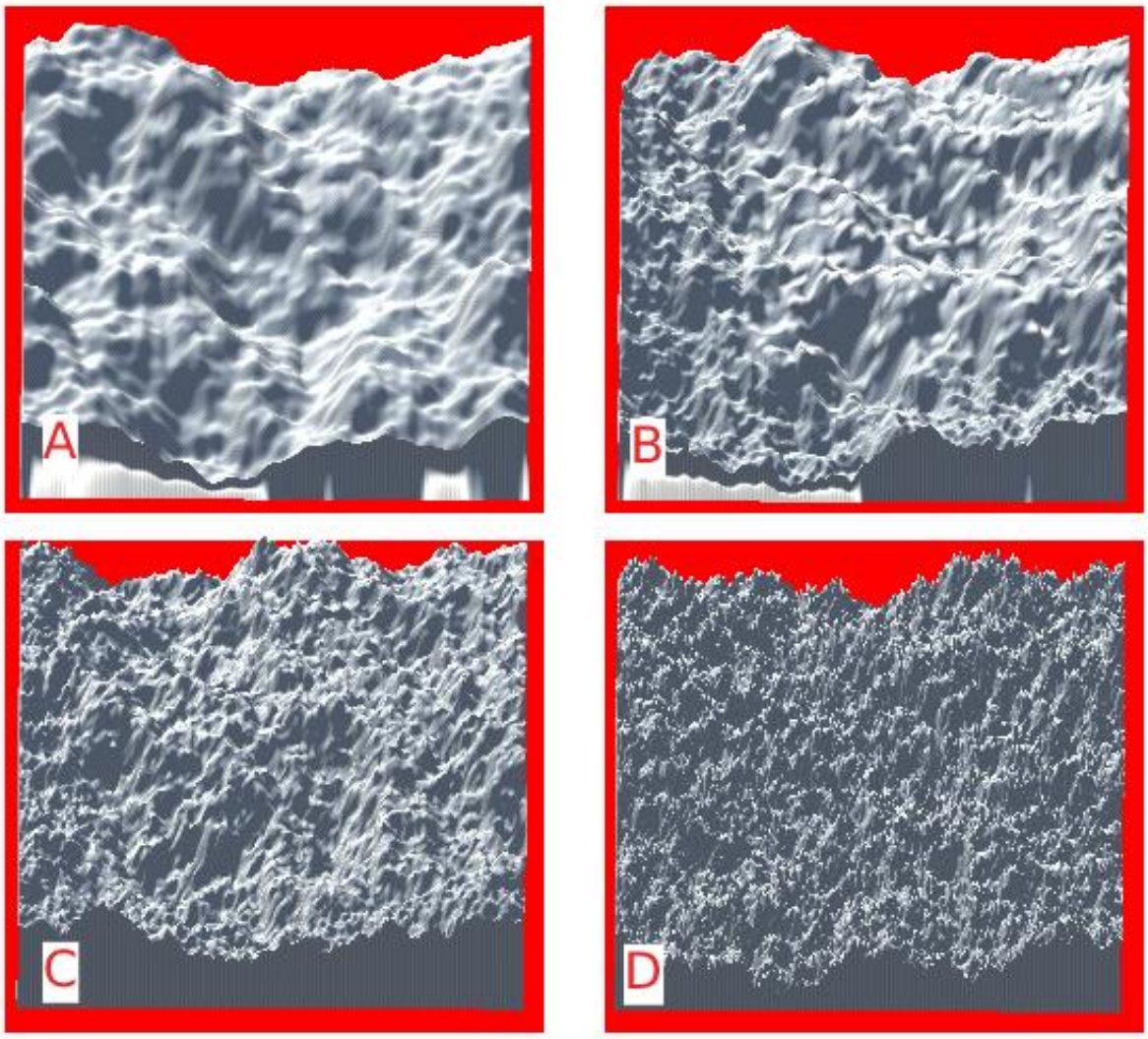


Figure 24 – Differences in lacunarity

3. LAYER BASED TERRAIN TOOL

We can combine these layers of noise together to create a bit more controlled noise, but since there wasn't really a lot of people who talked about it this is where I start taking inspiration from previously used techniques.
In the application I tried two ways of combining the layers together.
What both of them have in common is that they force both terrain layers to be on the same width and height.
On top of that both of them add the depths of the layers together and make that the terrain's new depth.
By making the depth higher Unity's terrain component won't cut off parts of the top of the mountains.

3.1. ADDITIVE LAYERS

In the additive version we just add the generated heights together and put it into the height map. The terrain option is not that natural looking compared to the lerped variant as most of the time you have these kinds of outcomes. They look interesting if you are looking for rocks poking out of the water underneath a cliff, but it isn't really the wanted outcome.

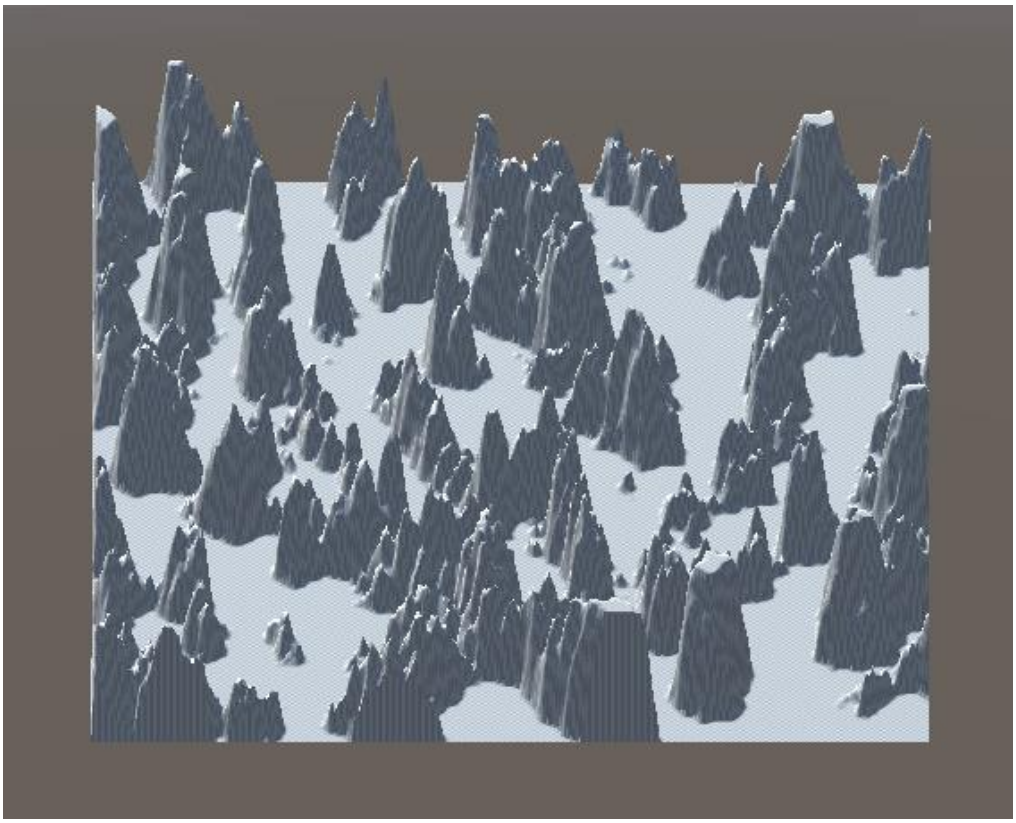


Figure 25 - 2 Layers combined using just additive layers

3.2. LERPED LAYERS

Just like the additive version, the heights get added up but after they are all added up the minimum and maximum value is searched. With those values we can inverse lerp the terrain and get a more coherent landmass. [7]
The inverse lerp was normally meant to be used after a single layer has finished using his octaves, but it has some nice coherent. The two layers can have any value different from each other and sometimes it does take some time to get it right but if a layer is dragging another one down, you can just disable the layer.

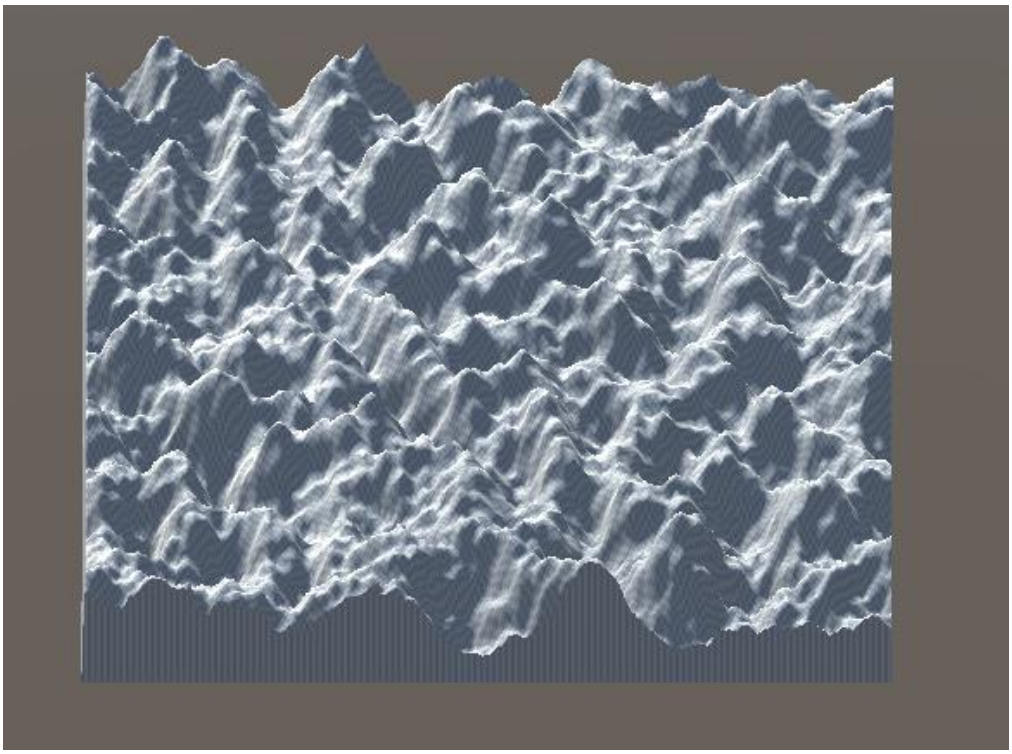


Figure 26 - The same terrain but with inverse lerped values

In figure 27 the 2 child game objects (both having a terrain layer with parameters).

The first terrain has a depth of 25, a terrain detail scale of 12.77, uses 12 octaves, 0.272 persistence and 3.56 lacunarity. Also, qua random generator the default offset is used.

The second terrain has a depth of 27, a terrain detail scale of 10.44, uses 13 octaves, 0.097 persistence and 2.81 lacunarity. Here however the offset is 2.34 for the X and 0.88 for the Z.

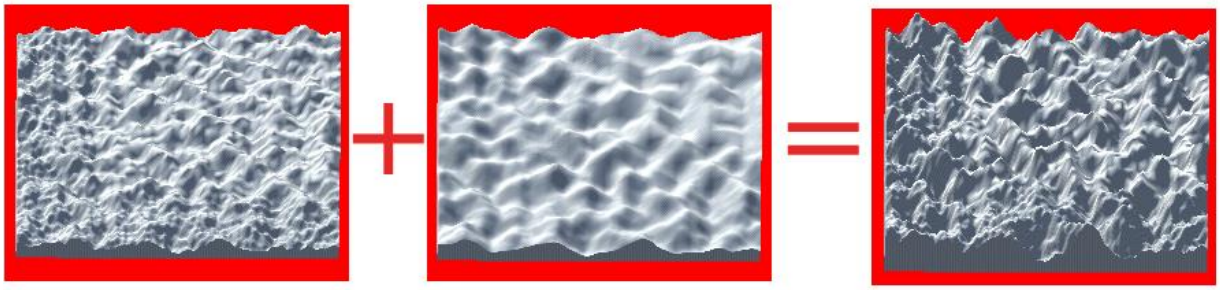


Figure 27 - the 2 helper layers forming the end result

4. SPLINE BASED TERRAIN TOOL

Initially I wanted to do the same approach as was said by Arran Langmead. Mainly that I start with a spline and make a mask out of it as shown in figure 8. This already seemed to be a bit of a challenge as I had no idea how he wanted to implement a texture mask.

The spline was connecting the Bezier curves that we explain before in the related work section part 4.

Eventually I opted to go without the mask and later on also without the jump flood as I couldn't really figure out how to use them to obtain a distance field.

The signed distance field is executed rather crude and most of the times when I try to update the spline the performance could've been better too. Although only when it is drawing it on a texture like shown below.

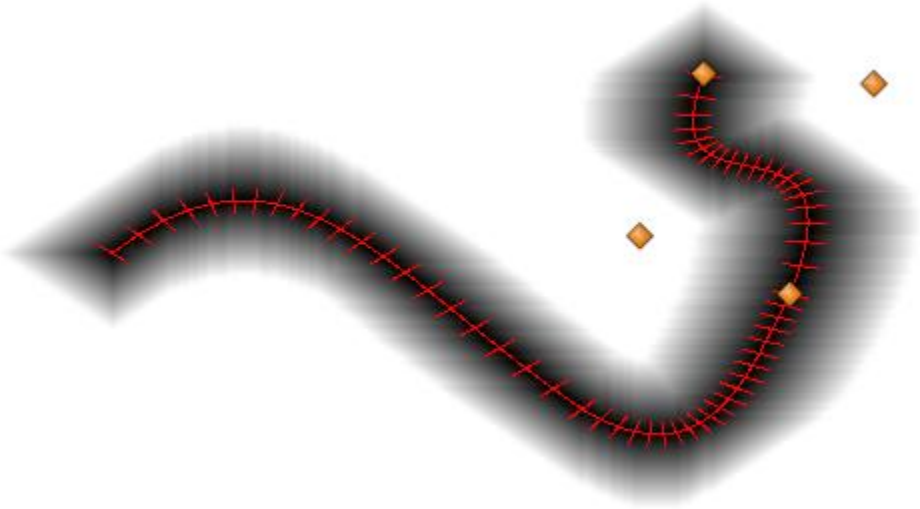


Figure 28 - distance field around spline

The distance field is created by first making a grid resembling the terrain grid.
Then filling it all in with the max value of the float, so we know that those pixels still need to be redone.
Next we are going over every pixel of the terrain, if they don't have the max value of the float then we can go look at its neighbors and whether or not they have the max value of the float or a lower value than the value you would get if you were to step from this tile. Keep doing that until every tile has been filled.

Once this is done you get a ravine-like shape that carves through the terrain.
For a more mountain-like terrain we need to take the height and subtract that value from 1.

```
while (ThereAreStillSomeEmpty(ref heights))
{
    for (int x = 0; x < _width; x++)
    {
        for (int z = 0; z < _height; z++)
        {
            float pixelHeight = heights[x, z];
            if (pixelHeight != _fillerValue)
            {
                float newHeight = pixelHeight + _StepSize;

                if (x - 1 >= 0 && (heights[x - 1, z] == _fillerValue || heights[x - 1, z] > newHeight))
                {
                    heights[x - 1, z] = newHeight;
                }
                if (x + 1 < _width && (heights[x + 1, z] == _fillerValue || heights[x + 1, z] > newHeight))
                {
                    heights[x + 1, z] = newHeight;
                }
                if (z - 1 >= 0 && (heights[x, z - 1] == _fillerValue || heights[x, z - 1] > newHeight))
                {
                    heights[x, z - 1] = newHeight;
                }
                if (z + 1 < _height && (heights[x, z + 1] == _fillerValue || heights[x, z + 1] > newHeight))
                {
                    heights[x, z + 1] = newHeight;
                }
            }
        }
    }
}
```

Figure 29 - Code snippet of naive approach distance field

```
for (int x = 0; x < _width; x++)
{
    for (int z = 0; z < _height; z++)
    {
        heights[x, z] = (1f - heights[x, z]);
    }
}
```

Figure 30 - Code snippet of one minus

As shown in figure 31, the spline guides the terrain where to go or stay away from. The terrain layers can influence the way the spline interacts with the other terrain generation but in general it follows its direction.
As shown with the mountain on the left picture, the spline guides the mountain even when the other layers disrupt it slightly.

The ravine is also a nice example on how those other layers can make the walls more interesting compared to what you see in figure 32.

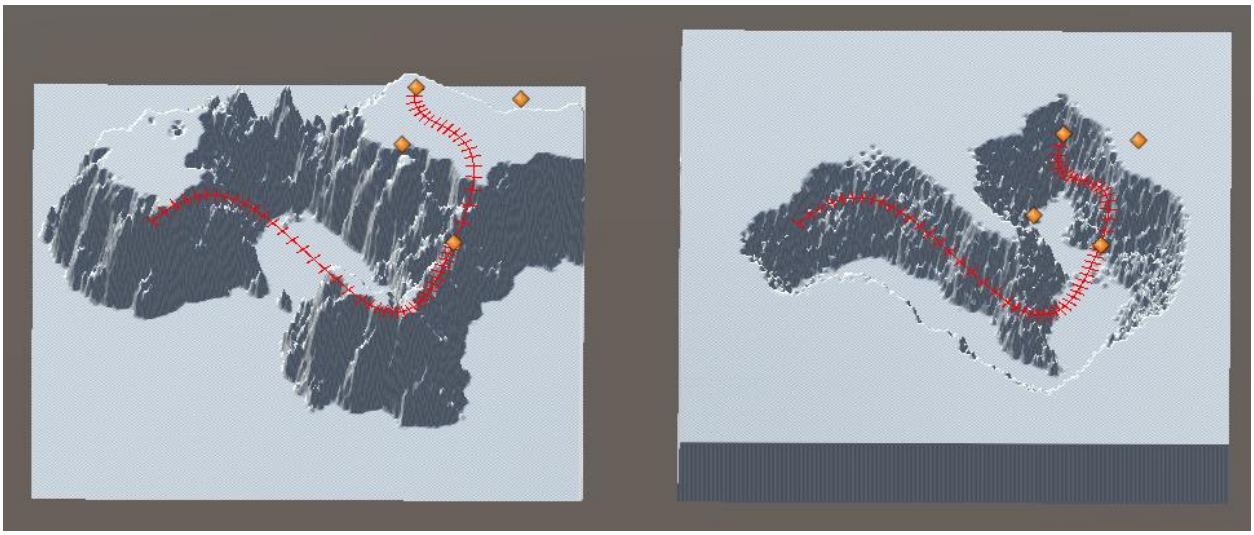


Figure 31 - Mountain and ravine guided by spline

HARDWARE AND SOFTWARE

In this section, we shall present the results of the experiments performed. We will have a look at how close we got to the real deal.

Throughout the entirety of the study the same computer was used. This laptop has the following specifications:
- Intel(R) Core (TM) i7-7700HQ CPU @ 2.80GHz
- 16GB DDR4 DRAM 2400 MHz
- Intel(R) HD Graphics 630 with 8GB VRAM
- NVIDIA GeForce GTX 1060 with 6GB VRAM

The project itself was made in Unity 2021.2.5f1 using the 3D Core template.

DISCUSSION

In this section we will discuss what went wrong or what I would've liked to do different.

First of all, it would be nice to make use of the Fortune's algorithm for the jump flood. Personally, I couldn't figure out how to integrate it, further research and testing is required.

I would've liked to make it possible to create a terrain in the hierarchy using right click.
Because I was focused on staying as close to Arran Langmead's explanation, I may have drifted off causing me to eliminate certain techniques since they weren't fitting compared to the methods he presented.

CONCLUSION & FUTURE WORK

In this section, we will talk about what we learned and what we could potentially add or change.

In this paper we aspire to implement a terrain generation tool in Unity based of the Landmass terrain tool used in Unreal Engine. By doing this I acquired the basic knowledge about Perlin noise, procedural terrain generation as a whole and Bezier curves. Further deep delving into these points should result in even more usability and flexibility for the tool.

The main findings are that the implementation is a good start but needs further development.

In the future I plan to continue this research in the form of adding fortune's algorithm and adding biomes for more specific generation.

BIBLIOGRAPHY

[1] R. Fischer, P. Dittmann, R. Weller and G. Zachmann, "AutoBiomes: procedural generation of multi-biome landscapes," University of Bremen, Bremen, Germany, 2020.

[2] A. Langmead, "Building Worlds with Landmass," Unreal Engine, 6 January 2021. [Online]. Available: https://www.youtube.com/watch?v=GwJiN8LLnQI. [Accessed 16 December 2021].

[3] B. Douglas, "The Jump Flood Algorithm | Visualized and Explained," 17 May 2021. [Online]. Available: https://www.youtube.com/watch?v=A0pxY9QsgJE. [Accessed 6 January 2022].

[4] K. Perlin, "An Image Synthesizer," 1985.

[5] J. Talle, "Cubic noise," 31 October 2017. [Online]. Available: https://jobtalle.com/cubic_noise.html. [Accessed 16 December 2021].

[6] F. Holmér, "The Beauty of Bézier Curves," 19 August 2021. [Online]. Available: https://www.youtube.com/watch?v=aVwxzDHniEw. [Accessed 16 December 2021].

[7] Brackeys, "GENERATING TERRAIN in Unity - Procedural Generation Tutorial," Brackeys, 24 May 2017. [Online]. Available: https://www.youtube.com/watch?v=vFvwyu_ZKfU. [Accessed 2 January 2022].

[8] R. Szeliskit and D. Terzopoulos~, "From Splines To Fractals," Computer Graphics, 1989.

[9] A. Mezhenin and A. Shevchenko, "Optimization of Procedurally Generated Landscapes," ITMO University, St. Petersburg, Russia, 2020.

[10] M. O. E. Letters, "Voronoi Diagrams and Procedural Map Generation," 14 January 2019. [Online]. Available: https://youtu.be/3G5d8ob_Lfo. [Accessed 6 January 2022].

[11] Stanford, "Polygonal Map Generation for Games," 4 September 2010. [Online]. Available: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/. [Accessed 6 January 2022].