Masaryk University
Faculty of Informatics

Multi-Fractal Terrain Generation

Master's Thesis

Bc. Jakub Kříž

Brno, Spring 2019

Masaryk University
Faculty of Informatics

Multi-Fractal Terrain Generation

Master's Thesis

Bc. Jakub Kříž

Brno, Spring 2019

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Jakub Kříž

Advisor: doc. Fotis Liarokapis, PhD.

Acknowledgements

I would like to thank my supervisor Fotis Liarokapis for his guidance and motivation and my parents and friends for supporting me throughout my studies.

Abstract

This thesis evaluates methods used in procedural generation of terrain. The main focus is on pseudorandom generation of terrain that is continuous and possible to generate over infinite distances, procedural texturing with custom shaders and erosion of the terrain to create realistically looking environment. For practican part of the thesis a multi-fractal noise generation method for generating basic terrain consisting of multiple meshes was implemented and expanded with droplet erosion method to smooth the terrain after generation. Another expansions were generation using partitioning, simulating simple biomes, discrete level of detail (DLOD) and generation of lakes and caves to create more diverse terrain. The results are evaluated based on the efficiency and effectivity of the algorithms used.

Keywords

Multi-fractals, Interpolation, Random number generation, Noise, Procedural random terrain generation, Procedural texturing, Partitioning, Level of detail, Droplet erosion.

Contents

List of Tables

List of Figures

1 Introduction

The diverse world that surrounds us has always been in the spotlight of mankind. The numerous attempts to describe seemingly random phenomena of the world date back to the origin of the world and human thought. Since the discovery of fractal geometry in the 1970s these attempts strove not only to describe, but also to recreate the visual appearance of the world.

Modern 3D games, simulations or even animations often need highly detailed terrain for the purpose of visualizing scenery or possible obstacles and landing zones in flight simulations. The main goal is to create visually realistic terrain, which has realistic properties like heights, slopes and cracks in the terrain. Doing this by hand is slow, costly and creates large files. All of these problems can be minimized by using procedural generation. And even though the majority of higher budget simulations and games usually use both the procedural terrain and handmade terrain for some sections or to add details to the procedural parts [1] [2], there is a clear trend in games and simulations to generate as much of the world as possible.

There are some projects using only procedural generated worlds [3] which can be used as an example that it is possible. Some projects take it even further and generate thousands of years of the history of the world [4].

Over the last years procedural generation became an important part of development of virtual worlds used in games, simulators, simulations and other areas. Its technology is still fairly new and therefore there is new research done all the time to improve and optimize the methods used and to develop new methods focusing on the shortcomings of the older methods.

1.1 Aims and objectives

The aim of this thesis is to investigate and implement some of the computer graphics algorithms used in procedural generation of randomized realistically looking terrain. Main focus is on ways to achieve multi-fractal terrain, describing and using hydraulic erosion to smooth the terrain, implementing texturing based on the terrain properties

and research and implement the possibility of adding detail to the terrain by generating lakes and caves. It also contains critical evaluation of the implemented methods and the project as a whole.

The practical output of this thesis is on-fly multi-fractal terrain generator based on the unity game engine using C#. This generator is expanded by previously listed methods allowing the user to customize the generation based on multiple parameters.

1.2 Thesis structure

The thesis is divided to four main chapters.

Background chapter focuses on the research of modern methods used in procedurally generated terrain, the history of their development and their comparison. This part will be used as an overview of available and usable methods for dealing with certain problems of procedural generation.

Implementation chapter gives an insight to the practical part of the thesis by describing the use of the methods that were researched in the *Background* chapter and the design decision that were made in pursuit of the best results.

Results chapter evaluates the implemented algorithms. It is also discussed how they hold up based on the requirements and how they fit to the project. This chapter also critically evaluates the project and the accomplished results from the efficiency and effectivity standpoint. This means evaluating the speed of the used algorithms and the possibility of recreating the real world terrain.

Conclusion chapter compares the results with other projects and includes a section on possible future improvements and expansions.

2 Background

In this chapter we will go through the methods used for generating multi-fractal terrain and other methods that are commonly used in terrain generation.

2.1 Interpolation

People perceive the world around them as continuous, even though it does not necessarily need to be so. To simulate this perception in computer programs and games which work with discrete signals, computer graphics need a way to display things described by discrete values as continuous shapes or textures. These continuous shapes and values are calculated from discrete values. By interpolating between these values, a position of a point or a color of a pixel that is slightly moved from the given values can be calculated.

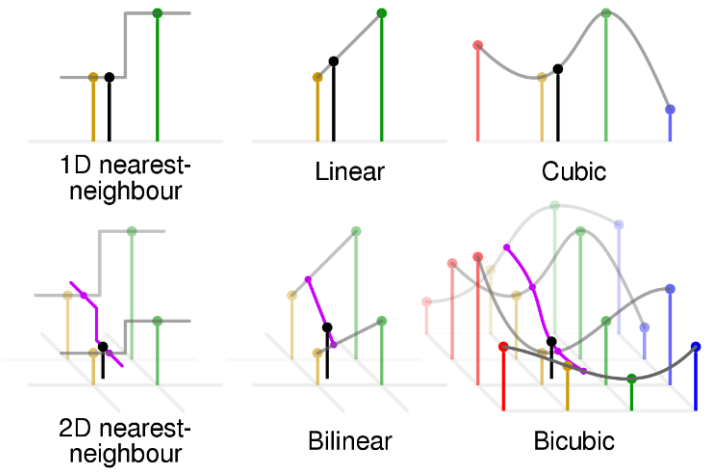

Figure 2.1: Interpolation methods [5]

2.1.1 Linear interpolation

The simplest type of interpolation is the linear interpolation. Linear interpolation is useful if we need to get the value of a point that lies

between two defined discrete values. This is done by drawing a straight line between the two discrete values.

By using the y and x-coordinates of the neighbouring points and one coordinate of the coordinates of the new point we can calculate the other coordinate as follows:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0} \tag{2.1}$$

$$y = y_0 + (x - x_0) * \frac{y_1 - y_0}{x_1 - x_0} \tag{2.2}$$

In the following interpolation methods, a modified formula will be used for better readability, where t encapsulates the calculations of the distance of the new point x-coordinate from the neighbouring points when calculating y.

$$y = y_0 + t(y_1 - y_0) \tag{2.3}$$

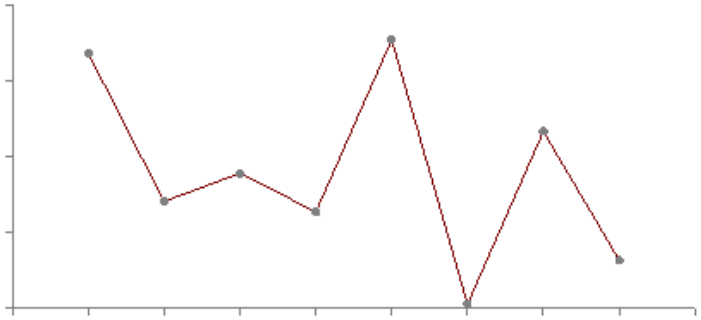

Figure 2.2: Linear interpolation [6]

The problem with linear interpolation is the use of straight lines that make the transitions very sharp and that can look very artificially. It can be appropriate in some cases, but it can be perceived as inaccurate in the others. The advantage of this method is that the formula is very simple and fast to calculate in comparison with other methods used for interpolation.

2.1.2 Cosine interpolation

To solve the problem of the sharp lines when using linear interpolation, new methods were developed for the purpose of smoothing the interpolation. One of the simplest if not the simplest method is cosine interpolation. This method uses cosine function suitably oriented to connect the two discrete values.

$$t_{cos} = \frac{1 - cos * (\pi * t)}{2} \tag{2.4}$$

$$y = y_0 + t_{cos}(y_1 - y_0) \tag{2.5}$$

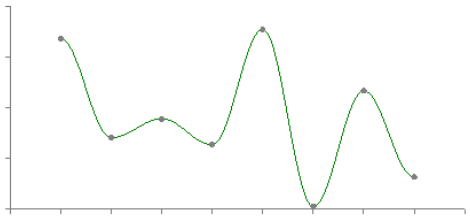

Figure 2.3: Cosine interpolation [6]

2.1.3 Cubic interpolation

Cubic interpolation deals with the continuity problem of the cosine interpolation. For that reason, it has to take into account the neighbouring points of the two points that are being interpolated, making the function more complex.

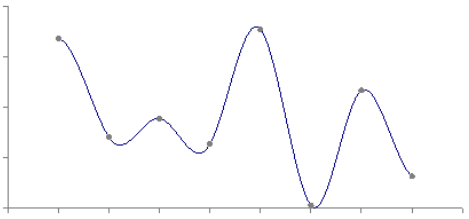

Figure 2.4: Cubic interpolation [6]

2.1.4 Bilinear interpolation

Bilinear interpolation is used for getting a value of a point surrounded by four points with known value. The existence of the four values that can encapsulate the value given a prerequisite of this method. It uses the linear interpolation described in the section 2.1.1. For interpolation between four values, the bilinear interpolation uses three iterations of the linear interpolation.

The method calculates the value of the given point based on the offset of that point from the values of the four known values of the node. Each iteration takes into consideration the x axis of the points with the same y-coordinate and the last one uses the values computed in the previous two iteration that have the same x-coordinate.

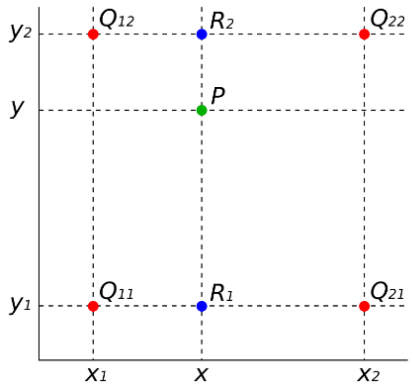

Figure 2.5: Bilinear interpolation [5]

Assuming that we know the values of a function f in four points (Q_{ij}) of a node that encapsulate the point whose value we are trying to calculate, we can calculate the value as displayed in the following figures:

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \tag{2.6}$$

$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \tag{2.7}$$

$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \tag{2.8}$$

2.2 Noise generation

Noise generation is used all around us. It can be used for generating better controlled pseudo-random values, for creating textures or even geometric shapes like terrains.

This section will investigate modern methods used for generating pseudo-random noise. In terrain generation this noise is usually used to determine the height of a pixel or a point that describes the terrain.

2.2.1 Heightmaps

Heightmaps are used in procedural generation to save, access and modify the height values of pixels or control points of a terrain from which meshes are generated from. It can be understood as a 2D projection of a 3D space, typically created from a plane, where each point that is projected has its third, y-coordinate, that is necessary for transferring it to the 3D space saved as a value describing the height of a point.

Apart from the plane-based terrains, heightmaps can also be used in simple shapes like spheres that might be used for generating planets, where the height value represents the distance from the center of an object.

The height map is usually saved as an array of values that is either one-dimensional or two-dimensional. The height value of each point on the map can then be accessed by providing the two coordinates of a point. This simple structure also makes it ideal for algorithms that run in large number of iterations while modifying the height values of neighbouring points, e.g. algorithms used for erosion simulation.

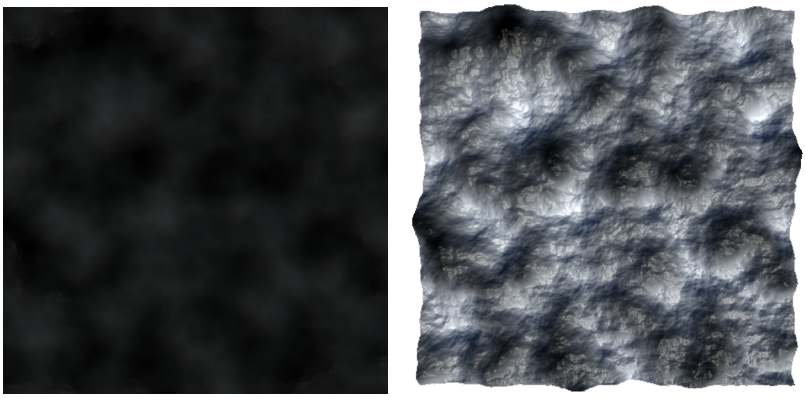

Figure 2.6: 2D noise to 3D terrain

2.2.2 Random number generation

Random number generators are used in many fields of informatics. Although they are called random, software solutions for generating numbers can only generate pseudo-random values based of some other value which is usually time or predetermined seed that can be used if the possibility of recreating the same result is desired.

The most simplistic noise generator would be based on random number generation by generating a random number for each pixel in a texture or a shape. Although simple, this method is not suitable for such tasks because of the discontinuity of the returned values.

2.2.3 Lattice noise

This method divides the space into a lattice where each lattice coordinate gets assigned a pseudo-random value.

The algorithm works as follows [7]:

- First a hash of values is created as a permutation without repetition
- A lattice is created based on resolution, where every lattice point is pseudo-randomly mapped to one of the hash values.

- To get the value of a point with real number coordinates, the algorithm uses bi-linear interpolation using values from its neighbouring lattice points.

For the purpose of controlling the speed of transitioning between values, the method uses *frequency* parameter that multiplies the provided position before calculating its value.

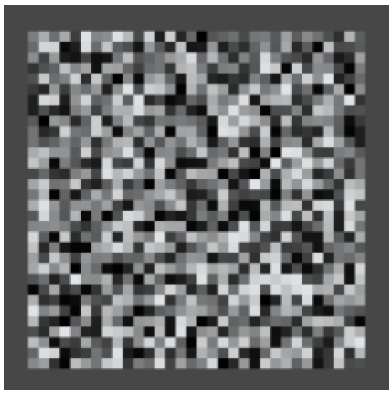

Figure 2.7: Lattice noise [7]

2.2.4 Perlin noise

Perlin noise was introduced by Ken Perlin in 1985 and since then it became one of the most used algorithms for noise generation, because of its simplicity and effectivity. [8]

Fixed value per lattice coordinate was not enough to create noise that would be able to describe natural looking shapes and textures. To solve this issue, Ken Perlin was the first to suggest that the gradients be associated with each lattice coordinate. [9]

Since the linear interpolation produces rather sharp transitions between the neighbouring values, Perlin noise uses a smoothing function:

$$t = 3t^2 + 2t^3 \tag{2.9}$$

This function smooths the curve by using a function that has a first derivative which is zero at both ends. That way the rate of change is always zero at gradient boundaries. [7]

The algorithm works similarly to lattice noise, but uses gradient vectors of the neighbouring lattice points instead of discrete values for interpolation for which it also uses the smoothing function.

For instance, in 3D, noise is determined at point (x, y, z) by computing a pseudo-random gradient at each of the eight nearest vertices on the integer cubic lattice and then doing splined interpolation. [8]

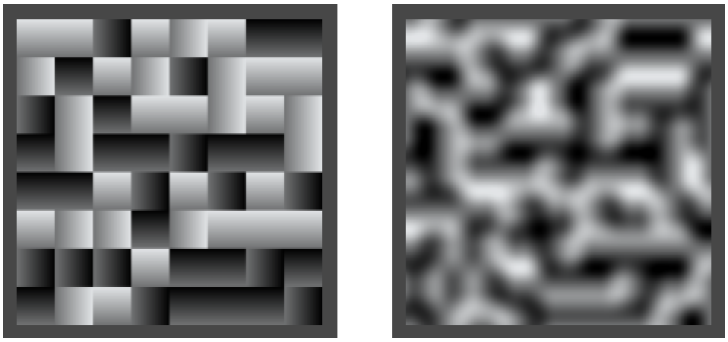

Figure 2.8: Perlin noise gradients (left) and Perlin interpolated values (right) [7]

2.2.5 Simplex noise

Ken Perlin released simplex noise in 2001 as an improvement of his previous Perlin noise algorithm. Targeting the limitations of Perlin noise to create more robust, simpler and faster version. [9]

Simplex noise makes use of the fact that there is no need for hypercube grid and uses the simplest shape possible for the grid cells. In 1D nothing changes since the simplest is still a line. In 2D it uses triangles instead of squares and for 3D tetrahedrons instead of cubes.

This makes the simplex noise overperform Perlin in higher dimensions where the complexity is $O(D^2)$ as compared to Perlin complexity $O(2^D)$, where D is the number of dimensions. To further reduce the complexity Perlin lowered the number of multiplications and their complexity. This was among other things achieved by using numbers with lower count of ones in binary form reducing the multiplication time. [9]

Simplex also uses summation instead of interpolation. Where classical Perlin noise includes sequential interpolations along each di-

mension which significantly increases computational complexity, especially for higher dimensions, simplex noise uses a direct sum of values from each corner. Value of the corner is a multiplication of extrapolation of the gradient and a radially symmetric attenuation function. Radial attenuation is carefully chosen to influence so that the influence from each corner reaches zero before crossing the boundary to the next simplex. [9]

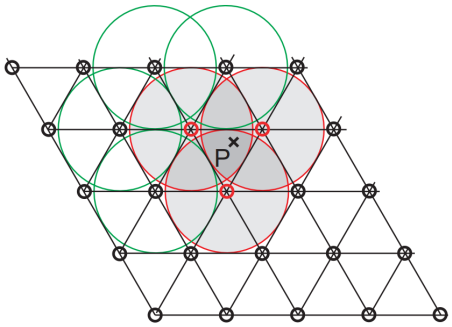

Figure 2.9: Determining point P value in 2D by simplex considering the values of the 3 closest neighbours [9]

Simplex noise also implements new smoothing function.

$$t = 6t^5 - 15t^4 + 10t^3 \tag{2.10}$$

This function is still viable in second derivative, which makes the noise function better suited for the common computer graphics tasks of surface displacement and bump mapping.

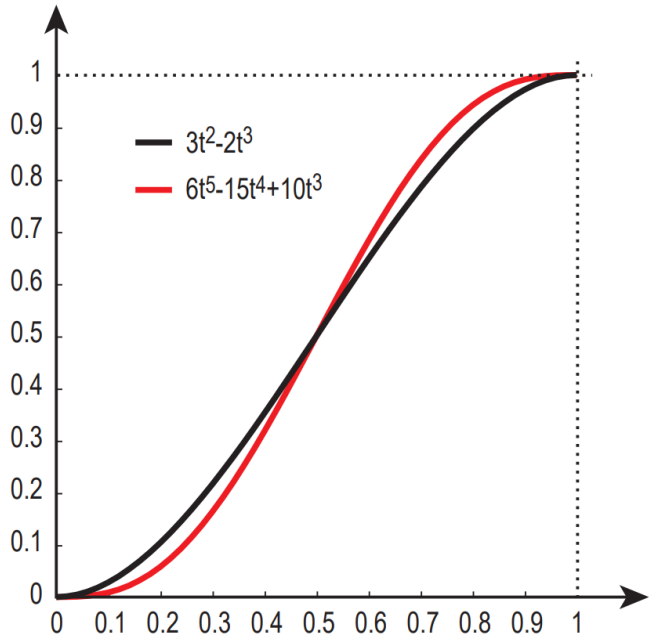

Figure 2.10: Classic Perlin smoothing function (black) vs Simplex smoothing function(red) [9]

2.2.6 Ridged noise

Further modification of the previous noise generating methods can be used for ridged noise. It is produced by returning the absolute value from previous methods that return range of values <-1;1> and multiplying it by -1. By subtracting the result from the terrain values, it creates a shape resembling a river basin in the terrain. By adding it to the terrain, it creates a shape resembling ridges. It can be used to generate more detail, creating rivers, or roads by randomly determining areas for texturing and flattening.

$$RigidNoise(x,y) = Abs(NoiseFunction(x,y)) * (-1) \qquad (2.11)$$

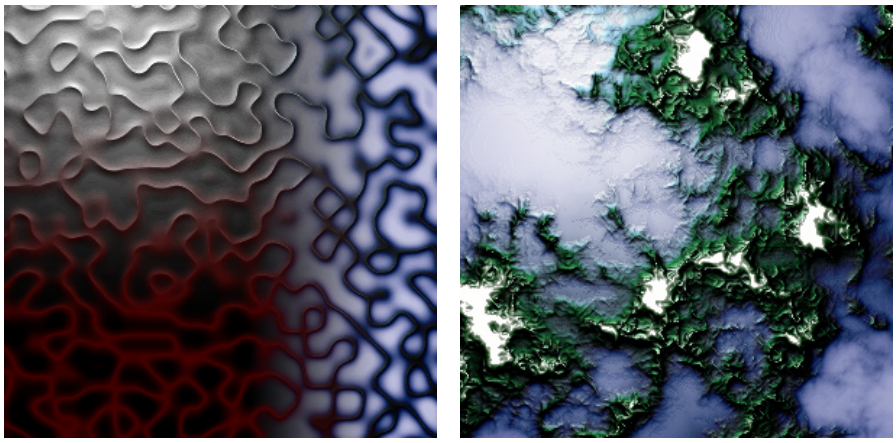

Figure (2.12) Ridged noise on flat surface (left) and ridged noise used to increase detail in terrain (right)[10]

2.3 Simulated fractal noise

Even though simple noise functions are sufficient in some areas, the noise that is created using the above described methods significantly lacks detail when it comes to inspecting the artificially created shapes and objects from a closer distance.

When zooming in on an object in nature, the detail becomes somehow fractal. This means that the shapes that can be described as a tree or a mountain from a greater distance resemble the shapes of branches and rocks from a very close distance.

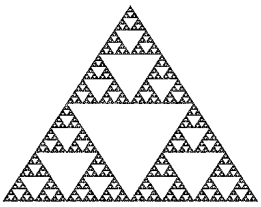

Figure 2.12: Typical fractal shape [11]

The way to simulate this effect with noise-based algorithms is to stuck multiple noise functions on top of each other with each having different frequency and intensity. Each of these noise functions is called octave.

By adding all the octaves together very interesting and fractal-like looking results can be created. These results make the terrain look more natural. Furthermore, there is a possibility to use different methods of noise generation for each octave which can make the result more random and less repetitive.

In practice it is common to use each following octave with double the frequency of the previous and half the intensity which follows the basic principles of fractals. This is also suggested by Benoît Mandelbrot [12]. In his work he discusses frequency as being proportional to intensity in fractals.

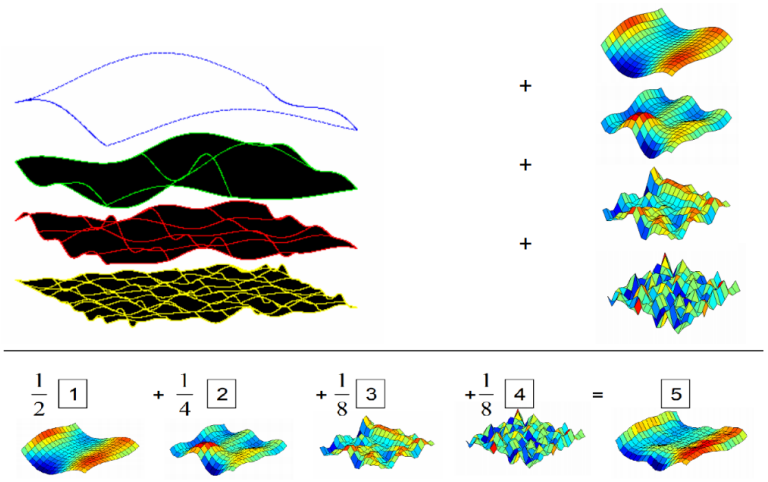

Figure 2.13: Multiple octaves of Perlin noise used to generate fractal terrain [13]

Advantages

- More control over the result - it is possible to use different noise generating methods for each octave, while also setting their intensity and frequency.

Disadvantages

- Each octave adds on complexity of the algorithm.
- Adding octaves results in value distribution moving towards the mean value, so the max and min values are not so common.

2.4 Midpoint Displacement – Diamond-square algorithm

The alternative to previous fractal terrain generating method is midpoint displacement. There are multiple methods implementing it. The most known and used one is the diamond-square method.

Instead of calculating multiple octaves for each point and adding them together, which can be inefficient in larger number of octaves, this method calculates the value of each point only once by using recursive function. [14]

The algorithm works as follows:

a) The points representing the edges of a mesh are initialized with pseudo random values

b) The **diamond step** – The point in the middle of the mesh (in the middle of known values) is calculated by averaging the known values and offsetting them by a random value. The offset range is dependent on the iteration. With each iteration known values get closer to the value that is being calculated so the random displacement needs to get lower as well.

c) The **square step** – Here, the points in between each pair of the values from step a) are filled in. It is done again by averaging the known values and adding random offset.

d) The algorithm keeps doing the steps b) and c), using the newly calculated values, until the resolution of the terrain is sufficient.

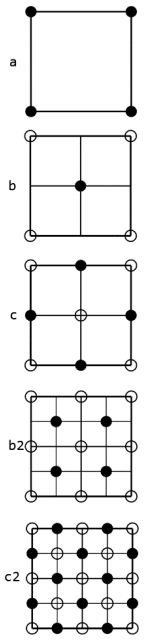

Figure 2.14: Diamond-square algorithm [14]

The advantage of this method is that it is faster and simpler than the fractal noise method, while it produces

fairly similar results. The disadvantage is that it is not so easily manageable and customizable, because of the much lower number of variables contributing to the result.

2.5 Multifractals

The previously described methods result in a homogeneous and isotropic function. The roughness is the same for each octave in noise fractal and at any detail in midpoint displacement. This is fairly accurate but it can be too accurate as it strips down the complexity of real objects from nature, because not many of them can be described as homogeneous and isotropic. Multifractals can be defined as heterogeneous fractals. They are typically created by modifying fractals. This is done by changing fractal dimensions based on the location of the point, that is being calculated.

In 1974 Benoît Mandelbrot published the initial research on multifractals, although the term multifractals was recognized in 1983 by Parisi-Frisch multifractal model, describing hierarchies consisting of sets which have different Hausdorff-Besicovitch dimensions. [15]

Looking at terrain in nature, we can see more roughness at mountainous regions that tend to be in higher altitude with lower slopes. Regions with higher slopes tend to peel away the roughness which can be easily simulated with erosion algorithms such us droplet erosion.

This phenomenon led to the development of algorithms that change the roughness of terrain detail based on the altitude of the point, creating simple but effective multi-fractal terrain.

2.5.1 Smoothed midpoint displacement

Used for creating multi-fractal terrains based on the altitude from midpoint displacement, this method changes the range of the random offset of a new point by not just using the iteration number, but also by averaging heights of the neighbouring known values, making the offset range lower in the lower altitude regions and its look smoother and rougher in higher altitudes. [14]

2.5.2 Multi-fractal noise based on altitude

Using the fractal noise described in section 2.3 and considering the change that is again associated with altitude, each octave after the first one can base its intensity on the sum of previous octaves, making the latter octaves less relevant at lower altitudes, therefore making it smoother and more relevant at higher altitudes, making it rougher.

```
Offset = NoiseFunction(x, y);
If (octave > 1)
{
    Offset *= octaveBaseWeight * const * Sum;
}
Sum += Offset;
```

2.6 Vegetation generating - L-systems

Trees, plants and grass are generally associated with almost any location on the earth. Their geometric features have been studied for centuries for their beauty and symmetry. In computer graphics there is a need to create a large variety of some types of plants, typically of trees for forest, so that each tree from the forest has its unique features, but is still recognizable as a specific type of tree.

For the purposes of creating variability, developmental algorithms, which describe the development of a plant in time are used. The most recognizable are L-systems that were introduced in 1968 by Alistid Lindenmayer as a mathematical theory of plant development. The topology of the neighbouring cells of a plant can be clearly generated using this method, but there is no detail to use in modelling. [16]

2.6.1 Rewriting systems

"In general, rewriting is a technique for defining complex objects by successively replacing parts of a simple initial object using a set of rewriting rules or productions." [16]

It was first proposed in 1905 by von Koch. He described the procedure as follows "one begins with two shapes, an initiator and a generator. The latter is an oriented broken line made up of N equal

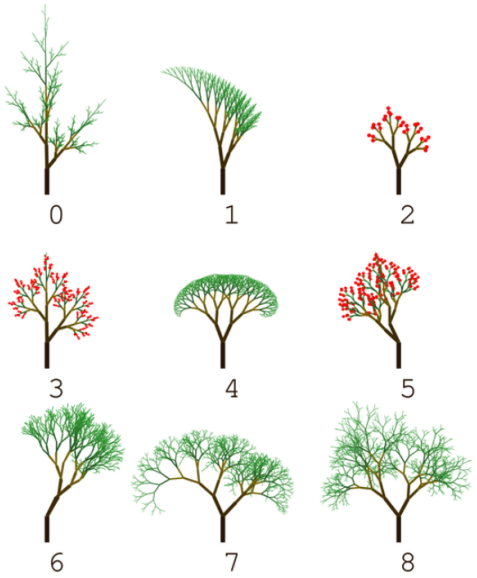

Figure 2.15: L-system generated trees [17]

sides of length r. Thus, each stage of the construction begins with a broken line and consists in replacing each straight interval with a copy of the generator, reduced and displaced so as to have the same end points as those of the interval being replaced." [16]

2.6.2 Introducing randomness

To make the shapes look natural, there needs to be some kind of randomness employed. For example, when generating trees, the possibility of not generating branch which would be there based on previously described algorithm should be introduced. To further randomize the results and make them more natural, randomized offset to an angle of each new branch can be introduced.

2.6.3 Genetic Programming

The L-systems can be further expanded by genetic programming based on Charles Darwin's theory of the survival of the fittest. This method creates more variability while maintaining suitability based on the set parameters. That is accomplished by generating or creating a popula-

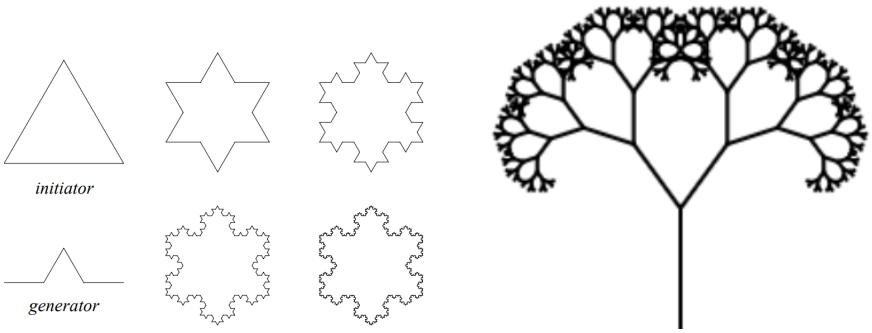

Figure (2.17) Rewriting system snowflake example (left) [16] and tree example (right) [18]

Figure 2.17: Randomized trees [18]

tion of plants which are then combined together using more parents or using one parent and pseudo-random mutation to create another generation. There needs to be a way to determine fitness so the algorithm can stop combining unsuitable plants for another generations. [17]

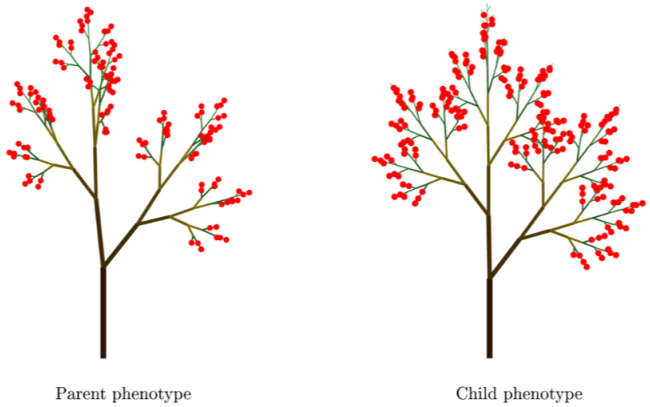

Figure 2.18: Single-parent mutation [17]

2.7 Partitioning

Partitioning is a process of dividing mesh into sections. Most of the algorithms solving partitioning are based on the mesh values such as normal maps, or height maps. This means that the majority of partitioning methods are called after the mesh is partially or entirely finished. Values from the mesh are typically used in UV mapping. That is used in most 3D modelling programs and texturing programs like Substance painter to, for example, generate realistic visuals of worn-out effect.

Partitioning of terrains or maps can range from the most basic algorithms like Binary Space Partitioning that is used in 2D gaming map creation to partitioning of terrain based on its roughness data in the real world [19].

Taking a look at partitioning before the mesh is constructed, to help with creating a more realistic looking mesh structure, there must be a defined set of parameters based on which the division of the mesh can happen. The most basic partitioning would be dividing the space of the mesh based on y and x-coordinates. Although for creating natural looking terrains, more sophisticated algorithms are used.

2.7.1 Noise Partitioning

Noise generating algorithms can be used in a same way in which they are used in generating heightmaps, only with lower number of octaves and frequency. Partitioning generated by noise function like perlin or simplex is continuous and therefore can be used to seamlessly transition from one partition to another by assigning each partition to a range of values inside of the range of possible return values of used noise.

2.7.2 Voronoi diagram partitioning

Voronoi diagram is cellular-based partitioning used in many areas from biology and medicine to procedural texturing and partitioning of 2D space.

Cellular methods for texturing were first introduced by Steve Worley [20]. Cellular algorithms are commonly used in conjunction with noise generating methods. Especially in procedural texturing where they are used to generate cases which noise function would not be able to generate on its own like the motives of scales, organic tissue, mosaic, tiles, etc.

The Voronoi diagram algorithm generates points over the plane area. The position of the points can follow certain rules or it can be completely random. Usually there are some limitations to the minimal distance of the points to prevent their clamping. [21]] For that purpose, Lloyd's algorithm, also known as Voronoi iteration can be used in one or multiple iterations. In each iteration, Lloyd's algorithm moves each of the Voronoi points towards the centroid of its region. [22]

Each of the points of the terrain is then assigned to the closest Voronoi point and inherits the properties of the region. Since the regions have cellular qualities, this should not change parameters which are expected to be continuous.

Complex projects usually use more iterations of partitioning sometimes using more methods to generate more variables to determine the biome of a certain zone. For instance in a project [21], Voronoi diagrams, which are used to determine the height were used in conjunction with Perlin noise that determined the moisture amount. Together

they were able to determine the "Biome" of a the zone. By combining multiple methods, the result becomes more diverse.

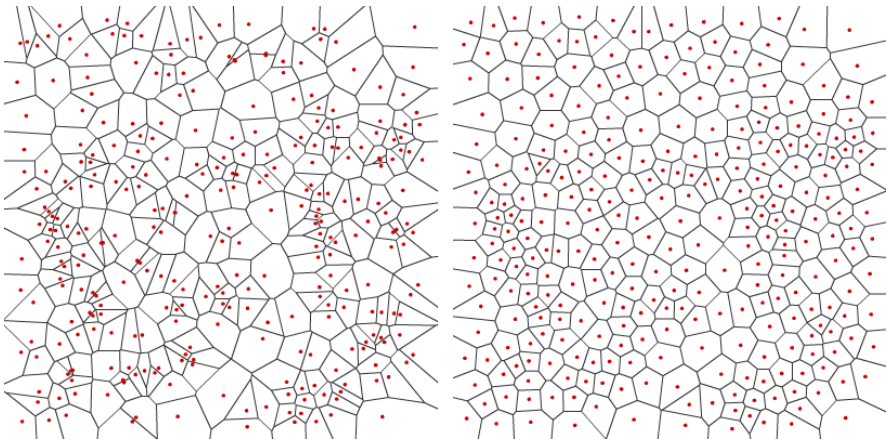

Figure 2.19: Voronoi maps with random points (left) and the same map after 2 iterations of Lloyd's algorithm (right) [21]

2.8 Erosion

The term erosion covers many naturally occurring phenomena. Different terrain types and climates will produce many types of changes to a landscape. Overall, most types of erosion dissolve material from steep slopes, transport it downhill and then deposit the material at lower inclinations. [14]

Wind, water and other natural agents seemingly do not affect the earth much in a short time span, but in a span of thousands of years they can separate continents and change the terrain beyond recognition. Because these types of erosion are partially responsible for the image of the world as we know it, there is a need to simulate these phenomena to create a realistically generated terrain.

2.8.1 Droplet erosion

Droplet erosion algorithm is a particle-based algorithm. Each particle is presented as a water droplet that lands on a terrain at random location with random speed and direction of the flow. This process

typically simulates rain or rivers that take sediment from places where they have the highest speed, typically flowing down the hill and then distributes the sediment in dips or areas where the flow gets slower.

The algorithm works as follows [23] [24]:

1. Droplet is generated on a random location inside a heightmap with predetermined speed, water amount, lifetime cycles and other values describing its potency.

2. Droplet direction vector is calculated based on the gradients of neighbouring heightmap points and previous direction. The droplet is then moved in the direction using normalized vector.

3. Sediment capacity is calculated based on the altitude change, speed and water amount. If the sediment capacity is lower than the sediment currently being held by the droplet, the algorithm continues with the point 4 a) In all other cases it continues with the 4 b)

4. a) Droplet deposits some parts or all of its stored sediment based on the altitude change.

   b) Algorithm erodes the heightmap around the droplet, by lowering altitude of surrounding points and adding the eroded amount to the droplet sediment.

5. Speed and water amount values are updated and if step count did not reach the lifetime cycles of droplet continue at point 2.

6. Algorithm continues with reiteration from point 1 for the erosion iteration count which is usually in thousands.

2.9 Level of Detail (LOD)

The majority of modern applications, which tend to display a lot of 3D meshes at the same time, utilize some kind of level of detail algorithm as a tool to control the balance between quality and performance. They are usually based on distance between the camera and rendered object, but they can also be based on the viewpoint [25].

In usual cases LOD algorithms only change the number of polygons of a mesh. Either full mesh or its parts. In cases where the mesh is further altered after its initial creation, e.g. in the case of this project by the erosion, the algorithms used for further altering the mesh might be run with different sets of parameters or disabled entirely based on the LOD configuration.

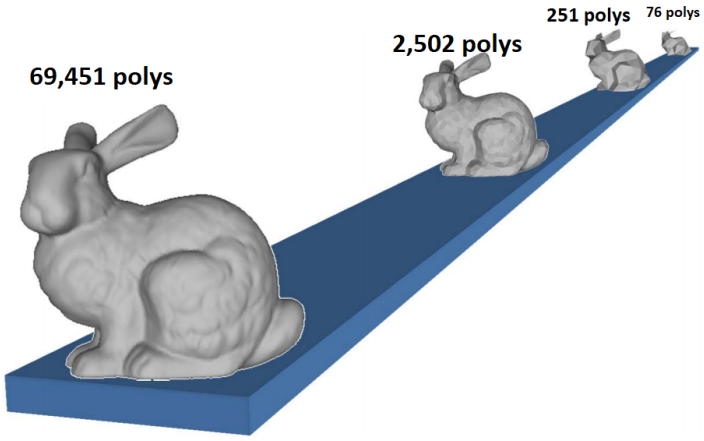

Figure 2.20: LOD based on distance [25]

In 1976 Dr James H. Clark published an academic paper *Hierarchical Geometric Models for Visible Surface Algorithms* in which he described the foundations of a system which he calls "multiple levels of description" and created the initial concept that is the foundation for our current LOD systems. [26]

2.9.1 DLOD - Discrete level of detail (1976)

This LOD framework got its name because each object using DLOD has each of its LODs created in a pre-process at fixed resolutions. In its runtime it only changes the preloaded LODs based on the distance of the camera from the object. [25]

Even though it is the oldest LOD framework it is still the most popular one in gaming and other applications for its simplicity and performance.

Advantages

- Simple algorithm
- Decouples creation of LODs from run-time rendering – better performance, less constrains, possibility of simplifying each mesh manually.
- Fits modern graphical hardware – thanks to pre-process LOD generation LODs can be compiled to simpler types with faster render speed.

Disadvantages

- Not as useful in procedurally generated scenery - can be modified, but most of its advantages are based on the pre-process LOD generation.
- Project size overhead.
- Subdivision of the object (object created from multiple meshes) can result in very noticeable LOD changes, where part of the object is detailed while the other is in a low resolution.
- Popping effect when transferring from one LOD to another.

2.9.2 CLOD - Continuous Level of Detail (1996)

Instead of creating individual levels of detail in the pre-process, CLOD creates structures from which the LOD can be extracted at the runtime. LOD then increases or decreases, based on the distance from rendering camera.

Advantages

- Better granularity and therefore better fidelity.
    - LOD is generated with resolution based on the distance – less polygons at certain distances, i.e. where DLOD would still use the most detailed mesh, CLOD already scales down.
- Removes popping effect - gradually increasing LOD instead of changing it makes the change smoother.

Disadvantages

- More complex and more compute expensive at runtime.
- Cannot use the pre-run compiling of LODs.

2.9.3 View-dependent LOD (1997)

Improving the CLOD, view-dependent LOD adds the function of dividing objects based on the distance from camera and shows closer parts of those objects in a higher resolution than those further.

Advantages

- Polygon allocation based on camera = even better granularity.
- Large meshes do not have to be divided.

Disadvantages

- More complex – more difficult to implement.

2.10 Lake generation

The majority of the algorithms used for creating lakes are another step after generating rivers, either using the river generating algorithm itself [27], or as an expansion of the river algorithm [28]. Other algorithms just use predetermined altitude as a sea level and fill each space under the altitude with water, which prevents any randomness and is not realistic. Since this thesis is not focused on river generation and some randomness should exist across the terrain, the implementation will be based on an algorithm designed specifically for this thesis and described in the implementation chapter 3.6.

2.11 Cave generation

There are many ways of implementing cave generation. It can be done within the generation of the terrain itself using 3D noise [29]. It can be

done separately by artists or by procedural generation and connected with the terrain [1]. It can also be generated from a 2D map after the terrain generation and put inside the terrain using certain rules to define cellular automata [30] [31].

Algorithms generating caves usually use some sort of marching squares in 2D, or marching cubes in 3D algorithm [32]. These algorithms are used for more precise vertex positioning. They divide the space where the cave or terrain is generated to squares or cubes using theirs edges or sides respectively. In 3D, faces are generate based on which edge points of the cube are active. These are then connected with the neighbouring square triangles creating a more complex 3D shape.

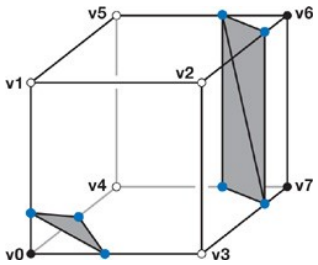

Figure 2.21: Marching cubes [29]

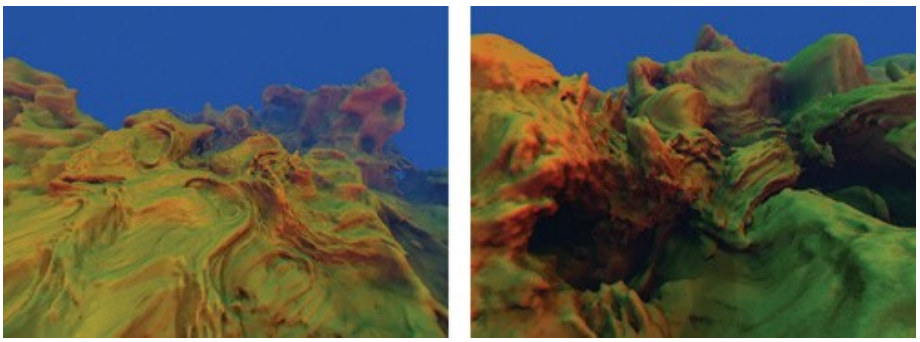

Figure 2.22: Terrain generated by 3D noise and marching cubes [29]

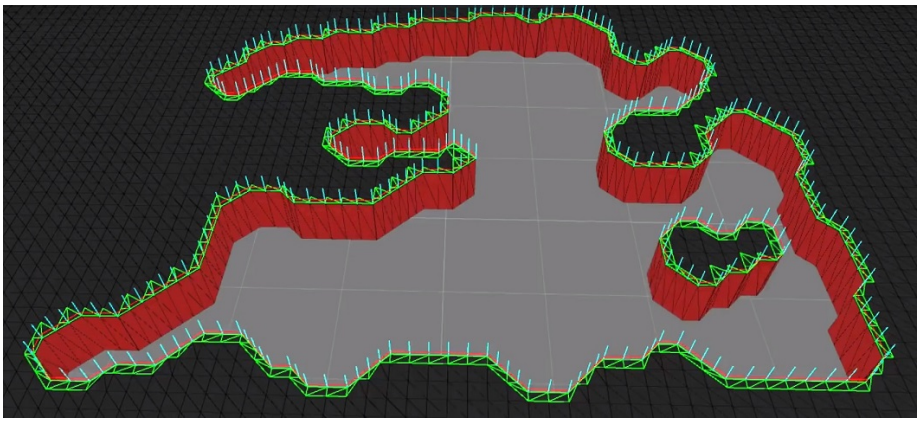

Figure 2.23: Terrain generated by cellular automata in 2D and extruded to 3D [30]

In cases where the cave is a new mesh that is placed inside the terrain, it is also needed to create a passage from the terrain to the cave. This can be done by generating new mesh that will represent the passage and using modified 3D Boolean function to cut of the rendering of the terrain at the beginning of the passage. The other way is to connect vertices of the terrain with the vertices of the cave and disable the rendering of the triangles that are inside the connected vertices.

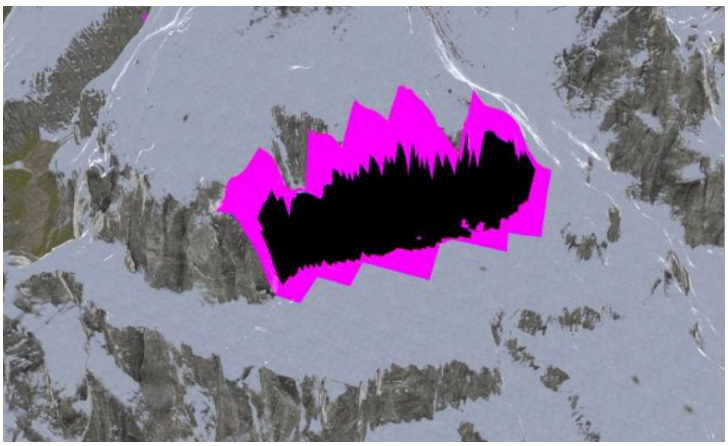

Figure 2.24: Terrain faces at the entrance to the cave [1]

2.12 Related projects

This thesis is inspired by multiple papers and projects, which I will use to compare the results with. Some of them are high budged modern projects [1] [2], some of them are works of single or multiple individuals building on each others work [21] [24] [23]

The goal isn't to compete with the high budgets projects, but to use them as a reference of what is possible.

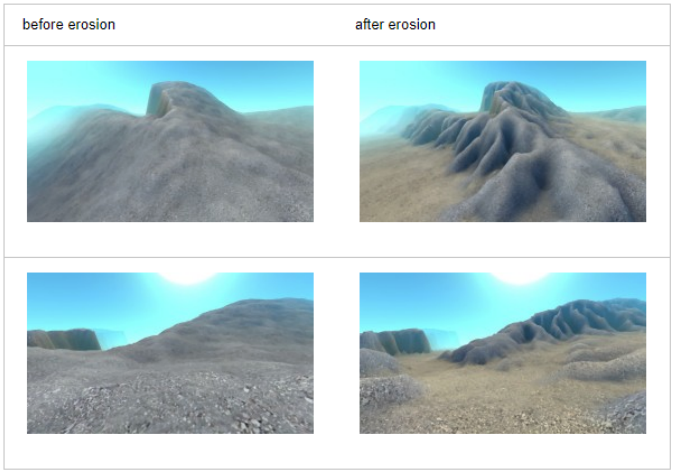

Figure 2.25: Droplet erosion project [24]

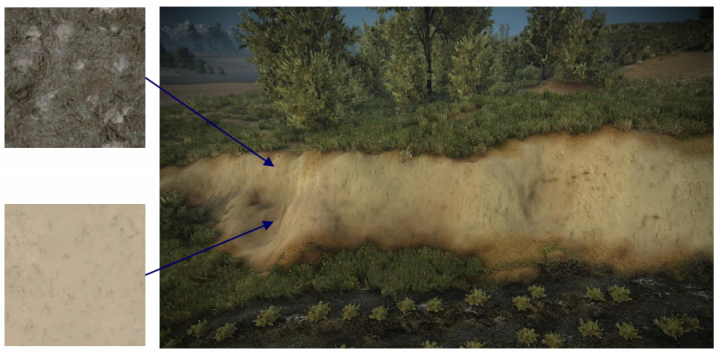

Figure 2.26: REDengine 3 texturing based on slopes [1]

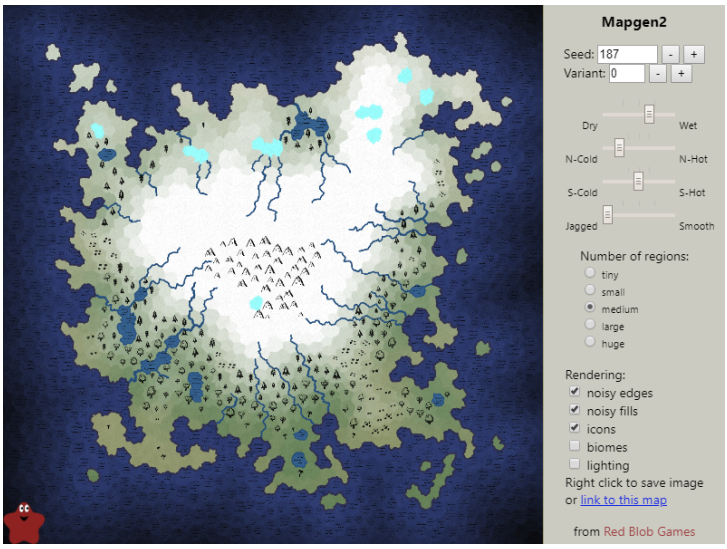

Figure 2.27: Voronoi map generator project [33]

3 Design and implementation

This chapter introduces the implemented methods, the requirements that were supposed to be fulfilled and the methods used, as well as the design decision made to meet those requirements.

3.1 Base terrain generation implementation

First thing that had to be implemented is the randomized terrain generation with heightmaps that will be modified by later methods.

Terrain is supposed to be based on the multifractal generation method. Terrain will consist of parts that can be added and destroyed based on the distance from the camera.

For the reason of having better control over the resulting terrain, I decided to use the multi-fractal noise method to implement the base terrain generation. Three noise generating methods will be added for the possibility of combining them and using them in different octaves of the noise generation. These will be Perlin, Simplex and Perlin ridged noise.

A class will be created to hold the settings and parts of the terrain. It will have a method checking the distance of the camera from the meshes and it will be creating and deleting them based on that distance.

3.2 Partitioning Implementation

Partitioning as described in the section 2.7 is used for dividing the mesh or in this case terrain composed of multiple meshes into a section with different properties. These properties will be influencing other generating methods used in this project, so it was important to implement the partitioning right at the beginning. The partitioning will be used for three methods which need to be taken into consideration:

- The First method partitioning will be used for is procedural texturing of the terrain.

- Another method will use the partitioning for manipulating the parameters of classes responsible for creating the terrain mesh, mostly impacting the intensity and frequency of octaves based on the zone, but also multiplying the height for the mountainous regions.

- Lastly, after the mesh is created, partitioning will be also used for modifying the parameters of erosion by changing the parameters of generated droplets that will differ in strength and number of iterations.

The partitioning is to be implemented with the possibility of adding new parts of the map, as well as with the possibility of recreating previously generated results for the purpose of on-fly generation. Partitions should have random or pseudo random sizes and shapes. There should also be a smooth transition between the neighbouring partitions. As the output of the function will be used for UV mapping, it should return values in a predetermined interval. Therefore, it should preferably return values from a function which is continuous and which has some sort of predetermined limits.

Since the partitions must be determined before the mesh creation, the values of partition will be mapped to another height map that will be referred to as "partition map".

The partition map is expected to be continuous and generated before the heigtmap. Since Voronoi diagrams would not offer much in this case, the method that will be implemented for the map partitioning will also use noise. This noise will use very low frequency so the transitions between the partitions will not happen too often and there is enough space to spot the difference between the heightmaps generated by each partition.

The octaves used in this noise generation should stay low, preferably only single octave should be used, so the value would not fluctuate too much around the middle of the value range.

3.3 Droplet Erosion Implementation

Droplet erosion was implemented to make the noise smoother at higher slopes, so that the terrain looks more realistically.

Erosion should add a realistic element to the terrain by smoothing it and redepositing the sediment around the map.

Erosion will behave differently based on which partition it starts on. The parameters that influence the erosion behaviour will be modifiable in the partition configuration.

Erosion will be implemented based on the algorithm described in the chapter 2.8.1 which is based on projects in [24] [23] and rewritten to suit the needs of this project. To add the possibility to change the erosion based on partition, each partition will have settable droplet parameters. The droplet parameter values will then be set by interpolation based on the value of partition of a point, where the droplet is generated. Erosion intensity determined by iteration count will be settable in the configuration of each LOD described in the chapter 3.4.

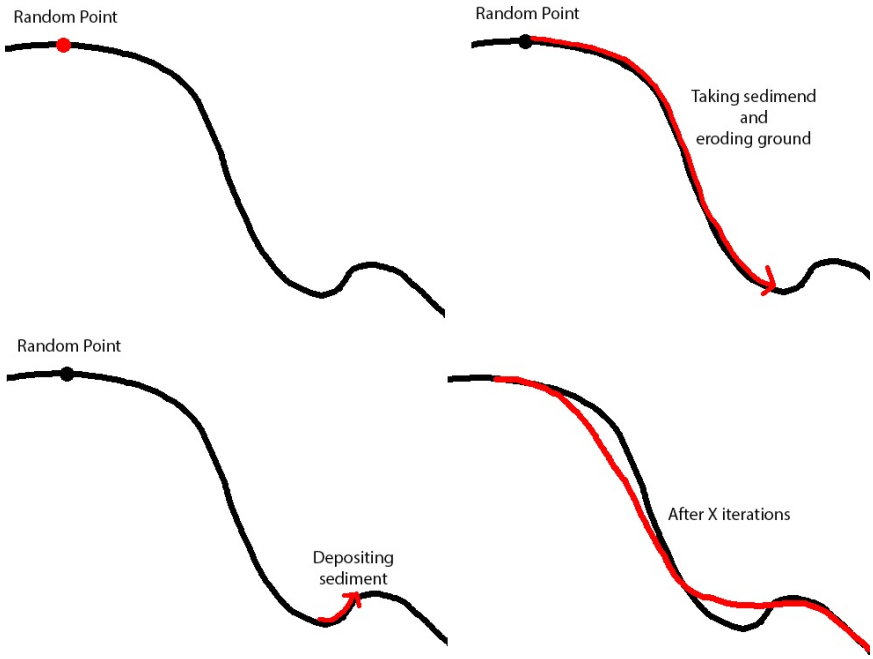

Figure 3.1: Droplet erosion functionality

3.4 Level of Detail Implementation

Level of detail was implemented to manage the rendering and generating speed of terrain with more meshes in the on-fly generation. LOD will be implemented as DLOD, i.e. as dynamic level of detail, for its simplicity and better manageability. It should be based on the distance from position of the camera on a 2D grid.

Each LOD will define:

- Distance from the camera from which the rendering will commence.
- The resolution of a mesh.
- Number of iterations for the erosion.
- Boolean determining whether lakes and caves will be generated.

Each of the meshes should only have erosion iterations equal to the current LOD setting, therefore it is needed to take into account the iterations done in previous LODs.

As there is a need to continuously check the distance between positions where meshes were or can be created and the camera position, the algorithm will evaluate the distance of each mesh in each frame of the program. LOD will be then assigned based on that distance.

The first LOD will generate the height map with the resolution of the highest LOD, so there is no need to recreate it every time the LOD changes. This brings a limitation for setting the resolution of each LOD to a value that can divide the resolution of the highest LOD based on which height maps are created, because each pixel height is calculated as an approximate index of the heightmap.

Upgrading the LOD will modify the heightmap in order to create better detail. There will be no downgrading implemented as it would just change the resolution and therefore the mesh would need to be rewritten. As the rendering speed is not the bottleneck in this implementation, it would do more harm than good and slowed the CPU operations. Instead, meshes will be destroyed if the critical distance from camera is reached.

Each of the meshes will save the current LOD for the comparison when reassigning it. Each LOD will be based on a serialized object

that can be modified during or before the runtime. There will also be a possibility to add new LODs in the unity editor.

3.5 Procedural texturing Implementation

Texturing in this project is used to provide more information about the terrain and to make it look more realistically. The basic idea is to change the terrain texturing based on:

- **Partitioning** – Based on the previously described partitioning of a map, each point will be mapped to a gradient of colors linked to the partition, e.g. grades of green for grassland and grades of grey for mountains.
- **Height** – There will be a gradient of colors describing the change of the color of a partition zone based on height, e.g. mountains will have snow in higher altitudes, therefore the points over some given threshold will be mapped to be white.
- **Slope** – Based on a slope, calculated from normal vectors of each point, points will be mapped to a different color, e.g. grass will turn from green color to brown earth color at higher slopes to simulate that grass cannot grow there.

As in some cases it might be desirable to see the height of each point, there should also be a second material based on a shader that will set colors only on the height of mesh vertices.

Because the partitioning is based on a continuous function it will be possible to use 2D texture for the transitions between the sections. The texture will therefore consist of two axes, where x-coordinates will be used for the height gradient and y-coordinates for the transitions between the partitions. The height gradient is predetermined as a parameter so the texture will have enough data to generate its partitions by a linear interpolation.

The problems may arise when the slope is considered. Because there is no more space to create seamless transitions from regular gradient of colors to the gradient of color used for greater slopes. This could be solved by using texture with a higher dimension, but that would increase the complexity. The other way is to use another 2D

texture that will be created in the same way as the first one, only using a different color gradient. To put them together, a shader will be created. This shader will implement an interpolation between the two textures based on the slope calculated from a normal map for each point.

For the second material based only on height, a simple shader will be created that will interpolate color from white to black based on the y-coordinates of each point.

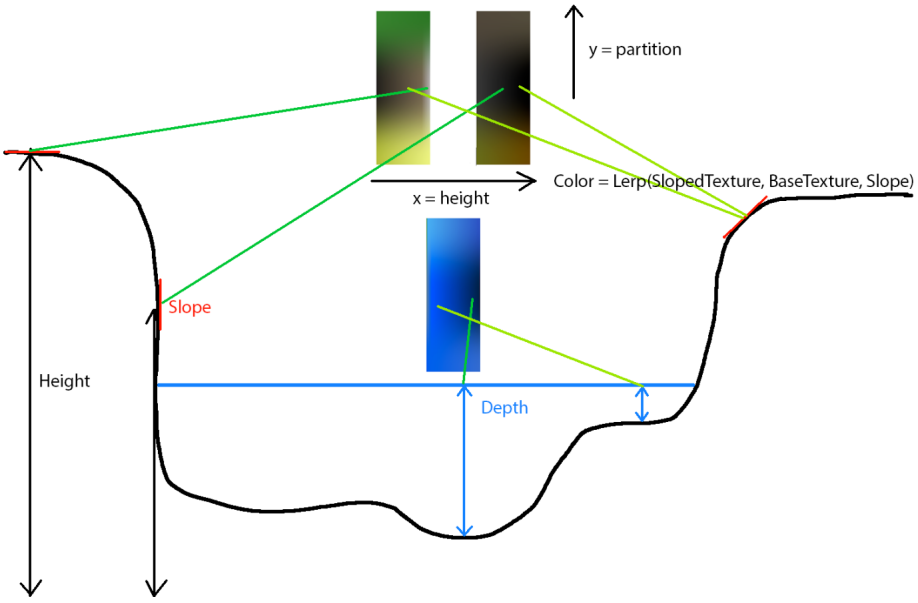

Figure 3.2: Texturing terrain based on heights and slopes

3.6 Lake generation Implementation

Lakes will be generated in the LODs in which the erosion is already done, because the heightmap in the lake area is flattened down and the erosion would have to consider it if it was to run after, which would make it behave less accurate.

Lakes should not be based on a certain altitude, but should be rather generated in any height where it is possible to find a pit that is surrounded by higher ground. This place should be found pseudo randomly so the result can be recreated, but variety across map exists.

Lakes might cross multiple mesh tiles. Thus it will be necessary to edit height maps from more of them at the same time.

The water tiles are supposed to render with a new water texture. It should have higher reflection properties that should also scale with the depth of the water. This texture should transition smoothly from the terrain nearby to simulate lower terrain that is visible in lower depths.

For determining the area by height, a modified flood-fill algorithm [34] was created. The flood-fill algorithm is used in filling areas will color being bounded by pixels of other colors than the one from the starting point. This algorithm works in the same way, but instead of color it uses height as its determining value and returns all the height map points connected to the original point bounded by higher altitude points.

Because it is very probable that the lake will interfere with the terrain across multiple meshes and, at the same time, there needs to be a limit for determining the unsuitability of putting lakes in some places and heights, the algorithm will use the height map consisting of the main mesh and its neighbours which was introduced in the droplet erosion described in the chapter 4.2.2.

All the mesh faces will get another map for lakes similar to partition map. This map will be based on integer numbers and will be zero for anywhere but the lake for which the values will be positive based on the altitude of the lake.

Algorithm for generating lakes works as follows:

1. First the algorithm determines a pseudo-random starting point based on a seed. This point will be inside the middle mesh never in the neighbouring meshes so it doesn't run out of bounds so easily.

2. The starting point will be moved to the lowest neighbour until it ends up in a dip, where all neighbouring vertices have higher heights.

3. From that point a height will be taken and a slight offset will be added to it. Then a modified flood-fill algorithm, will get all connected points that are under the given height.

4. If the customized flood-fill algorithm gets out of the range of the height map depending on if it is the first iteration the lake won't be created, or the previous iteration value map will be returned.

5. This algorithm will run in multiple iterations always adding some offset to the height to get as much terrain filled as possible based on the surrounding hills.

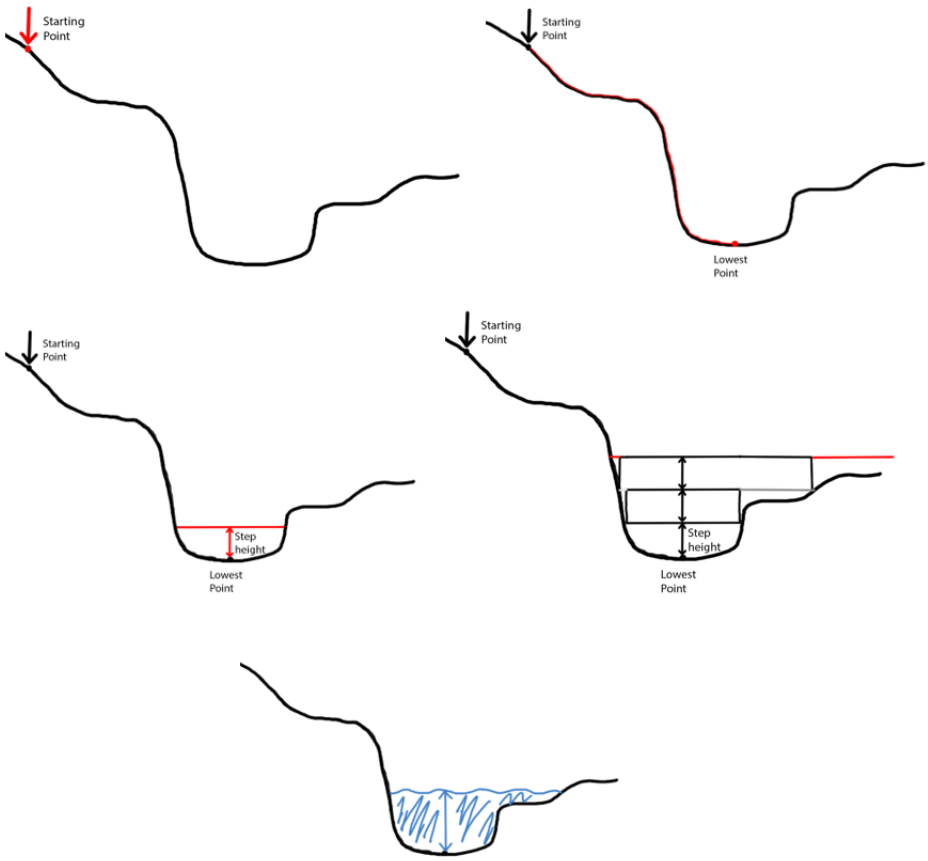

Figure 3.3: Steps of lake generation

To make the lakes visually stand out, a new texture will be needed. Therefore a new gradient will be added to each partition for the water values. From these gradients a new texture will be created.

Since the shader used for terrain will need both information about the altitude and the depth of the lake at each point, it will be mapped in a new (second) UV-set. The water texture will then be added as a third texture to this shader. Based on both UV-sets the color of a lake will interpolate based on the depth of the lake and partition it exists in. Another thing that the depth will be used for is the smoothness of the texture which will make it more reflective as the depth increases.

3.7 Cave generation implementation

Caves will be created in mountainous parts of the terrain as well as in parts with higher heights and to each cave there should be an entrance.

Since every mesh already holds a *lake map* which determines the areas that will be flooded, the cave generation will save itself on the same map. This will save some memory and calculating power and also exclude the possibility of lake and cave coexisting at the same part of the terrain. This opens the possibility of creating caves as independent meshes.

The algorithm will start just as the lake generating algorithm, but the opposite parameters will be used. From random point, it will find the highest point and from there it will start going down until it overflows or finds enough space to create a cave.

The cave ground will be created flat and later adjusted by noise. The ceiling will be copied from the mountain above mirroring its shape. These two parts will be connected with a slight offset to create a wall as a part of the cave.

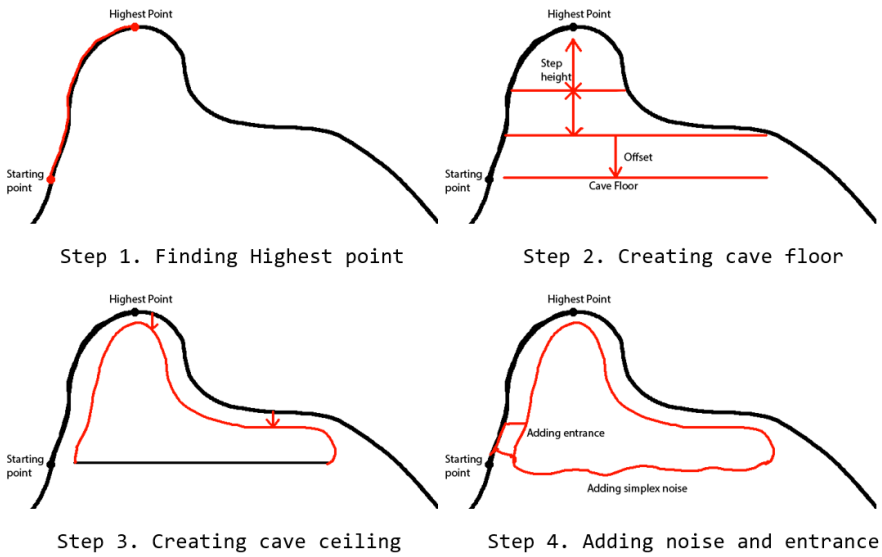

Figure 3.4: Cave generation steps

To create entrances, it was planned to generate a corridor as a new mesh and then use customized Boolean 3D function to stop the terrain from rendering from the beginning of the corridor. Unfortunately, the chosen technology, specifically Lightweight Render Pipeline package used for shader graph, made this task impossible. Later it was decided not to render some faces of the terrain and use them as an entrance.

3.8 Technologies used

Attached project that generates multi-fractal nature through noise based heightmap, partitioning and erosion was coded in C# and implemented in Unity engine version *2018.3.6f1*. Sample program is intended to run on Microsoft Windows system only. The unity project is modified to work with .NET 4.x and higher and some of the code depends on that. The project also uses Lightweight Render Pipeline package that enables shader graph in which some of the shaders are created and on which all the materials used in the project depend.

4 Results

This chapter presents the implemented algorithms and examines their compatibility with the project.

At the end of the chapter there will also be an evaluation of the effectivity and efficiency of the implemented algorithms based on speed, accuracy and other factors in different resolutions and scenarios.

4.1 Base terrain

I used the code from this source [7] to implement Perlin and Simplex noise, slightly modifying the methods. These methods were later expanded with ridged noise which was implemented a new method based on Perlin noise method.

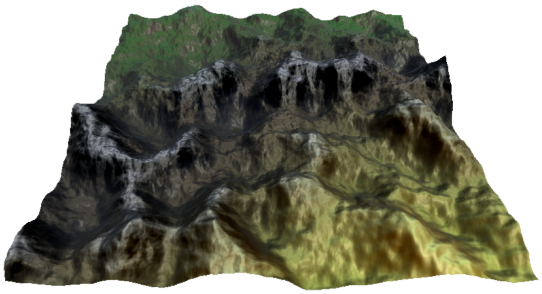

Figure 4.1: Terrain using Perlin in all 7 octaves

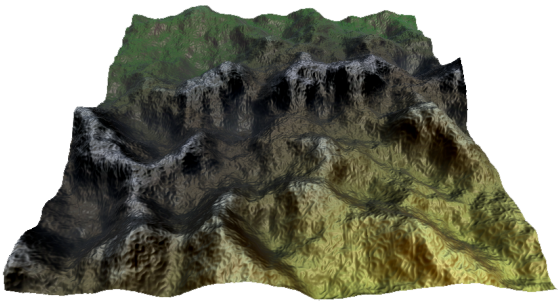

Figure 4.2: Terrain using ridged noise in the fourth octave

4.2 Droplet erosion dealing with gaps

The fist implementation of droplet erosion algorithm dealt with the erosion effect only on one mesh, modifying its height map. The problem that arose from this was that the mesh was no longer connected with its neighbours. Although not big, gaps were created on the borders of the mesh because the erosion ended with the end of the mesh and did not affect the meshes of the neighbours. To deal with that, two methods were designed and implemented.

4.2.1 Removing gaps by interpolation

The first method was designed to do a simple interpolation of the borders in order to glue together the gaps.

This method was designed with a simple, but important limitation in mind: the terrain must always be generated in the same way. This, unfortunately, means that one mesh cannot influence other meshes because they can be created later or sooner. The results therefore would not be the same in every case if they were to be influenced by the neighbouring meshes.

Advantages

- Simple algorithm without any complex computation.
- Always generates the same result.

Disadvantages

- Creates a slightly visible seam (usually only visible from top-down camera view)

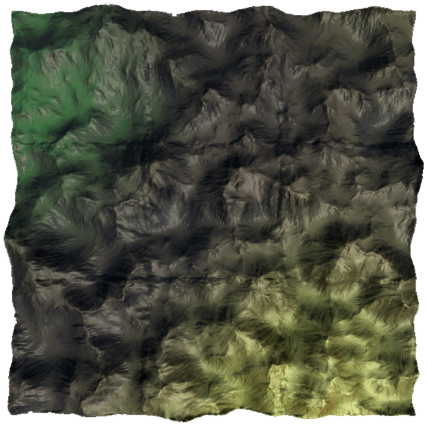

Figure 4.3: Top-down view on the eroded terrain connected by interpolation

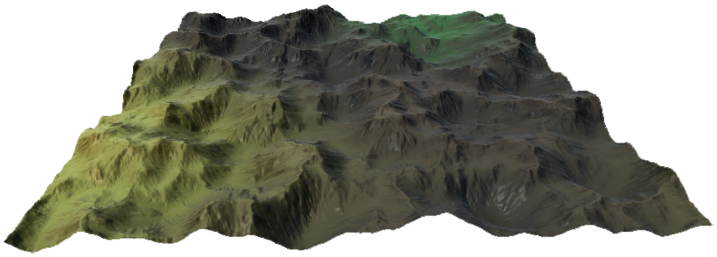

Figure 4.4: Side view on the eroded terrain connected by interpolation

4.2.2 Removing gaps by eroding neighbours

The second method was designed to deal with the seams. It functions well, but, unfortunately, falls flat in other areas. By using this method, the erosion will not only erode the mesh it started on, but the droplets can also travel to the neighbouring meshes modifying their heightmap as well.

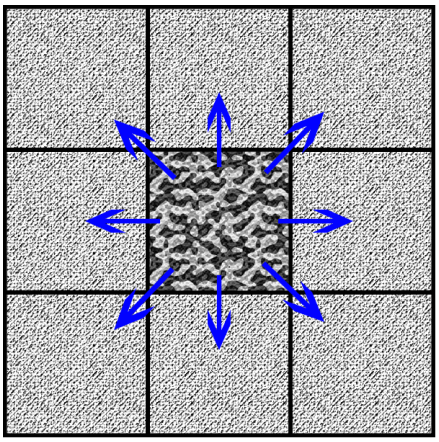

Figure 4.5: Erosion overflowing to the heightmaps of neighbouring meshes

To do that, a new method had to be created. This method merges all the heightmaps of neighbouring meshes with the one that the erosion is done for. This new merged heightmap is then used in the droplet algorithm. The droplets are only generated in the middle section, but are now able to travel to other sections as well. After the erosion is completed, the merged heightmap is redistributed to each affected mesh.

Since the erosion is mostly used in higher resolutions, merging and unmerging the heightmaps is a costly operation.

The most serious problem with this method is the possibility of getting different results based on the route taken by the player. Because the erosion affects neighbouring meshes, a mesh will look differently if it was rendered before or after its neighbours. Even though this difference is not very much noticeable, it can affect the generation and placement of other things like lakes or caves. And if a multiplayer is implemented, these little changes might end up as very noticeable on each of the machines connected.

Advantages

- The results have better quality.
    - No visible seams.

- Erosion is slightly more detailed.

Disadvantages

- More complex and slower algorithm.
- Generates different results depending on which mesh was eroded first.

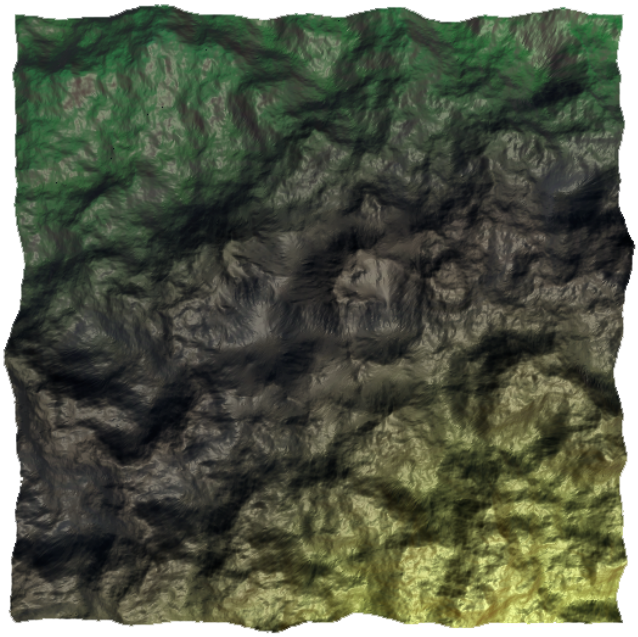

Figure 4.6: Top-down view on the eroded terrain using overflowing

4.3 Level of detail

For the base purposes four LODs were created. The fist LOD with the least range assigned is set with the highest resolution and ideal count of iterations for erosion. The second LOD has same resolution to maintain the connectivity of the meshes with the fist LOD, but it has less erosion iterations. The last two do not have any erosion iterations and their resolution is set lower. This means that they will

be disconnected from the neighbouring meshes with higher LOD, but it should be at a distance not noticeable for the player.

As can be seen in the figure 4.7, when looking from a point that is close to the highly detailed LOD, there are no gaps or low quality visible at the meshes in the distance which use low detail LODs as they can be seen in the figure 4.8. Here the camera is positioned at the end of the map where the lowest LOD is used.

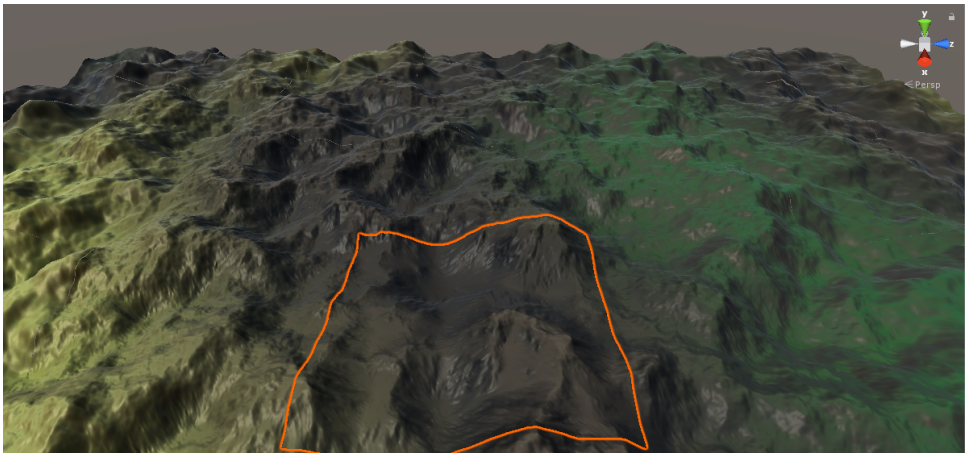

Figure 4.7: Level of detail at the center of the terrain

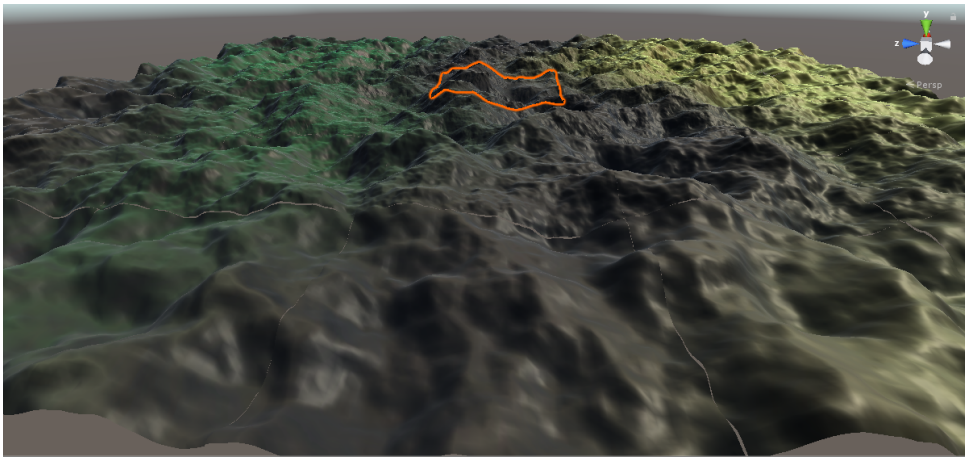

Figure 4.8: Level of detail at the edge of the terrain

4.3.1 Evaluation of procedural texturing

As it can be seen in the pictures in the figure 4.9, first a texture based on partitions gradient colors was created. These colors were distributed along the width of the texture by using gradient function. The second picture shows a linear interpolation between each zone and their gradients. Since the linear interpolation produced too blurred results in which the middle zone started to fade too early, a smoothing function was implemented. In this case the smoothing function is the same as the smoothing function used in simplex noise, but even a simpler function would provide similar results. The result after the smoothing was much better than simply using the linear interpolation and therefore the texture with smoothing was used from that point on.

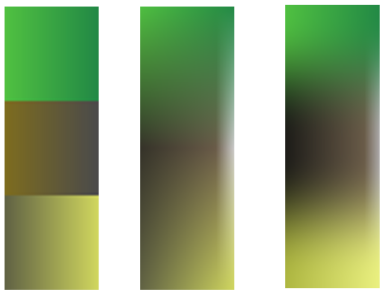

Figure 4.9: From left, texture without smoothing, texture using linear smoothing and texture using smoothing function ($6t^5 - 15t^4 + 10t^3$)

As seen in the figure 4.10, similar function was used to generate the second texture for sloped terrain and third for water surface used in lake generation which will be explained in chapter 3.6.

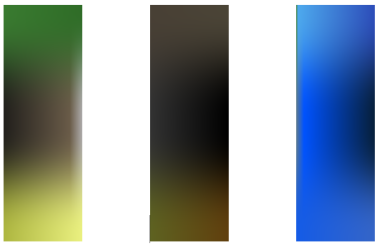

Figure 4.10: From left, base texture, sloped texture and water texture

The main shader uses UV mapping to determine where to take color from both the sloped and the base texture that are supplied into the material using this shader. The linear interpolation between these two colors is based on normal maps by simply adding the x and z-coordinates from the normal vector of each point from which the shader gets the slope value between 0 and 1. For the purpose of the possible management of the color linear interpolation, a multiplier parameter that scales the t in the linear interpolation function was added.

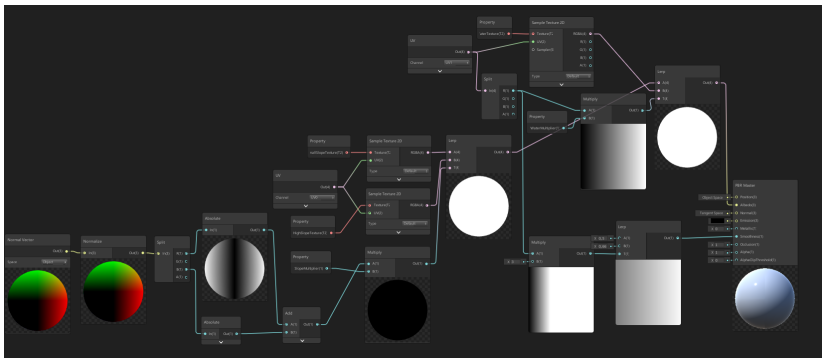

Figure 4.11: The main shader logic

The shader for heightmap simply takes the position of vertex and sets the color base of the y-coordinate by interpolation from white to black. Since the noise that was used for generating the heightmap varies in a range depending on the parameters, a parameter of the max value was added, so the linear interpolation would not clamp every value over 1.

Texture colors can be modified through the unity editor in the partitions config (scriptable object) which can be found in the config folder in project files and can be generated from configurator object in the main scene.

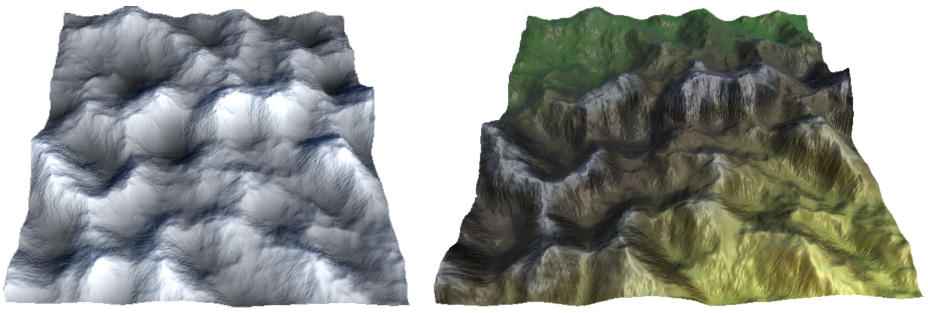

Figure 4.12: Terrain using height based material (left) and terrain using base material (right)

4.4 Evaluation of lake generation

The algorithm can flood large areas and thanks to the shader changes creates pleasant looking parts of the map.

The texture color of the water can be modified same as the other textures, in the partitions config and then re-generated in the config object in the scene. The color gradient can also be changed in the material with the parameter of water multiplier that will basically multiply the depth of the lake so that shores are less visible.

The only thing that the algorithm lacks behind is the speed of determining whether the flood would stay in the bounds of the neighbouring meshes. The figure 4.13 illustrates a fairly large lake which was created by three iterations of the previously described algorithm. The third iteration got out of the bounds of the neighbours and therefore it used the second iteration as an end result. In the figure 4.14 the algorithm made five iterations were set as a maximum for this test and then ended. The first situation took the algorithm 390 milliseconds to complete, while the second only took 74 milliseconds, even though it went through more iterations. It is so because it takes a lot of time before the algorithm gets to the borders if it over-floods the middle region, for it checks every point in between.

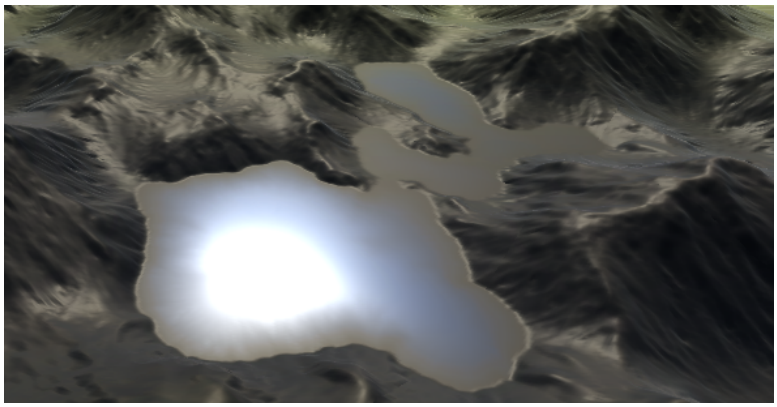

Figure 4.13: Lake generated in 390 milliseconds

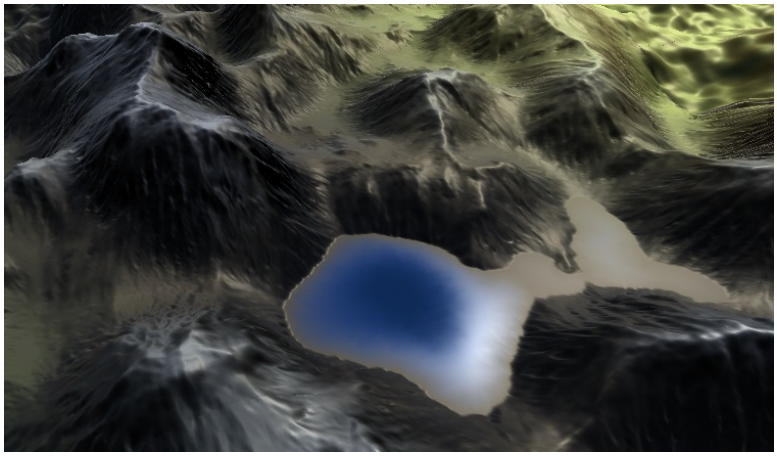

Figure 4.14: Deeper lake generated in 74 milliseconds

4.5 Evaluation of cave generation

As it was mentioned in the implementation chapter 3.7, the used technology prevented employing some of the proposed functionality elements and compromises had to be made. This, to a greater or lesser extent, led to somewhat diassapointing results.

The result, although functional, was not good enough to justify the integration with the rest of the project. Especially since it would

mean adding a new logic for the added meshes that were not part of the terrain.

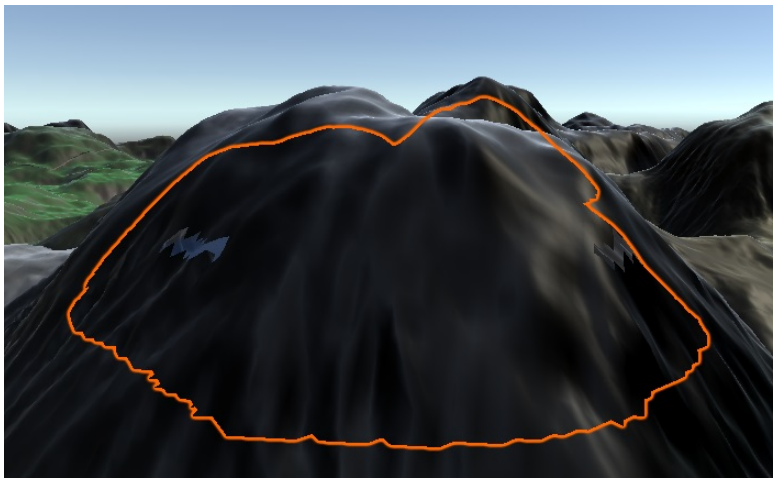

Figure 4.15: Cave from outside of the terrain

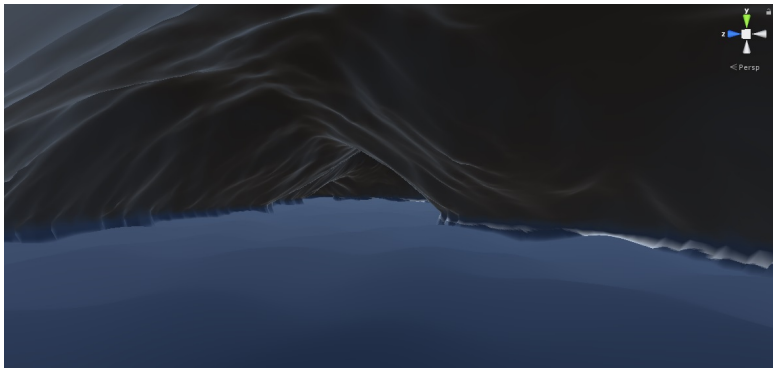

Figure 4.16: Inside of the cave

4.6 Evaluating speed and finding the bottleneck

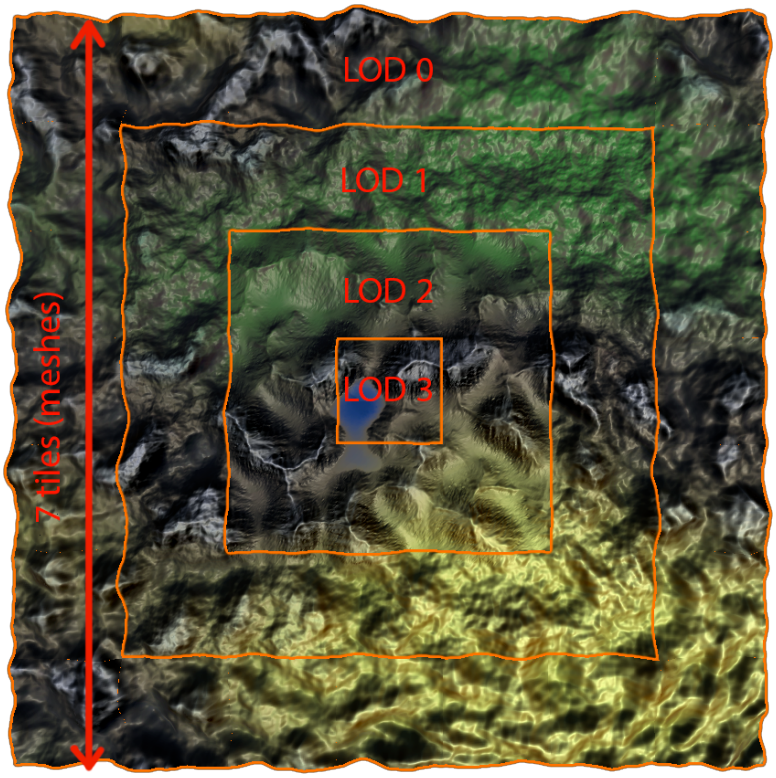

Figure 4.17: Base testing parameters

In this scenario speed of implemented algorithms in different resolutions will be compared to find out which ones slow down the generation process the most and how is the speed dependent on the resolution.

To do this we will present the generations from the most basic terrain generation in which meshes will only be generated from multifractal noise, to the terrain generation based on partitioning with erosion and lake generation. This will be done for 7x7 meshes map where the lake will only be generated on the middle mesh and erosion only on the nine middle meshes.

Each generation will be repeated three times with different seeds and the result will be the average of the three measured times. These

results will still be very indicative for they highly depend on the hardware used.

A. Base terrain generation only using seven octaves of Perlin noise.

B. Again seven octaves but this time based on partitioning. Again only Perlin noise was used for every octave so the comparison is more accurate.

C. Adding 20 000 iterations of erosion to the nine middle meshes.

D. The erosion will also affect the neighbours using the overflowing method.

E. Adding Lake to the middle mesh.

F. There is also some additional info about average speed of algorithms based on single mesh at the end of the table.

For these experiments two machines, which I have at my disposal will be used:

1. CPU: overclocked Ryzen-1600, GPU: GTX 1060, 16GB RAM
2. CPU: i7-4720HQ, GPU: GTX 860m, 12GB RAM

Table 4.1: Algorithms and their speed in different resolutions using machine 1

Algorithm/Resolution	252x252	126x126	63x63
A. Terrain generation	6.73s	1.76s	0.65s
B. A + partitioning	12.37s	3.16s	0.98s
C. B + erosion	19.83s	9.2s	5.87s
D. C + overflowing	25.11s	12.11s	8.68s
E. D + lake	25.15s	11,91s	8.75s
Heighmap	98ms	25ms	6ms
Partition base heightmap	212ms	54ms	13ms
Errosion 20000 iterations	831ms	684ms	534ms
Lake	55ms	28ms	13ms

Table 4.2: Algorithms and their speed in different resolutions using machine 2

Algorithm/Resolution	252x252	126x126	63x63
A. Terrain generation	8.81s	2.19s	0.78s
B. A + partitioning	13.7s	3.51s	1.11s
C. B + erosion	24.39s	10.73s	6.7s
D. C + overflowing	31.81s	14.11s	10.3s
E. D + lake	31.73s	14.23s	10.36s
Heighmap	131ms	33ms	8ms
Partition base heightmap	253ms	59ms	15ms
Errosion 20000 iterations	931ms	753ms	626ms
Lake	63ms	45ms	18ms

Based on this experiment we can see that a lot of speed is gained when the resolution is lowered since the heightmap and partitioning are done much faster. The erosion in lower resolutions is a bit misleading, since to erode a lower resolution mesh less iterations is needed. This being said, it slows down the generation considerably just as basing heightmaps on partitioning does.

This also shows that to run it smoothly in on-fly generation, the resolution will have to be lowered and erosion iteration count will need to be smaller.

4.7 Evaluating effectivity

In this part of the evaluation, the project's results will be compared with a real world terrain.

For this scenario, a way in partition generation to only return values for one zone was implemented so the evaluation can be done separately for each zone.

Because there is no vegetation implemented, the grassland is more or less unusable for comparison. Hence, only the deserts and mountains will be compared.

4.7.1 Generating realistic desert

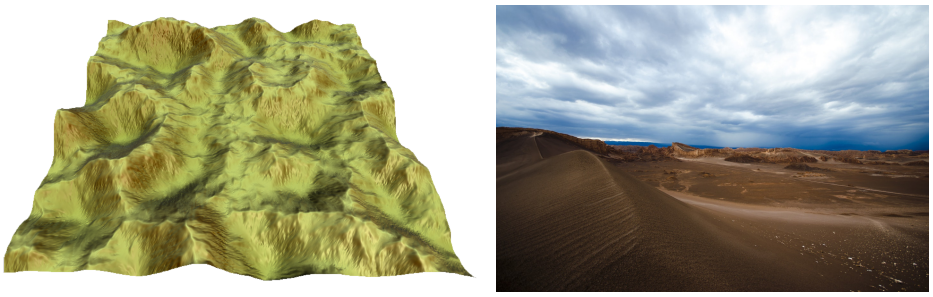

Figure 4.18: Generated desert vs real world desert [35]

The ridges in the photo shown in the figure 4.18 are much smoother than in the terrain generation. This is something that the erosion is responsible for, so by adding more iterations we can get closer to the wanted result.

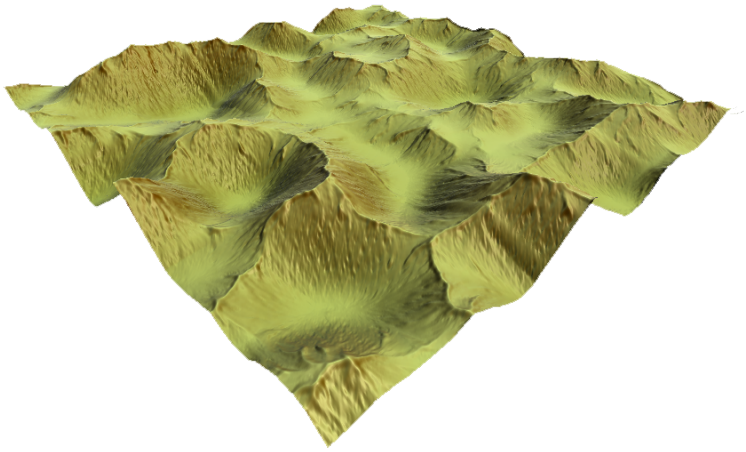

Figure 4.19: Generated desert with 200 000 iterations of droplet erosion

The result in the figure 4.19 is much closer to the wanted result, although it is still limited by its resolution. The ridges are thus slightly wider.

4.7.2 Generating realistic mountains

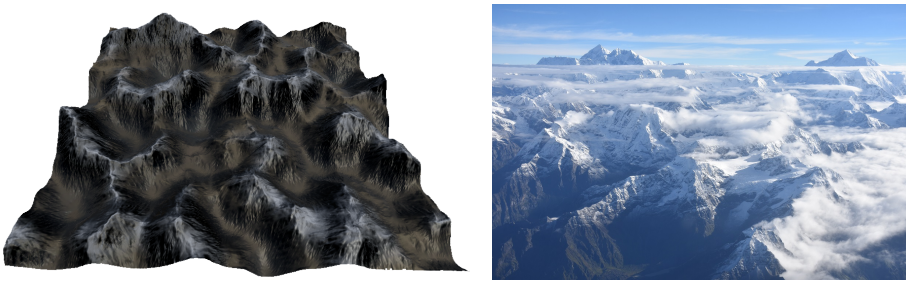

Figure 4.20: Generated mountains vs real world mountains [35]

In figure 4.20 we can see that in the real world snow is distributed more randomly and the altitude differs more than in the generated terrain. To solve this, the partitioning was modified so that the height is multiplied by higher value and therefore there will be higher differences between the low and high ground. Then the material was modified so the slope gradient which does not contain white, simulating the lack of snow, isn't influencing the result to such an extent.

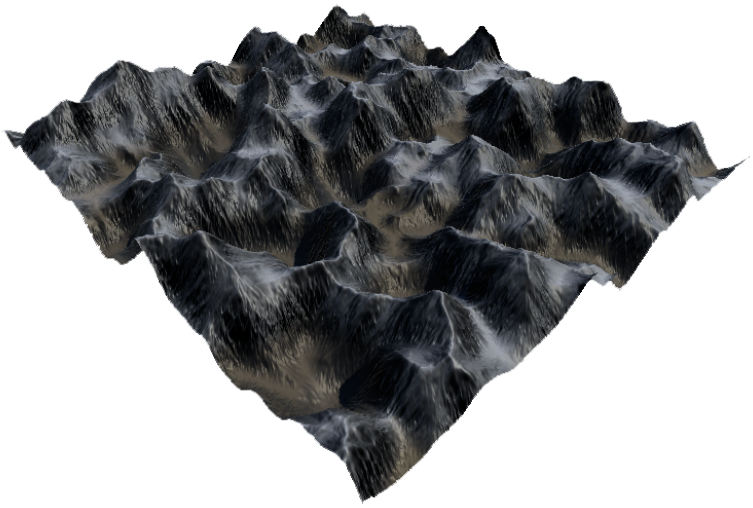

Figure 4.21: Generated mountain with modified partitioning

The result visible in the figure 4.21 is much closer to the real world mountains than the previously generated terrain.

As it was demonstrated in these cases, the implementation of the generation offers many ways to change the generation in a desired way through the use of parameters. It is done in a fairly simple way and does not require much of programming knowledge.

5 Conclusion

5.1 Summary

In this thesis, a generator of a 3D-terrain was implemented. This generator provides the ability to create terrain defined by multiple parameters allowing creation of terrain that is highly customizable by the user without much effort. This customization can be focused on recreating the real-world terrain as demonstrated in the section 4.7 or to suit personal preferences.

During the implementation multiple methods used in multi-fractal terrain or in other types of procedural generation were evaluated. They were also customized to fit the needs of the project.

Current implementation can be used for flight simulators or other applications that could use the ability of on-fly endless terrain generation or for generating terrain meshes with predetermined parameters. Yet there is still a lot of space for expanding and improving the program in the future 5.2.

In comparison with other projects, this thesis focuses on combining the functionality of used algorithms to create wider variety and possibility of customizing the generation. This led to higher quality, but also higher complexity of the project and slower generation speed. It also had to be dealt with multiple meshes, for instance, in droplet erosion. Here functions dealing with problems that arose from this were added, again increasing complexity, but at the same time increasing usability of the algorithm in comparison with the projects implementing droplet erosion on a single mesh [24] [23]. The increase in complexity, unfortunately, also made it harder to rewrite it into a compute shader, which would help with the performance as was done in the later version of one of the other projects [23].

The texturing of the terrain provided satisfying results. This was managed through generating textures from gradients which are used by a custom shader. This approach, although simpler, is similar to projects with high budgets [1] and could be expanded fairly easily with more parameters. The main thing missing here the use of seamless textures instead of the gradients.

On the other hand, cave generation, was of a disappointment. Multiple compromises had to be made because of the previous design decision which then made it impossible to use the marching cubes algorithms [29], that are usually used for cave generation, without reworking already implemented methods and because of the limitations of the used technology, which made it impossible to create custom 3D Boolean function for creating corridors.

Partition was made fairly simple focusing on speed while adding more randomness and controlled variability. This is ideal for this kind of project, focusing on on-fly generation of unlimited terrain. This being said, the zones sometimes look too randomized and it might be beneficial to add more biomes and more complex logic controlling their spreading as was done in the project focusing on generating an island with different biomes using Voronoi diagrams [33]. Nonetheless it might be problematic to implement similar kind of logic without restricting the size of the generated terrain or looping it at some points.

5.2 Future work

5.2.1 Compute shaders

As we discovered in the chapter 4.6, one of the methods slowing down the generation the most is droplet erosion. Since this is based on simple calculations which only modify a float array used as heightmap, it could be reworked from CPU based algorithm to GPU computed shader which has the potential to speed up the process considerably.

The same could be done for generating the heightmap which could solve the slower speed when the generation is based on partitioning.

5.2.2 Replace gradients with textures

Although faster to implement and easier to manage, gradient colors lack the ability to generate truly realistically looking terrain. For that reason, textures could be used, again based on height, slope and partitioning. This would mean reworking the shader and finding - or creating - multiple seamless textures for each partition.

5.2.3 Roads

Using the ridged noise and textures, randomized roads could be easily achieved making the terrain more diverse.

5.2.4 Vegetation

The theory of L-systems was introduced in the chapter 2.6. Unfortunately, due to time constrictions, they could not be implemented.

5.2.5 Expanding biomes

The thesis is using simple partitioning to create some sort of biomes. This could be largely expanded by adding more variables into the biome generation such as randomized weather changes, humidity and so on.

5.2.6 Better user interface and executable

The executable application and the user interface was created only to demonstrate the basic features due to time constrictions. Therefore there is a lot of space to improve it by exposing more functionality that is present in the program to user by expanding the user interface.

5.2.7 Marching cubes

Recreating the project using new technology such as marching cubes would probably be very time intensive, but it would also be very rewarding since it would unlock the possibility of generating better cave systems and more detailed terrain.

Bibliography

1. GOLLENT, Marcin. Landscape creation and rendering in REDengine 3. 2014. Available also from: https://twvideo01.ubm-us.net/o1/vault/GDC2014/Presentations/Gollent_Marcin_Landscape_Creation_and.pdf.
2. GOLLENT, Marcin. Far cry 5 procedral world generation. 2018. Available also from: https://twvideo01.ubm-us.net/o1/vault/gdc2018/presentations/ProceduralWorldGeneration.pdf.
3. EBERT, David; MUSGRAVE, F.K.; PEACHEY, D; PERLIN, K; WORLEY, Steve; MARK, W.R.; HART, John. *Texturing and Modeling: A Procedural Approach: Third Edition*. 2002.
4. FENLON, Wes. Dwarf Fortress creator Tarn Adams talks about simulating the most complex magic system ever. Available also from: https://www.pcgamer.com/dwarf-fortress-creator-tarn-adams-talks-about-simulating-the-most-complex-magic-system-ever/.
5. *Bicubic interpolation* [online]. Wikipedia [visited on 2018-12-09]. Available from: https://en.wikipedia.org/wiki/Bicubic_interpolation.
6. BOURKE, Paul. Interpolation methods. 1999. Available also from: http://paulbourke.net/miscellaneous/interpolation/.
7. Noise, being a pseudorandom artist. Available also from: https://catlikecoding.com/unity/tutorials/noise/.
8. PERLIN, Ken. Improving Noise. *ACM Trans. Graph.* 2002, vol. 21, no. 3, pp. 681–682. ISSN 0730-0301. Available from DOI: 10.1145/566654.566636.
9. GUSTAVSON, Stefan. Simplex noise demystified. 2005. Available also from: http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.
10. ÖBERG, Einar. Ridged Perlin Noise. Available also from: http://www.inear.se/2010/04/ridged-perlin-noise/.
11. BRADLEY, Larry. Fractals. Available also from: http://www.stsci.edu/~lbradley/seminar/fractals.html.

12. MANDELBROT, Benoit B. *The Fractal Geometry of Nature*. Times Books, 1982. ISBN 0716711869.
13. COATS, David. *Fractal Terrain Generation and Erosion for Playful Applications*. Available also from: `https://www.math.hmc.edu/~dyong/math164/2008/coats/presentation.pdf`.
14. OLSEN, Jacob. Realtime Procedural Terrain Generation. 2004. Available also from: `http://web.mit.edu/cesium/Public/terrain.pdf`.
15. RAMSTEDT, Rami; SMED, Jouni. Midpoint Displacement in Multifractal Terrain Generation. In: 2016.
16. PRUSINKIEWICZ, Przemyslaw; LINDENMAYER, Aristid. *The Algorithmic Beauty of Plants: The Virtual Laboratory*. New York: Springer-Verlag, 1990. Available also from: `http://algorithmicbotany.org/papers/abop/abop.pdf`.
17. CURRY, Roger. On the Evolution of Parametric L-Systems. 2008. Available also from: `https://prism.ucalgary.ca/handle/1880/46565`.
18. SHIFFMAN, Daniel. Fractals. Available also from: `https://natureofcode.com/book/chapter-8-fractals/`.
19. WU, Jiang; YANG, Qinke; LI, Yuru. Partitioning of Terrain Features Based on Roughness. *Remote Sensing*. 2018, vol. 10, pp. 1985. Available from DOI: `10.3390/rs10121985`.
20. WORLEY, Steven. A Cellular Texture Basis Function. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1996, pp. 291–294. SIGGRAPH '96. ISBN 0-89791-746-4. Available from DOI: `10.1145/237170.237267`.
21. PATEL, Amit. Polygonal Map Generation for Games. 2010. Available also from: `http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/`.
22. DAVIES, Jason. *Lloyd's Relaxation*. Available also from: `https://www.jasondavies.com/lloyd/`.
23. LAGUE, Sebastian. *Hydraulic-Erosion*. 2019. Version 1.0. Available also from: `https://github.com/SebLague/Hydraulic-Erosion`.

24. VOLYNSKOV, Alexey. Water erosion on heightmap terrain. 2011. Available also from: http://ranmantaru.com/blog/2011/10/08/water-erosion-on-heightmap-terrain/.

25. ALIAGA, Daniel G. Level of Detail: A Brief Overview. 2010. Available also from: https://www.cs.purdue.edu/homes/aliaga/cs535-10/lec-lod.pdf.

26. CUDWORTH, Ann Latham. *Extending Virtual Worlds: Advanced Design for Virtual Environments*. CRC Press, 2016.

27. *Procedural World - Water bodies*. Available also from: http://procworld.blogspot.com/2013/10/water-bodies.html.

28. *Procedure for rivers and lakes: erosion, deposition and lake formation*. Available also from: http://www.entropicparticles.com/procedure-for-rivers-and-lakes.

29. GEISS, Ryan. Generating Complex Procedural Terrains Using the GPU. Available also from: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch01.html.

30. LAGUE, Sebastian. Procedural Cave Generation tutorial. Available also from: https://unity3d.com/learn/tutorials/s/procedural-cave-generation-tutorial.

31. *Cellular Automata Method for Generating Random Cave-Like Levels*. Available also from: http://roguebasin.roguelikedevelopment.org/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels.

32. The matrching cubes. Available also from: http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html.

33. PATEL, Amit. *Polygonal Map Generation for Games*. Red Blob Games, 2017. Available also from: https://www.redblobgames.com/maps/mapgen2/.

34. *Flood fill* [online]. Wikipedia [visited on 2019-03-17]. Available from: https://en.wikipedia.org/wiki/Flood_fill.

35. *Free images and pictures*. Unsplash. Available also from: https://unsplash.com/.

A Project settings

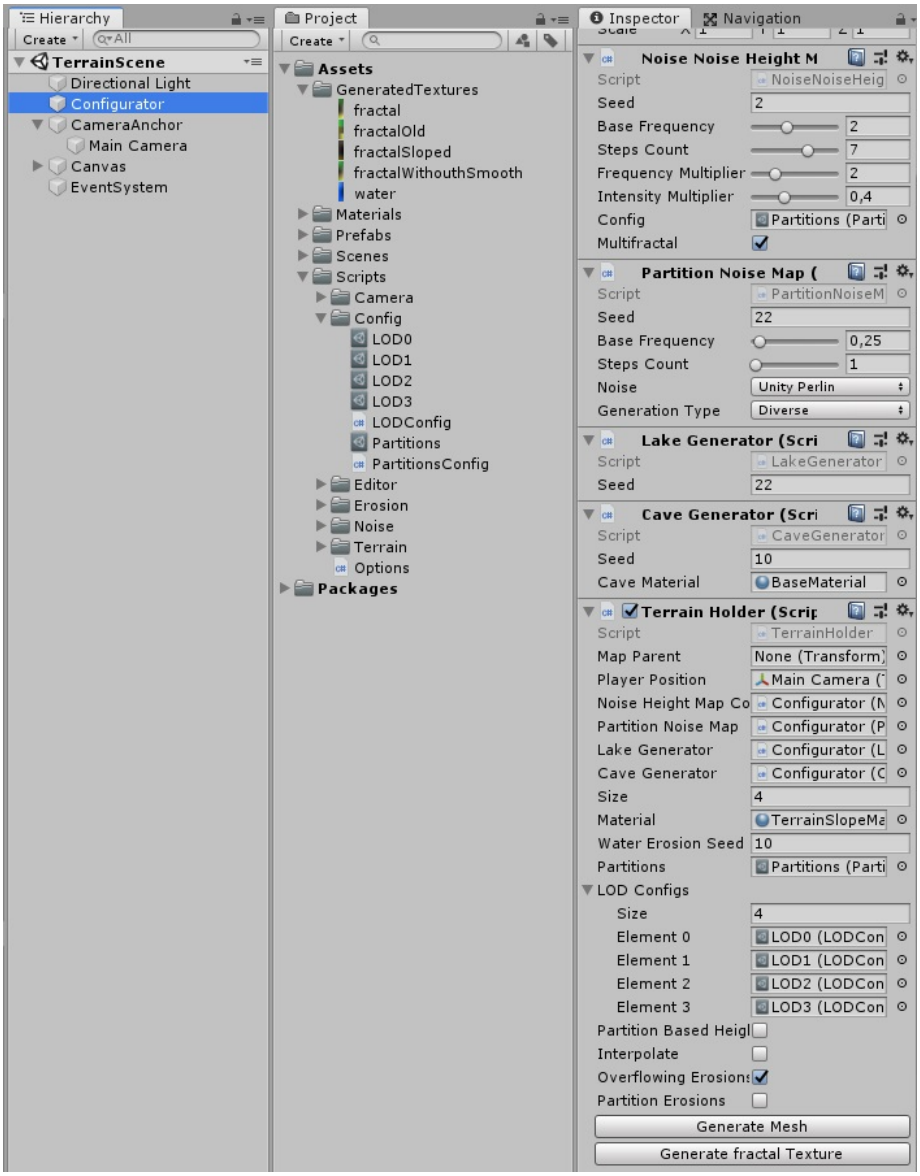

Figure A.1: Base settings of terrain generation

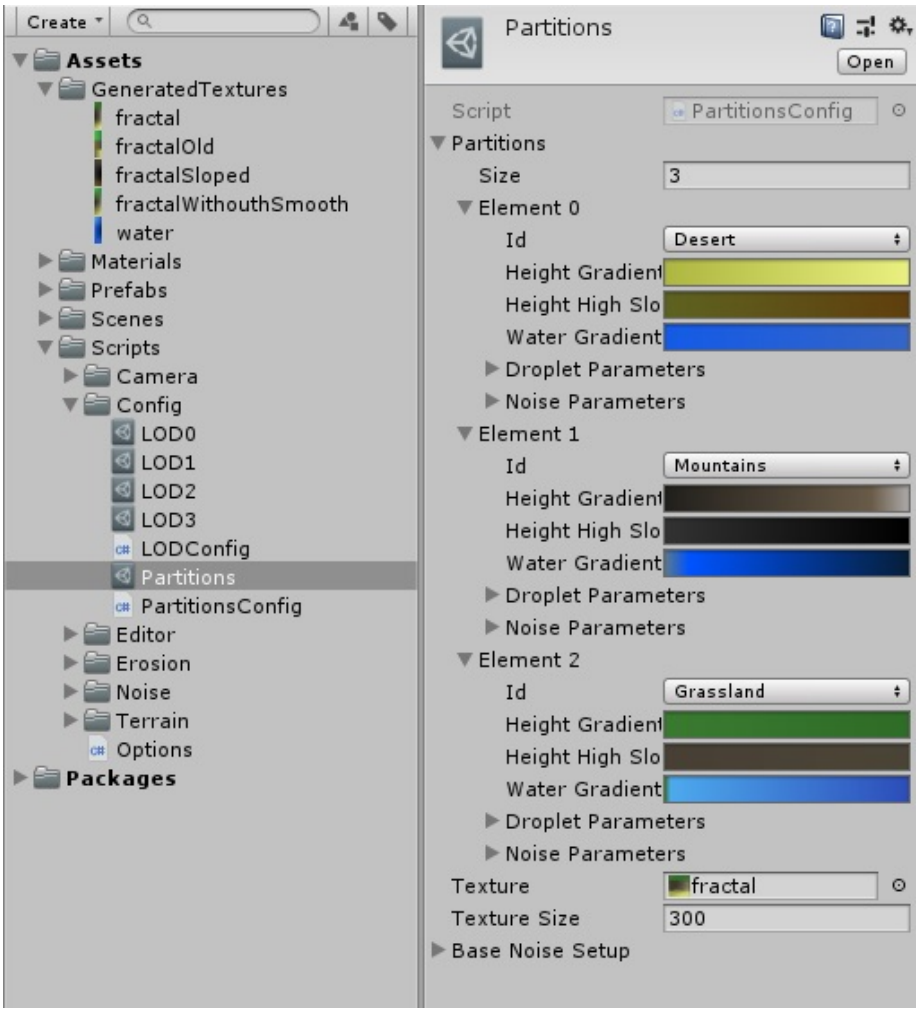

Figure A.2: Partitions settings

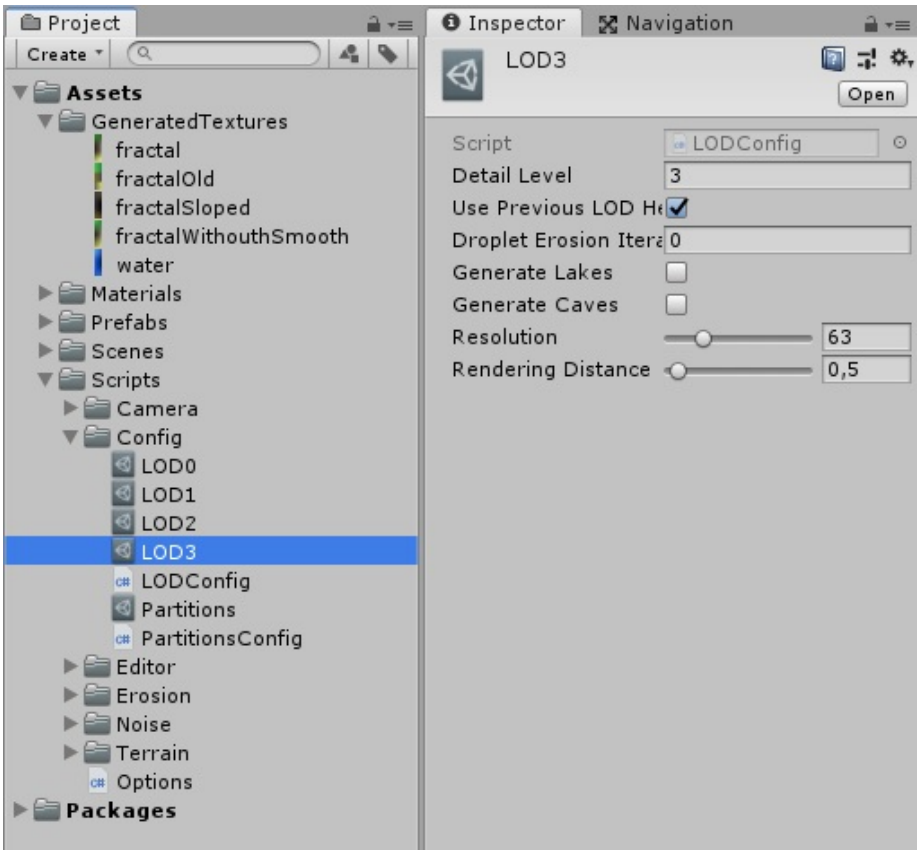

Figure A.3: LOD settings

B High budget projects