

# **Study of procedural terrain generation in plain and spherical surfaces**

LAB University of Applied Sciences

Double degree, bachelor of engineering, information and communications technology

2021

Lia de Belda Calvo

## Abstract

Author(s) Lia de Belda Calvo	Publication type Thesis, UAS	Completion year 2021
	Number of pages 60	
Title of the thesis Study of procedural terrain generation in plain and spherical surfaces		
Degree Double degree, bachelor of engineering, information and communications technology		
Name, title and organization of the client LAB University of Applied Sciences & Universitat Politècnica de Valencia (UPV)		
Abstract <p>This thesis presents a study of different methodologies to approach a procedural terrain generator on a plane and a spherical surface. Processing 3 will be the environment for the implementation of these methodologies. Thanks to this implementation the use of procedural algorithms (in videogames, cinema, and other fields) would be demonstrated. The components of this work are divided into two sections: First, a theoretical resume of the most relevant details of the terrain generation, and second, the implementation for both, the plain and spherical terrain generators, achieving a planetary-looking surface for the spherical case, and an infinite terrain for the plain one. This work is based on previous knowledge over the matter and older implementations of it using Java frameworks.</p>		
Keywords Terrain, generation, noise, procedural, simulation, Processing, Java		

## Contents

1	Introduction .....	5
2	Basics of the terrain generation .....	6
2.1	Main uses .....	6
2.2	Plain terrain generation .....	7
2.2	Spherical terrain generation .....	7
3	Details of terrain generation .....	8
3.1	Pseudo-random and random number functions .....	8
3.2	Diamond-Square algorithm .....	9
3.3	Perlin noise algorithm.....	10
3.3.1	Perlin noise basics .....	10
3.3.2	Fractal Perlin .....	13
3.4	Other noise algorithms .....	14
3.4.1	Poisson faulting.....	14
3.4.2	Fourier synthesis.....	15
3.5	Tessellation .....	15
3.6	Height maps .....	17
3.6.1	Height maps over a plane.....	18
3.6.2	Height maps on a sphere.....	19
3.7	Parameters that influence the terrain.....	19
3.7.1	Granularity.....	19
3.7.2	Strength.....	20
3.7.3	Noise scale.....	20
3.7.4	Persistence .....	20
3.7.5	Lacunarity.....	20
3.7.6	Octaves .....	20
3.7.7	Noise coordinates .....	21
3.7.8	Mapping .....	21
4	IDE approach.....	22
5	Use case: Procedural terrain generator .....	25
5.1	Case definition and main objectives .....	25
5.2	Data model .....	25
5.2.1	Plain terrain geometry.....	27
5.2.2	Sphere terrain geometry .....	28
5.2.3	Perlin noise implementation .....	30
5.2.4	Plain terrain developed methods.....	32

5.2.5	Spherical terrain developed methods.....	36
5.3	Data structures involved.....	39
5.4	Shaders and textures .....	42
5.4.1	Introduction .....	42
5.4.2	Plain terrain shader.....	42
5.4.3	Spherical terrain shader.....	45
5.5	Generations.....	47
5.5.1	Hydro erosion.....	47
5.5.2	Vegetation .....	48
5.5.3	Rivers and lakes .....	49
6	Conclusion.....	52
6.1	Future updates .....	52
6.2	Conclusions.....	53
	List of references .....	54

## Appendices

Appendix 1. Plain terrain renders.

Appendix 2. Spherical terrain renders.

## 1 Introduction

When a team of developers is asked to design a world, with its terrain, lakes, rivers, mountains, etc. the amount of work is overwhelming (Wang et al. 2011). Moreover when we realize that at some point the user will have explored it all, finishing every quest or mission that the team could think about. A way to surpass this is not to design that world, but, letting the computer generate it, infinitely. So, for example, a user playing Minecraft, could in, theory, walk forward and never reach an end, because each chunk of the terrain is procedurally generated, in a coherent way so it feels natural and logical to move from one to another. Another example could be a space exploring game, driving a ship forward into the void will eventually generate planets with unique terrain and ecosystem, this way the user could have a theoretical infinity amount of content (Smith, 2014). Since it began to be possible to represent scenarios graphically using a computer, different research has been done in how to improve the rendering of these virtual 3D spaces. With the increment in the processing power of the computers, it is possible to show more and more complex worlds to the user in real-time. When bounded and pre-rendered scenarios start to be a monotonous habit, the procedural terrain generation appears, contributing to a new experience for both, users and developers. In essence, the procedural terrain generation brings the possibility of creating surface simulations through different techniques which will be explored and commented on the points 4.1 Pseudo-random number functions, 4.2 Diamond-square algorithm and, 4.3 Perlin Noise algorithm. The next subsections want to give a general view of the main uses and terrain generation types that will be explored in this work. This thesis is based on previous experiences with other projects for learning and self-interest using Java, JavaScript, OpenGL, and frameworks like P5.js or Processing 2x, 3x made from scratch. This project will cover the theoretical and practical concepts about terrain generation, such as the main algorithms used, the parameters involved and, the implementation of those ideas into Java.

## 2 Basics of terrain generation

### 2.1 Main uses

The generation of procedural terrain has various uses, going from its implementation in video games, saving large labor costs associated with the detailed design of large scenarios, simulations of hydro erosion, or the rendering of large fields for movies or any type of multimedia presentation were useful (Kim et al, 2018). In the particular case of this work, a more thought-out approach will be used for the first case, mainly due to the amount of detail that can be achieved considering the computational costs. Some examples of these ideas are:

- *TerraGen.*

TerraGen is a non-real-time heightfield landscape synthesis and rendering system. Thanks to its built-in ray tracer system, TerraGen is capable of creating very realistic images, including realistic lighting, atmospheric effects, clouds, water reflection, and terrain shadowing. Although the heightfields synthesized with TerraGen look spectacular and need to be taken as an example of what could be achieved with this kind of technology, the number of different types of natural terrain that can be created with it is somewhat limited (Tegel & Mengotti, 2013).

- *World Machine by Stephen Schmitt.*

Like TerraGen, World Machine is a heightfield synthesis application. However, its main focus is flexibility to create these terrains. Simple real-time 2D and 3D rendering is supported, but this feature is by far not as impressive as TerraGen's (non-real-time) renderer. The user can design terrain by placing and connecting heightfield creation, blending, and transformation nodes in a flow graph system which although is limited in comparison with TerraGen, brings the possibility of seeing the changes in the terrain immediately, saving time in the check process (Zábský, 2011).

- *CryEngine Sandbox 1 & 2 by Crytek.*

Official WYSIWYG level editors for the Crytek game engines, used for the Far cry and Crysis games. These offer an impressive set of tools to aid the level designer. They are capable of loading stored heightfields and simple procedural terrain generation. Local editing is supported through the use of brushes, compared to the two previous, this is by far the most advanced, flexible, and impressive tool. (Tracy & Reindell, 2012.)

Regardless of the use, the tools and all other aspects of terrain generation can be separated into two broad categories: flat terrain (over a 3D plain ) and spherical terrain (over a 3D sphere).

## 2.2 Plain terrain generation

In the case of procedural terrain generation on the plane, a virtually infinite amount of terrain can be achieved (Schneider & Boldte, Rudiger Westermann, 2006). To do it, is possible to start from a rectangular plane of specific dimensions, which will have several points or vertices, modifying the height and the proximity of these vertices to each other. This brings the possibility to achieve a terrain such that, although not perfect, provides a level of realism that could give the impression of natural terrain. This generation of the terrain is based on the algorithms that will be explored in the following chapters, allowing the plane to “move” through a virtual space where the information to generate the terrain is located. Similar to what Minecraft does (C. Duncan, 2019).

## 2.3 Spherical terrain generation

The spherical terrain generation does not allow a virtually infinite amount of terrain. It can be thought of as the modifications over a sphere to achieve the texture of the terrain. It supposes a series of very interesting implementation differences with respect to the flat generation, since now instead of a plane, a sphere is used. It is made up of points or vertices, which, as in flat terrain, will be used to provide the texture, however, the implementation of the algorithms has subtle differences. Using this generation, instead of an infinite terrain, an entire planet is achieved. In a space travel kind of game, this method could be used to once the player has traveled a certain distance, new planets with different biomes could spawn for the user to discover and investigate them. One example could be star citizens (Ahrens et al, 2019 ) or no man’s sky even if this is an example of what not to do (R Tait & L Nelson, 2021).

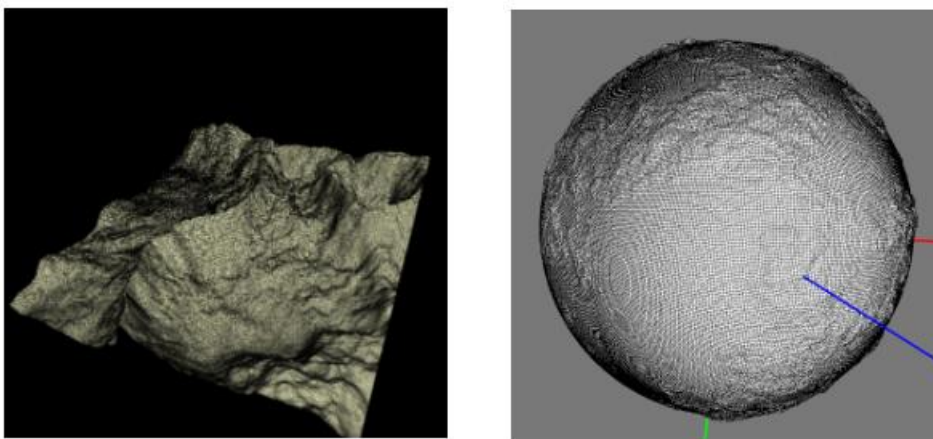


Figure 1. Plain and Spherical terrain, generated with Perlin Noise, and other techniques for this work.

### 3 Details of terrain generation

#### 3.1 Pseudo-random and random number functions

In the computational world, it is impossible to get truly random numbers (Tezuka, 2012). Random processes can only be found in natural processes such as the decay of radioactive isotopes or atmospheric data, therefore it is treated with a series of deterministic algorithms, that is, the sequences of numbers that are obtained will be given by a process marked by steps. The first and perhaps the simplest example is that the random function can be found in all programming languages, which is given by the following expression:

```
A = Constant1  
M = Constant2  
Q = M / A  
R = M % A
```

```
number = ( A * (number mod Q) ) - ( R * floor(number / Q) );
```

```
if (number is negative)  
    number = number + M;
```

```
End
```

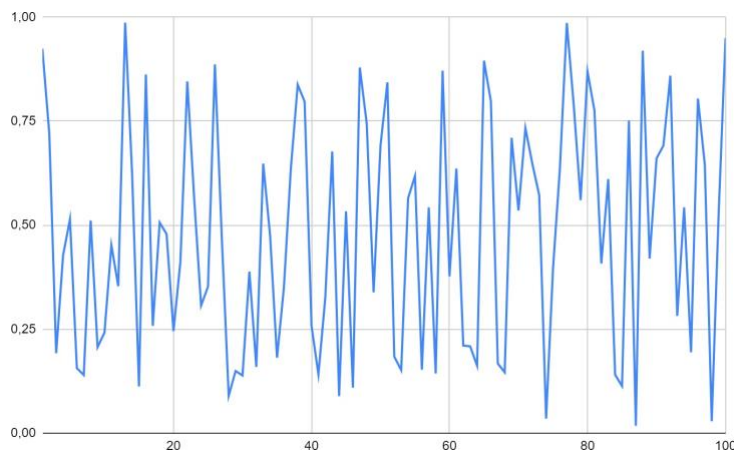


Figure 2 Graph of the random function. 100 samples with values between 0 and 1.

The algorithm, although it has its uses and its utilities, is inappropriate for the result wanted in this work. The terrain must be consistent, however, the random function throws a series of values that do not save any type of relationship between them. As it can be seen in Figure 2, this would generate ridiculous and incoherent situations in addition, in the long run, this type of randomness causes a normal distribution, which is not good for the desired result.



### 3.2 Diamond square algorithm

This algorithm, like Perlin Noise, is based on the idea of obtaining a set of coherent pseudo-random values that can be used for the creation of heightmaps, that is, obtaining a set of values (Generally between -1 and 1) which will be used for different applications, in the case of this work, determine the height of the points or vertices on a plane/sphere. This algorithm was first presented by Fournier, Fussell, and Carpenter in 1982 in SIGGRAPH. (A. Fournier et al, 1982.)

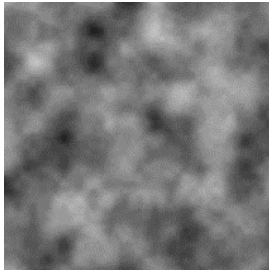


Figure 3 Heightmap example. 8 Octaves, 45% persistence.

The operation of this algorithm is rather simple, given a set of points on a plane, the diamond and square steps will be applied alternately after having initialized the values of the points belonging to the corners with random values.

In the step of squares, for each diamond within the set, the value of the point in the center of the diamond is established as the average of the other 4 points plus a random value.

In the passage of diamonds, for each square within the set, the value of the point in the center of the square is established as the average of the other 4 points plus a random value. (Wang et al, 2010.)

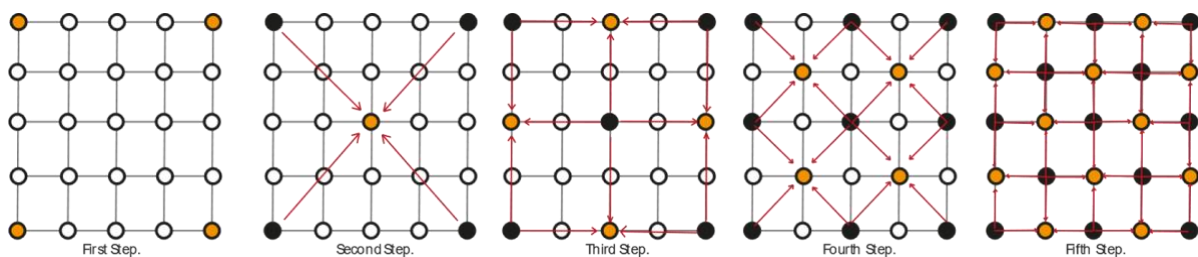


Figure 4 Diamond-square algorithm steps representation.

Although this algorithm is capable of generating height maps, it only does so from some sets that seem to not work correctly in non-square extensions. For an adaptive size plane or spherical generation this method is found inappropriate, although software how TerraGen uses it, achieving more than magnificent results, see Figure 5.



Figure 5 Example of the achievements accomplished by TerraGen using the Diamond-square algorithm. (PlanetSide, 2021)

### 3.3 Perlin noise

This algorithm was developed in 1983 by Ken Perlin at the Mathematical Applications Group, inc, with direct use in the Disney animated film Tron. It was formally described in a paper for SIGGRAPH in 1985 called An Image Synthesizer. For this work, Ken Perlin received an Academy Award for Technical Achievement from the Academy of Motion Picture Arts and Science for his contributions to CGI (Computer Generated Image).

Perlin carried out this development moved by his discomfort with the results of his time of CGI images, being that these were too rough and unnatural. This algorithm allows the creation of more natural textures for surfaces generated by computers. Perlin Noise provides a set of pseudo-random values with consistent differences between them, and with them construct a heightmap. Although its most typical implementations correspond to 2 or 3-dimensional spaces, this algorithm applies to  $n$  dimensions. (Perlin, 1985.)

#### 3.3.1 Perlin noise basics

The implementation of this algorithm may vary, since its publication, there have been different implementations beyond the original, such as the one published in 2001, Simplex Noise and Open Simplex Noise, although those improve performance and correct some errors linked to the original algorithm, for this work the implementation described in 1985 will be the one to be discussed and used (Perlin, 2002). This algorithm presents a computational cost of  $O(2^n)$  being  $n$  the number of dimensions in which the algorithm will work.

This implementation consists of three steps:

- Define a grid with random vector gradients.
- Calculate the cross product of the gradient vectors and their offsets.
- The interpolation of these values.

Next, the details of each of these steps will be explained. It must be considered that this explanation will be for two-dimensional space, but it can be implemented in a similar way for n dimensions. The first step is to define the grid, where each point in the said grid will have a value associated, this value will therefore be a vector (in this case of 2 dimensions) of unit gradient (values that oscillate between 0 and 1). (Perlin, 1985.)

In Figure 6, an example can be seen where the red arrows are a visual representation of the vectors described, for instance, an arrow with values of 1 in X and 0 in Y, represents a completely horizontal arrow pointing to the right.

The second step is the calculation of the cross-product. In Figure 6, an example of gradient grid can be seen. It can be thought of as the computer screen, this is divided into pixels so that, within each cell of the previous grid there is a certain number of pixels. Therefore, each pixel must be assigned a value between -1 and 1 (Green, 2005). To achieve this, for each pixel first locate the cell of the grid to which it belongs. Then, knowing which cell it belongs to, the values of the vectors corresponding to the corners of its cell are also known (since working with a two-dimensional grid, each cell has four corners, but generalizing for n dimensions it would have  $2^n$  corners). Next, a set of new vectors resulting from the displacement from the corner and the pixel being evaluated are calculated. Finally, the cross product between the corners vectors and the displacement vectors is calculated.

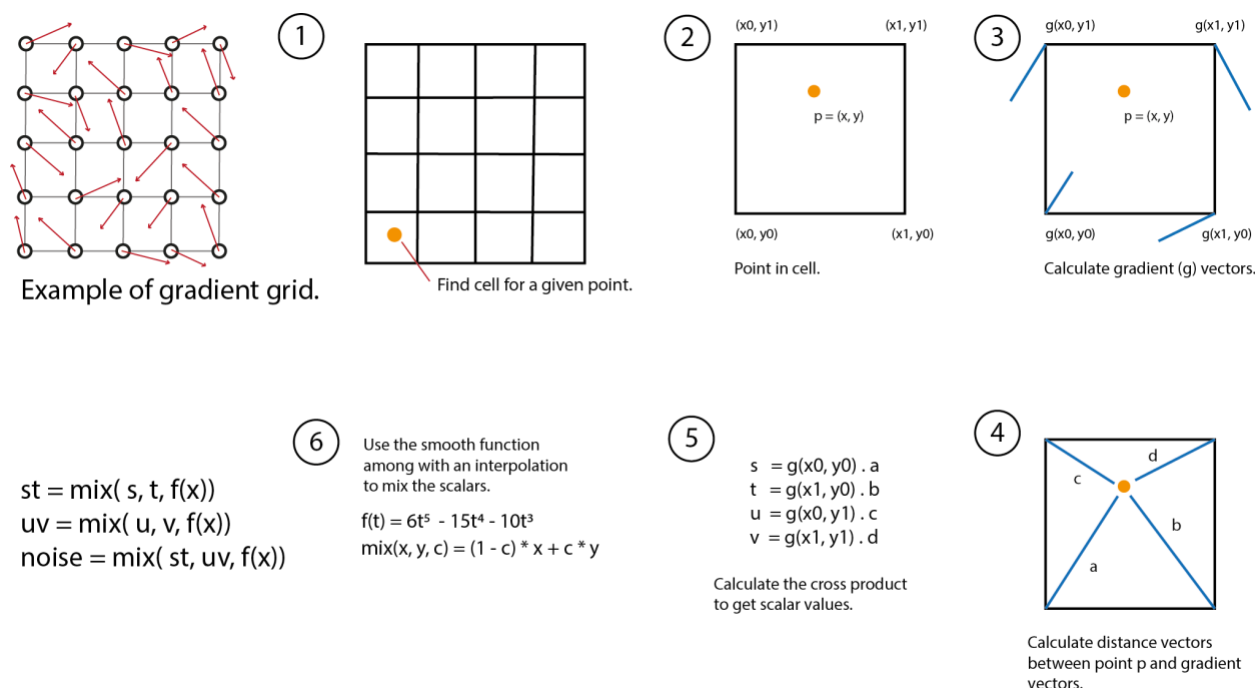


Figure 6 Representation of Perlin Noise steps.

For the third and last step, the interpolation of the crossed products must be performed (in the case of this explanation there will be four, one for each corner and its displacement vector, for n dimensions there will be  $2^n$  crossed products) (Archer, 2011). This interpolation has different approximations such as:

- The simplest and perhaps the easiest way, corresponds to a linear function such that to interpolate a and b values, with a factor that determines the weight c (whose value ranges between 0 and 1):

$$\text{return } (a - b) * c + b$$

- However, as it can be seen in Figure 7, the values returned by the linear interpolation are rough and has no smothering at all. To achieve that appearance so characteristic of this algorithm, this interpolation part comes hand in hand with a smoothing function used on the interpolation step. Ideally, the smoothing function will be such that its first and second derivatives have a value of 0 (Gustavson. 2005). Typically for this it is used a cubic function:

$$\text{Return } (a - b) * ((c * (c * 6 - 15) + 10) * c^3$$

- The downside of this method is that it adds a significant effort to the already heavy computational load. In personal implementations of this algorithm there have been found different functions that provide very pleasant and less expensive results, such as cosine interpolation:

*function Cosine Interpolate (a, b, c)*

$$ft = c * \pi$$

$$f = (1 - \cos(ft)) * 5$$

$$\text{Return } a (1 - f) + b * f$$



Figure 7 Comparison of the linear (left) cubic (middle) and cosine (right) interpolation (Aran D.).

As it has been pointed out these, are not the only available functions to implement in the interpolation step, there are plenty of them and in each implementation of the algorithm, it has to be taken into account different aspects like the overall performance, cost, and computational complexity. Ideally, every implementation should provide an interpolation function such that the first and second derivative equals zero, but as it has been said, that is not always an option.

### 3.3.2 Fractal Perlin

Fractal Perlin is the process of adding multiple layers of Perlin Noise values to create a combined one that will show a more complex and sophisticated appearance and self-similar properties (B. Mandelbrot, 1982).

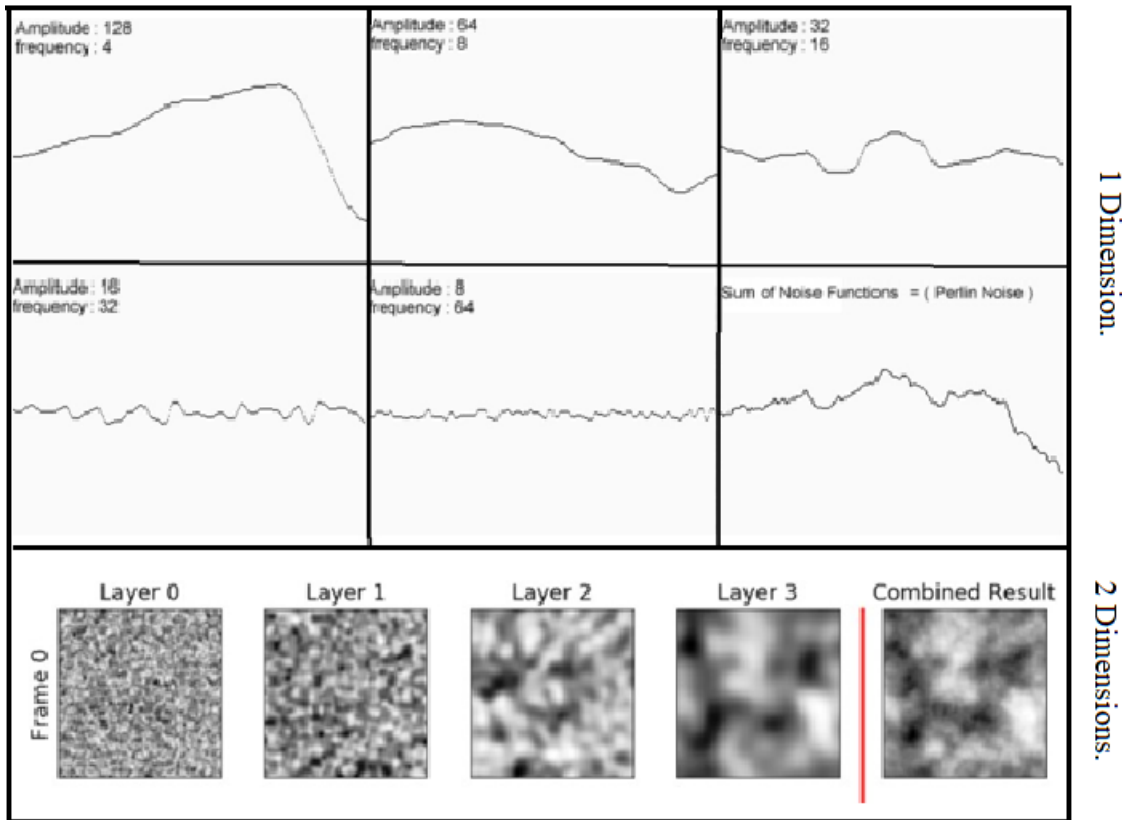


Figure 8 Representation of different noise functions with different amplitudes and frequencies ends up into one final detailed noise. One dimension, Similar effect happens in two dimensions.(Aran D,).

In Figure 8 can be seen how the lower amplitudes the lower the general change that layer produces in the final shape, and higher frequencies produce more pronounced changes. This process of course also happens in higher dimensions as can be seen also in Figure 8. Although the details of the Perlin noise parameters will be presented and discussed in point 4.7 parameters which influence the terrain, when adding together different noise functions, there is the question of exactly what amplitude and frequency to use for each one. In Figure 8 it is used twice the frequency and half the amplitude for each successive noise function added (in both one dimensional and two dimensional Perlin). This is the standard in almost every Perlin Noise implementation. A single number is used to specify the amplitude of each frequency. This value is known as Persistence and it has the following relationship:

$$\text{frequency } (f) = 2^n$$

$$\text{amplitude } (a) = \text{persistence}^n (p)$$

Where n is the nth noise function being added (starting from 0) and persistence goes between 0 and 1.

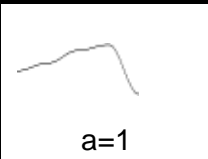
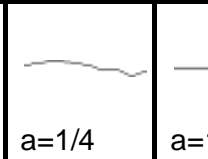
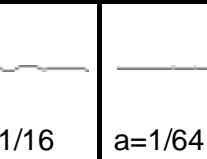
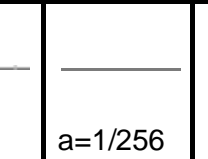
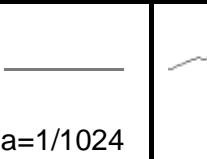
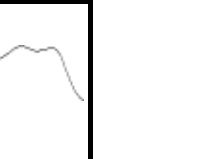

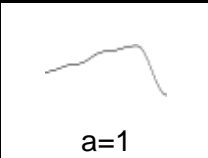
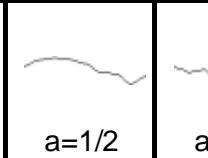
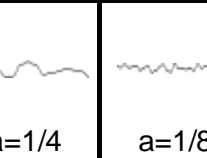
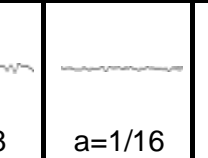
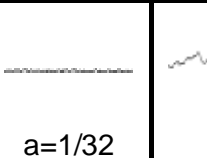


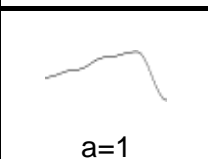
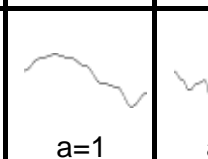
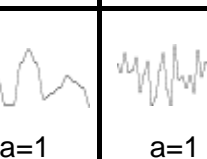
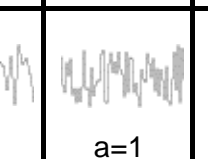
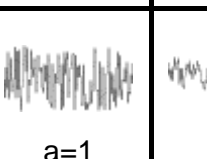
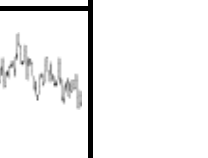
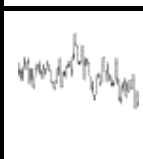
	f=1	f=2	f=3	f=4	f=5	f=6	result
p=¼	 a=1	 a=1/4	 a=1/16	 a=1/64	 a=1/256	 a=1/1024	
p=½	 a=1	 a=1/2	 a=1/4	 a=1/8	 a=1/16	 a=1/32	
p=1	 a=1	 a=1	 a=1	 a=1	 a=1	 a=1	

Table.1 Different noise function and result changing with different amplitudes and frequencies (Aran D, ) .

### 3.4 Other noise algorithms

Although algorithms like Diamond-Square or Perlin noise are the most visible ones when discussing about procedural textures, they are not the only ones. Other options as the Poisson faulting or the Fourier synthesis are based on more abstract concepts rather than a set of operations applied to a group of nodes as the Diamond and Perlin algorithms do.

#### 3.4.1 Poisson faulting

It was first introduced in Dr. Dobb's Journal (R. KRTEN, 2001). Starting from flat terrain, this algorithm generates in an iteratively way random vertical faults as lines, vertices on one side or another of the line are displaced upwards or downwards, depending on the distance to the fault. In the long term, this method can provide a semi-complex landscape with rich mountains and valleys. This method can be generalized to other geometries such as circles or curves, it can also be extended to a spherical geometry to achieve fractal planets as mentioned in the rendering of stochastic models (Fournier et al, 1982). In general, this algorithm provides an acceptable result but with two major inconveniences, first, it cannot be applied in order to achieve an infinite terrain, for every new "chunk" of terrain the algorithm will have to start again. The second problem directly attached to the previous is the computational cost,  $O(N^3)$  Where N is the width or height of the heightfield (measured in vertices or points). Although it isn't an appropriate algorithm for the goals searched in this work, this technique is used in commercial heightfield applications.

Another Poisson faulting kind of algorithm is the one called “particle deposition”. As it names points, this technique involves a simulation of dropping particles on a flat plane. In this idea over the heightfield, many droplets will fall, rolling through the height field surface, and carrying a small amount of sediment until a local minimum is found. Then in that point, the sediment will be released increasing the height of that point by a small amount. This technique resembles in some way thermal weathering. This algorithm works correctly for volcanic-like terrains.

### 3.4.2 Fourier Synthesis

Also called approximation with trigonometric polynomials. This method uses some rules to generate coefficients for sin and cos functions, summing the Fourier series the height on a point of the terrain is obtained, then in order to know all the heights in the heightfield it is used the Fast Fourier Transform (FFT). This method accomplishes removing some of the artifacts presented in the result of other algorithms such as the Diamond-Square, but at the cost of much bigger time requirements. Terrain generated with this method will present characteristics such as periodicity and a computational cost of  $O(N^2 \log N)$ . This periodicity can be an advance or a problem depending on the goal, for instance, this technique suits well with the generation of a smooth and old terrain (A. Mastin et al, 1987). Also, the possibility of extending the results of this algorithm in the creation of an infinite terrain is even though possible, much less straightforward than the ones accomplished with others such as Perlin Noise.

### 3.5 Tessellation

Tessellation is the process to cover an n-dimensional shape in individual and indivisible pieces called tiles, those have no overlaps and no gaps, in real life this can be seen in puzzles, mosaics, or floor tiling using different kinds of materials such as cemented ceramic shapes.

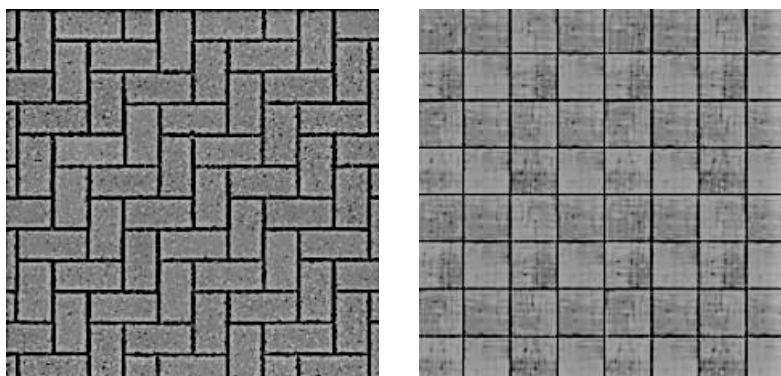


Figure 9 Examples of a tiled surface.

In computer graphics, tessellation is used to form and control datasets of polygons. Making it possible to represent objects in a scene and divide it into suitable structures for rendering. Arbitrary 3D bodies are usually too difficult to analyze directly (examples could be human bodies, animals, general 3D objects such as furniture, or terrain), because of this, tessellation is often used to decrease the complexity of those objects into simpler ones like polygons.



In this work, there are two scenarios at first very different but at the end practically the same. On one hand there is the plain tessellation and in the other the spherical one. In both cases, the tiles used in the tessellation process will be the triangles, as they are the most typical approach in almost every kind of shape in computer graphics.

In a plain tessellation, after disposing of the points in a grid position, each point is joined with the point to the right, up, and the one in diagonal to the right (considering that the startpoint is in the left-bot corner). Another way of constructing this triangle mesh is the typical clockwise method, in this case starting from the top left corner of the mesh, the vertices are joined following clockwise movement, as can be seen in Figure 10 right. For a square divided with two triangles, there are 4 vertices and therefore the order to join them will be  $a, b, d, c, a$ . generalizing it to an  $n \times n$  grid where  $i$  is the actual vertices, the order will be:  $i, i+1, i+n+1, i+n+1, i+n, i$ .

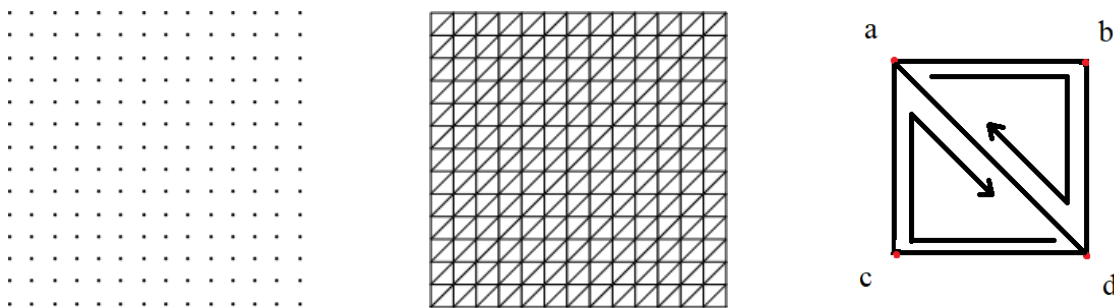


Figure 10 Example of a 14x14 point grid before and after triangle tessellation.

In the spherical case, there are multiple ways to construct the model. In general terms it can be identified in two main categories, using the spherical coordinate system (polar system), or constructing the sphere from a polygon.

Each one presents a different set of benefits and detriments, for instance, it can be seen in Figure 11 left how the geometry of the polar system is deformed in the pole in such a way that the density of the vertices is huge compared to the equator. On the other hand, the polygon approach implies several implementation difficulties as it can be intuited from Figure 11 right.

At the end each one is perfect according to the necessities of the project and the circumstances which it involves, for instance the polar system is more efficient and easier to implement, the polygon method provides much more detail and less artifacts, but with the cost of much more of both memory and computational complexity among with a harder implementation. Both the polar system and the polygon approach will be discussed in the use case, explaining the pros and cons of each other among with other methods, and finding a perfect fit for them.



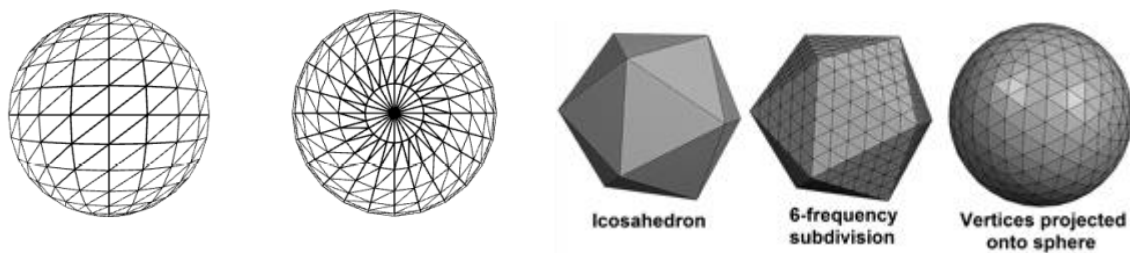


Figure 11 Comparison between polar sphere construction method (left) and polygon based sphere (right), in this case the base polygon is an Icosahedron.

### 3.6 Heightmaps

A heightmap or heightfield is the term used to describe a data structure which will be used in computer graphics to determine the elevation of a given point, this data structure corresponds typically to an array of float numbers, in which each position has a relation one to one to a given pixel. This elevation could be used in a bump map to create shadows in a material, to create a textured surface for a particular material, or most commonly to be applied over a 3D mesh to generate terrain (De Carpentier, 2008).

Heightmaps are widely used in terrain rendering software and modern video games. They are an ideal way to store digital terrain elevations; compared to a regular polygonal mesh, they require substantially less memory for a given level of detail. Most modern 3D computer modeling programs are capable of using data from heightmaps in the form of bump, normal, or displacement maps to quickly and precisely create complex terrain and other surfaces (S. Ebert et al. 2003).

In the earliest games using software rendering, the elements often represented heights of columns of voxels rendered with ray casting, simulating 3D spaces when in reality's everything was being rendered in 2D. In most newer games, the elements are represented with the height coordinate of polygons in a mesh in a true 3D environment. (POV-RAY.)

Heightmaps are commonly represented on greyscales see Figure 12, where white is the highest point and black the lowest. When the height field is stored in the form of an image the resolution of the height field is influenced by two factors: the resolution of the image and precision of the color/index values. The size of the image determines the resolution in terms of width and height. The bigger the image the bigger the total number of triangles it will have the tessellation and therefore the smoother the result. The resolution of the color/index value determines the resolution along the depth or z component. A height field made from an 8-bit image can have 256 different height levels while one made from a 16-bit image can have up to 65536 different height levels. Thus, the second height field will look much smoother in the z component if the height field is created appropriately. There are eight or possibly nine types of files that can define a heightfield: gif, tga, pot, png, pgm, ppm, tiff, jpeg, and possibly sys which is system-specific (for instance. Windows BMP or Macintosh Pict) format file. (POV-RAY.)

Most scenes requiring terrain will make use of a heightfield. This is typically because heightfields are the most efficient way to describe the numerous tiny fluctuations of a typical landscape at design time. When a heightfield's design is completed, it can be transformed or converted to a more efficient polygon mesh, it must be taken into account that not all heightfields benefit from this conversion (for instance) those containing only a few connected areas of similar slope). Although there exist methods to represent the heightfields without a typical mesh conversion (Albersmann. et al, 1999).

But heightfields have their limits (Feldmann & Hinrichs, 2012). By ignoring specific width and breadth information, (leaving only height values), each height value must occupy a unique 2D location when viewed from above. The important corollary to this is that no 2D location in the heightfield can have more than one height value associated with it. This effectively rules out using heightfields to model rocks, asteroids, bridges, caves, or terrain with slopes equaling or exceeding ninety degrees. Some objects, however, can be convincingly modeled with two or more heightfields carefully placed.

Although this fact makes heightfields unsuitable for many types of objects, it also makes heightfields very easy to work with, since the height values can effectively be treated using standard bitmap-editing interfaces that work with 2D images. This feature also makes it easy to paint heightfields by mapping bitmaps onto them. All that's required is to project the bitmap onto the heightfield's ground plane, and every point on the height field will be textured, also the heightfields can be modified and edited with numerous tools and programs (Yalcin, M. A., & Capin, T. K, 2009; Bradbury et al, 2014).

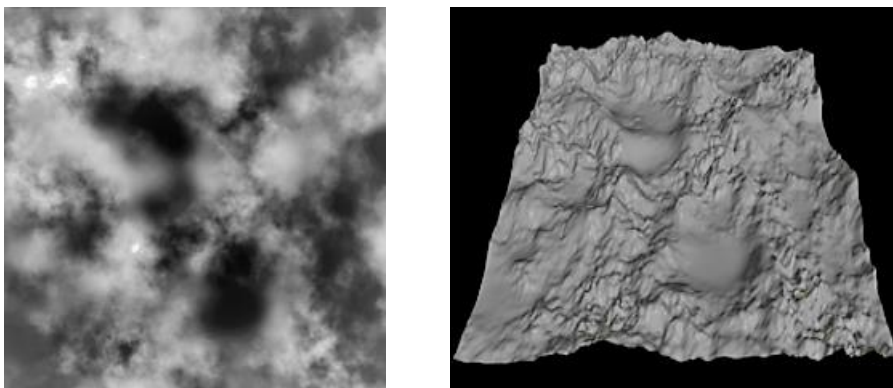


Figure 12 Relation between the heightmap and the terrain generated from it.

### 3.6.1 Height maps over a plane

In the plain case the implementation or use of the heightmap is rather simple (Garland & Heckbert, 1995), as has been said, the heightmap stores a set of decimal values which corresponds one to one to the vertices or points which forms the plain, then the given value is mapped between a minimum and maximum height and the result will be the Z component of the givenpoint.

### 3.6.2 Height maps on a sphere

For the sphere case, there is a bit more complexity, instead of having a one-to-one relationship between the image (heightmap) and the set of points. The value of the heightmap for a given point is calculated using the parametric sphere equations or a UV transformation, making the heightmap in a spherical surface instead of a plain one in a similar way to how an earth map works (Barmore, 2002). Although the one-to-one relationship is not impossible and can be used with the sphere too. Instead of making the result the Z component, now it is the magnitude of the point, being this equal to its current value plus the mapped result. Another method to consider is to use the Perlin Noise algorithm in 3 dimensions using the normalized vector between the point being evaluated and the center of the sphere as inputs.

## 3.7 Parameters that influence the terrain

### 3.7.1 Granularity

In the Perlin Noise context, the granularity can be interpreted as the level of detail that a heightmap is able to provide. In general, this is also related to the number of octaves or layers that forms the heightmap, being a big granularity a fuzzier result and small granularity a detailed one. (S. Ebert et al, 2003.)

In the terrain context, this parameter is very useful because, depending on the kind of biome that needs to be generated, the changes on the granularity would be crucial, for instance, a mountainous biome needs the level of detail corresponding to the small granularity in order to represent the rocks, cliffs, slopes, etc. but a desert one formed mainly by dunes doesn't need that many details and can be done with a bigger granularity.

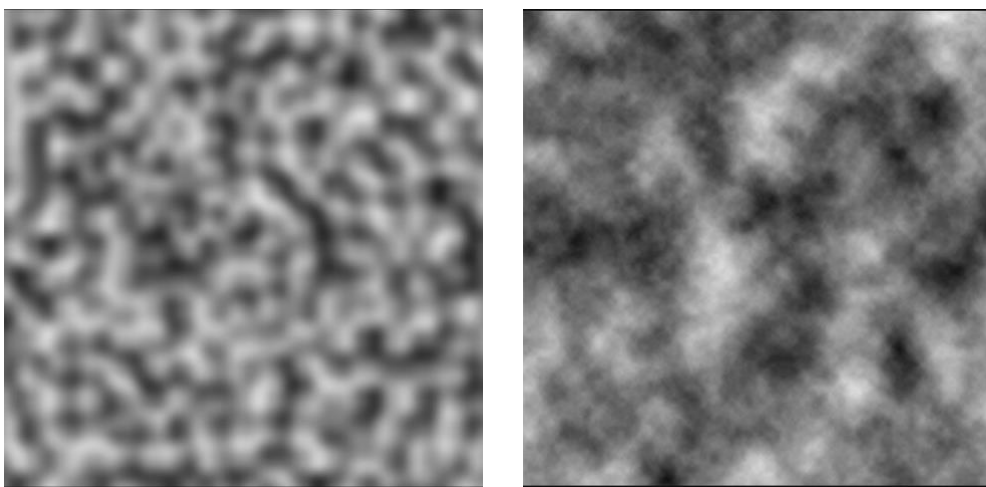


Figure 13 Comparison of two heightfields generated with Perlin noise, one with big granularity (left) and other with small granularity (right).

### 3.7.2 Strength

The strength or amplitude refers to the importance of a given layer of the noise in regard to the others (S. Ebert et al, 2003). As it has been shown in Figure 8 and Table 1, the more the amplitude the more that layer will influence the result, in general, the higher amplitudes are for the firsts layers, which have a low frequency, this means that the firsts layers will have an important impact in the shape of the function but will not be the ones responsibilities of the detail. This can also be seen in picture 18 where the first heightmap provides a general shape but fuzzy.

### 3.7.3 Noise scale

Another way to control the output of the Perlin Noise algorithm is the scale. Consider that this algorithm works in an infinite n-dimensional space in such a way that the coordinates given in the input are not really important, but the distance between points is. In general applications a scale with values between 0.005 and 0.03 works fine (S. Ebert et al, 2003), considering that with smaller distances the smoother the final output will be.

### 3.7.4 Persistence

A multiplier that determines how quickly the amplitudes diminish for each successive octave in a Perlin-noise function. The amplitude of each successive octave is equal to the product of the previous octave's amplitude and the persistence value (S. Ebert et al, 2003; Libnoise). Increasing the persistence produces "rougher" Perlin noise, on the other hand lower persistence tend to give a plainer terrain which resembles more to a valley or a smooth hill.

### 3.7.5 Lacunarity

A multiplier that determines how quickly the frequency increases for each successive octave in a Perlin-noise function. The frequency of each successive octave is equal to the product of the previous octave's frequency and the lacunarity value (S. Ebert et al, 2003; Libnoise). If the persistence makes each successive octave to have less impact in the result, the lacunarity is the responsible of adding the details, thanks to the increment in the frequency of each octave the result can be rich in details, both persistence and lacunarity must have an equilibrium in order to represent a coherent terrain.

### 3.7.6 Octaves

One of the coherent-noise functions in a series of coherent-noise functions that are added together to form Perlin noise. These coherent-noise functions are called octaves because each octave has, by default, double the frequency of the previous octave (S. Ebert et al, 2003; Libnoise). Musical tones have this property as well; a musical C tone that is one octave higher than the previous C tone has double the frequency. The number of octaves control the amount of detail of Perlin noise. Adding more octaves increases the detail of Perlin noise, with the added drawback of increasing the calculation time.

### 3.7.7 Noise coordinates

As said before, Perlin Noise provides an infinite n-dimensional space with values between -1 and 1 for each point evaluated. This point will be defined generally in 1, 2, or 3 dimensions, and with the values obtained from the algorithm on those coordinates the terrain will be modified. As commented in the scale section, the important thing is the space between the points which are being calculated and not the points themselves. This is a direct consequence of a coherent terrain, for example for a particular biome everywhere should look more or less the same regarding the details.

### 3.7.8 Mapping

- Sin/Cos based functions.

In some cases, after obtaining the noise value for a given point, it can be interesting and useful to map that value. Plugging that value into a given function to modify the noise value in order to conserve the coherent pseudo-random values of the noise, ending up with complex transformations. One example of this mapping function are the sin/cos functions. In Figure 14 is represented a classical sine function, and a modified sine function, this one will be used in the use case to map the noise values in order to create cliffs in the plain terrain.

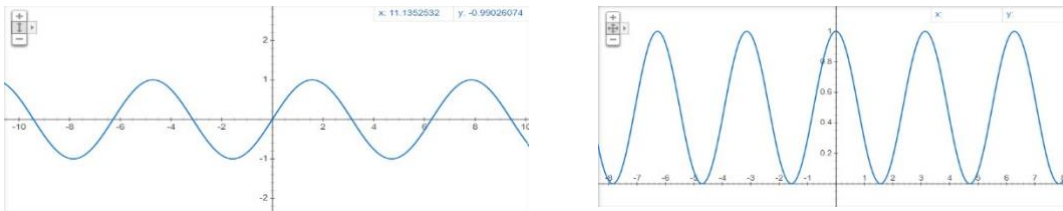


Figure 14 Graph representations of the function  $f(x) = \sin(x)$  (left) and  $f(x) = 1 - |\sin(x)|^2$  (right)

- Conversion of values.

The following methods are an extension of the concepts explained in point 4.3.2 Fractal Perlin Noise, in fact they are procedures which will be applied after calculating the fractal Perlin. First the Billowy turbulence (De Carpentier & Bidarra, 2009) presents a more 'billowy' aspect, defining a more eroded and less rigid terrain when applied as a height map. To apply the Billowy turbulence the only step required is to return the absolute value of the noise value.

$$\text{BillowyNoise} = | \text{PerlinNoise} (x, y) |$$

The next turbulence which can be applied in order to modify the noise value will be the Rigid turbulence or rigid noise (De Carpentier & Bidarra, 2009). It isn't anything more than the complement of the Billowy turbulence.

With Rigid turbulence a less eroded and more sharp terrain will be generated. To apply the Rigid turbulence first calculate the Billowy turbulence, and the return one minus that value.

$$\text{RigidNoise} = 1.0 - \text{BillowyNoise}$$

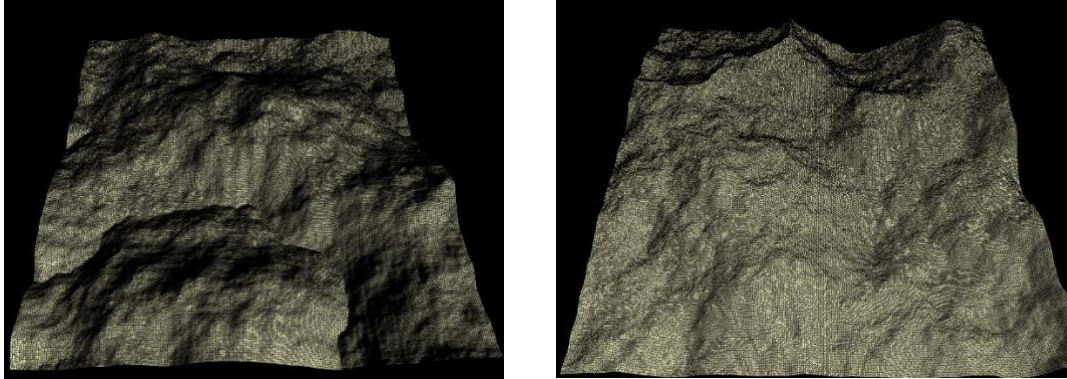


Figure 15 Plain terrain generated for this work. 128x128 grid with Billowy (left) and Rigid (right) turbulence.

#### 4 IDE approach

In computer engineering an integrated development environment (IDE) is the software for building applications that combines common developer tools in a single graphical user interface (GUI). An IDE typically is formed with a source code editor, a local build automation and a debugger.

For this work the chosen IDE was Processing, it was first started in Spring 2001 by Ben Fry and Casey Reas, along with Andres Colubri, Elie Zananiri and Samuel Pottinger. Processing is a flexible software sketchbook and a language to code within the context of the visual arts. Since 2001, Processing has promoted software literacy within the visual arts and visual literacy within technology. It is an open-source software based on Java, it provides outputs in multiple formats such as 2D, 3D, PDF or SVG, uses technologies such as OpenGL integration for accelerated 2D and 3D and it is available for GNU/Linux, Mac OS X, Windows, Android and ARM.

Processing is a descendant of the Design By Numbers (DBN) project and other initiatives at the Aesthetics + Computation Group (ACG). DBN is a simplified programming language that was developed to teach the structure and interpretation of software visually. Design By Numbers was created by John Maeda and accompanied a book of the same name. While at the ACG, Ben and Casey were involved in the development and maintenance of the DBN software and courseware, and that experience provided the basis for the Processing project. (Reas & Fry, 2014.)

Processing code is converted to straight Java code (using the "preprocessor") when hitting the Run button. This also makes it possible to embed other Java code in the sketches, or use the core.jar file from the Processing distribution to develop sketches with other environments. For instance, Eclipse.

Processing also relies on (and is indebted to) other open projects, namely:

- jEdit Syntax package, which is public domain.
- The ECJ Compiler from the Eclipse project, which uses the Eclipse license.
- Java Native Access (JNA) project, released under the LGPL.
- Release 0149, use a slightly modified version of launch4j to create processing.exe on Windows.
- The quaqu library, which makes Java applications look more like Mac applications when running on OS X.

In summary, the Processing Development Environment (PDE) makes it easy to write Processing programs. Programs are written in the Text Editor and started by pressing the Run button. In Processing, a computer program is called a sketch. Sketches are stored in the Sketchbook, which is a folder on your computer.

Sketches can draw two- and three-dimensional graphics. The default renderer is for drawing two-dimensional graphics. The P3D renderer makes it possible to draw three-dimensional graphics, which includes controlling the camera, lighting, and materials. The P2D renderer is a fast, but less accurate renderer for drawing two-dimensional graphics. Both the P2D and P3D renderers are accelerated if the computer has an OpenGL compatible graphics card.

The capabilities of Processing are extended with Libraries and Tools. Libraries make it possible for sketches to do things beyond the core Processing code. There are hundreds of libraries contributed by the Processing community that can be added to the sketches to enable new things like playing sounds, doing computer vision, and working with advanced 3D geometry. Tools extend the PDE to help make creating sketches easier by providing interfaces for tasks like selecting colors.

Processing has different programming modes to make it possible to deploy sketches on different platforms and programs in different ways. The Java mode is the default. Other programming modes may be downloaded. (Reas & Fry, 2014.)

One inevitable question is why to aboard this work with a Java-like technology when there are other options like Unity with C# and C++, the answer is that this work is not a technical demonstration of the capabilities of those languages or the potential peak performance that could be achieved, but instead a catalyst to explore the ideas of the procedural terrain generation and the capabilities of algorithms like Perlin Noise.

Between the simplicity that some languages like Python, Ruby or others scripting languages could offer, and the performance that more advanced languages (C-style) could offer, is Java, moreover with Processing Java counts with a new graphics and utility API along with some simplifications which simplifies several aspects of the development phase.





## 5 Use case: Procedural terrain generation system

### 5.1 Case definition and main objectives

The scope of this work is to develop a program capable of generating a complex and coherent terrain for both, plain, and spherical surfaces. To achieve these objectives a couple of problems have to be solved, for instance: First, a set of operations must be defined for the program to render the different objects that will make the scene. Second, the geometries for both the plain and spherical terrain must be implemented, establishing how the different terrains will be rendered. Then, the noise (Perlin noise) among its modifications, and the different function which support it must be developed and implemented. Also, the data structure that will provide the functionality to the whole project has to be designed among the different shaders that will contribute strongly to the aesthetic and visual details of the landscapes. In the end, the plain terrain must be able to provide landscapes formed by regular mountains, mountain ranges, valleys, cliffs, coast, oceans, and rivers, with a dynamical light system, and most importantly, a virtual infinite terrain extension. The spherical terrain must create a planetary-looking object, in which different continental masses can be distinguished, among oceans, islands, archipelago, and atmosphere systems, and finally, a cloud system which changes over time.

### 5.2 Data model.

This section will include some of the key points in the development and implementation of the ideas and concepts presented in the theoretical part of this work. The data model will be then, the model or skeleton of the project.

The first problem to be solved in order to accomplish a decent terrain generation system was how to represent the terrain in the first place. In the theoretical part it has been pointed out that the terrain will be represented with vertices in a 3D space, to accomplish this the following methods were used:

- *PShape*: Processing data type for storing shapes (collections of points in 2D or 3D, which can make a closed or open shape). Before it is used, it must be loaded with *loadShape()* (if it is a precomputed shape, for instance, a blender export) or created with *createShape()*. The *shape()* function is used to draw the shape to the display window. Processing can currently load and display SVG (Scalable Vector Graphics) and OBJ shapes. OBJ files can only be opened using the P3D renderer. The *loadShape()* function supports SVG files created with Inkscape and Adobe Illustrator. It is not a full SVG implementation but offers some straightforward support for handling vector data.
- *shape()*: Draw shapes to the display window. Shapes must be in the sketch's "data" directory to load correctly. Processing currently works with SVG, OBJ, and custom-created shapes (this will be the ones used in this work). The shape parameter specifies the shape to display, and the coordinate parameters define the location of the shape from its upper-left corner. The shape is displayed at its original size unless the parameters specify a different size. The *shapeMode()* function can be used to

change the way these parameters are interpreted.

- *createShape()*: The *createShape()* function is used to define a new shape. Once created, this shape can be drawn with the *shape()* function. The basic way to use the function defines new primitive shapes. One of the following parameters is used as the first parameter: ELLIPSE, RECT, ARC, TRIANGLE, SPHERE, BOX, QUAD, or LINE. The parameters for each of these different shapes are the same as their corresponding functions: *ellipse()*, *rect()*, *arc()*, *triangle()*, *sphere()*, *box()*, *quad()*, and *line()*.

Custom, unique shapes can be made by using *createShape()* without a parameter. After the shape is started, the drawing attributes and geometry can be set directly to the shape within the *beginShape()* and *endShape()* methods.

The *createShape()* function can also be used to make a complex shape made of other shapes. This is called a "group" and it's created by using the parameter GROUP as the first parameter.

- *beginShape()* & *endShape()*: Using the *beginShape()* and *endShape()* functions allow creating more complex forms. *beginShape()* begins recording vertices for a shape and *endShape()* stops recording. The value of the kind parameter tells it which types of shapes to create from the provided vertices. With no mode specified, the shape can be any irregular polygon. The parameters available for *beginShape()* are POINTS, LINES, TRIANGLES, TRIANGLE\_FAN, TRIANGLE\_STRIP, QUADS, and QUAD\_STRIP. After calling the *beginShape()* function, a series of *vertex()* commands must follow. To stop drawing the shape, call *endShape()*. The *vertex()* function with two parameters specifies a position in 2D and the *vertex()* function with three parameters specifies a position in 3D. Each shape will be outlined with the current stroke color and filled with the fill color.
- *vertex()*: All shapes are constructed by connecting a series of vertices. *vertex()* is used to specify the vertex coordinates for points, lines, triangles, quads, and polygons. It is used exclusively within the *beginShape()* and *endShape()* functions.
- *size()*: Defines the dimension of the display window width and height in units of pixels. In a program that has the *setup()* function, the *size()* function must be the first line of code inside *setup()*, and the *setup()* function must appear in the code with the same name as the sketch folder. The built-in variables width and height are set by the parameters passed to this function. For instance, running *size(640, 480)* will assign 640 to the width variable and 480 to the height variable. If *size()* is not used, the window will be given a default size of 100 x 100 pixels.
- *width()* & *height()*: System variables that store the width & height of the display window. These values are set by the first and second parameter of the *size()* function.

### 5.2.1 Plain terrain geometry

To create the plain itself a collection of dots in a 3D space were defined according to the width and height of the windows. The shape was generated using the *PShape()*, *beginShape()*, *endShape()* and *vertex()* functions. For the *beginShape()* function it was used the TRIANGLE\_STRIP parameter obtaining the results shown in Figure 16 left.

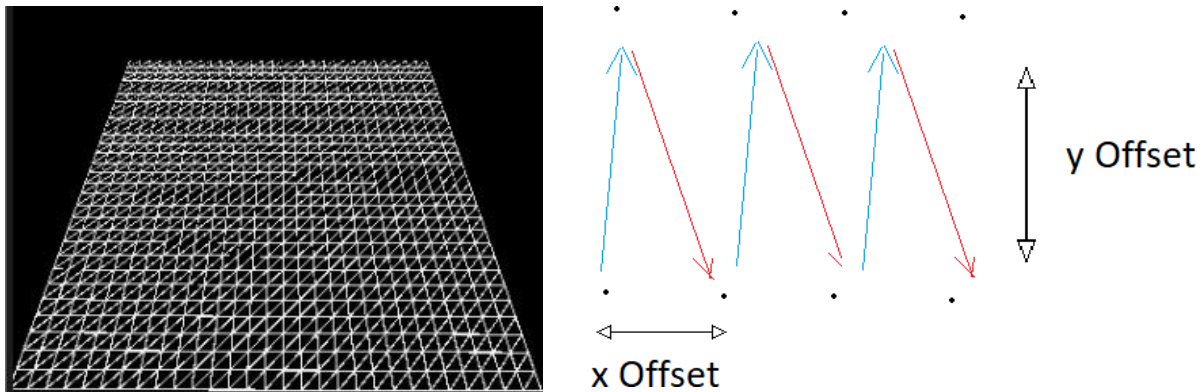


Figure 16 Simple plain (left). The vertices are joined using TRIANGLE\_STRIP parameter.

The space between each vertex will be called offset, this offset can be easily calculated as follows: For the x coordinate, offset  $w / points_{row}$  & for the y coordinate, offset  $h / points_{column}$ , being  $w$  and  $h$  the length in pixels of the plane's width and height,  $points_{row}$  the number of points per row and  $points_{column}$  the number of points per column.

To join all the points, it is needed to iteratively traverse each row of vertices using the *beginShape()* and *endShape()* making a point and the one of the upper row vertices of the shape as it is illustrated in Figure 16 right.

With the plain formed and all the vertices correctly joined, make the elevations will be as easy as modifying the z component of each vertex in the case of the plain terrain, and the magnitude of the vectors in the spherical case.

With this approach surges a problem, as it has been explained in point 4.3.1, Perlin Noise presents a computational cost of  $O(2^n)$ , considering that in order to generate a terrain in which the camera is able to move and render new terrain. Both the Perlin noise and the rendering of the vertices and strips algorithms must be recalculated in each frame. This situation can cause some performance issues. This problem will be explored in the Data structures involved point.

## 5.2.2 Sphere geometries

For the sphere, there are different approaches as was introduced in point 4.5. The first and maybe the most extended way is creating the sphere through the polar sphere equations. These equations bring a method to calculate a sphere formed with an arbitrary number of points. The position of each point is calculated according to 2 angles. They are simple and easy to implement but present an important problem, near the poles of the sphere the point's density increases significantly compared to the equator. This situation can involve different problems like a distortion of continents near the poles and in the equator, as it can be seen in Figure 17, the distortion in the equator is more evident with a low level of detail. Contrary to the density problem which is easier to appreciate with higher numbers of vertices.

$\theta \in [0, \pi]$  (for inclination)

$\varphi \in [0, 2\pi]$  (for azimuth)

$r$  being the radius of the sphere.

$$x = r \cos(\varphi) \sin(\theta)$$

$$y = r \sin(\varphi) \sin(\theta)$$

$$z = r \cos(\theta)$$

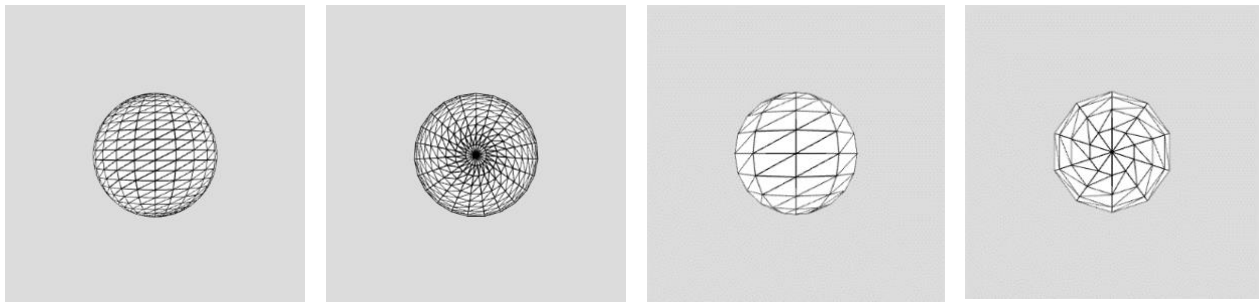


Figure 17 Comparison of the distortion near the equator and the density near the poles.

Although this method presents problems for the rendering of the planet as mentioned before, it is useful to represent simpler structures. For this project this method was used to implement the atmosphere of the planet and clouds.

Another approach is the Fibonacci sphere or Golden ratio sphere. Here modifying the polar equations an equal distribution of points is achieved as it can be seen in Figure 18 (consider that a truly equal distribution of points in a sphere is a very complex problem in mathematics. This method presents an almost equal distribution, more than enough for the scope of this work). Although this method presents obvious benefits as the density of points and also some performance improvements in both computational complexity and memory, it presents the challenge of sorting and arranging the points to form a solid 3D shape. Although this is possible, it adds several layers of complexity to the original problem, making it not ideal but worthy of mention.

$$\Phi = 1 - \sqrt{5}$$

$$\lambda = \Phi \pi$$

$$t = i / n$$

$$a_1 = \arccos(1 - 2t)$$

$$a_2 = \lambda i$$

$i$ , being the index of the point.

$n$ , being the total number of points.

$r$ , being the radius of the sphere.

$$x = r \cos(a_1) \sin(a_2)$$

$$y = r \sin(a_1) \sin(a_2)$$

$$z = r \cos(a_1)$$

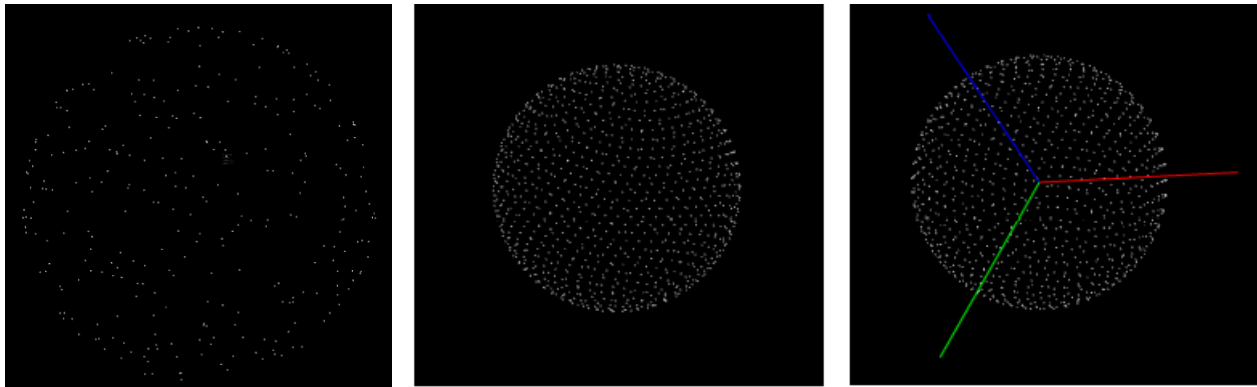


Figure 18 Starfield (left) Fibonacci sphere forming an equal distribution of points (center and right).

Due to the difficulties associated with the process of joining the points, this method was used to create a starfield as can be seen in Figure 18. The position of the points was first initialized with the Fibonacci sphere algorithm for later, displace each point a determined distance using arbitrary movements with coordinates of the points as input. The points are then distributed in a way that resembles more to the distribution of the stars, also, to implement a feeling of depth some of the points are brighter than others simulating different distances.

Finally, the method which will be used to render the spherical surface is the cube sphere. This method presents the idea of defining a set of 3D vectors forming a cube. Forcing those vectors to have all the same magnitude a sphere emerges, as it can be seen in Figure 19. With this method, the density problem is partially solved and the construction of the 3D mesh is easier (in the end it works like 6 plains joined together). There are two evident benefits of this method: first, the sphere is divided into 6 equal parts, therefore the computational cost of the rendering of the planet could be decreased, this is because now is possible to only render the nearest face of the planet to the camera, and second, it will open the possibility to implement a dynamical level of detail (LOD) in each face according to the distance between the terrain and the camera. In summary, this method falls in the middle of performance and accuracy, and at the same time, presents fewer problems and difficulties in the implementation. The process of making the vectors to have all the same magnitude can be easily made in Processing with the function `setMagnitude()`, which is based on the following equations:

$$x' = x \sqrt{\left(1 - \frac{y^2}{2} - \frac{z^2}{2} + \frac{y^2 z^2}{3}\right)} \quad -1 \leq x \leq 1$$

$$y' = y \sqrt{\left(1 - \frac{z^2}{2} - \frac{x^2}{2} + \frac{x^2 z^2}{3}\right)} \quad -1 \leq y \leq 1$$

$$z' = z \sqrt{\left(1 - \frac{x^2}{2} - \frac{y^2}{2} + \frac{y^2 x^2}{3}\right)} \quad -1 \leq z \leq 1$$

Where  $x, y, z$  are the components of the cube vector and  $x' y' z'$  are the new components for the sphere vector. It is worthy of mention to say that the cube is not the only geometrical body that can be used as a base to construct the sphere, other methods provide better results like the ones which use an octahedron or an icosahedron. For the sake of simplicity, the cube is the one chosen for this work.

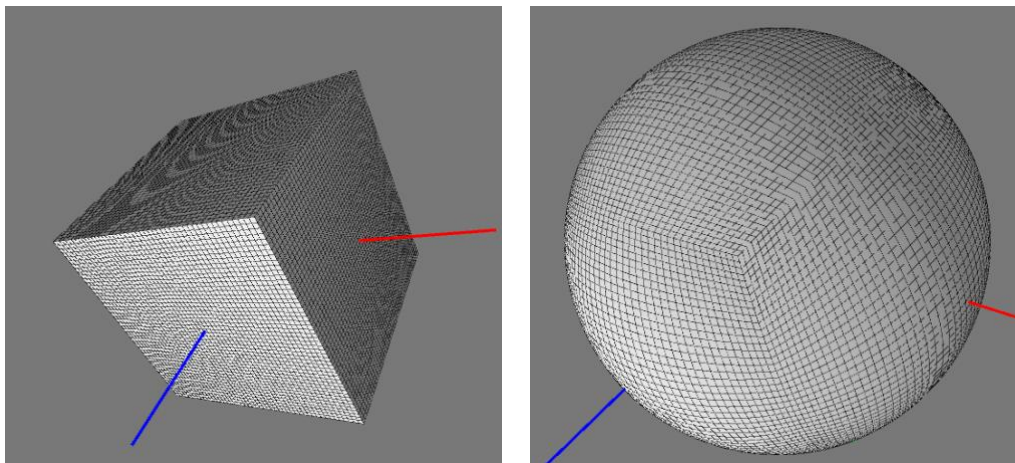


Figure 19 Comparison of a cube before and after setting the magnitudes.

### 5.2.3 Perlin noise implementation

With the plain and spherical terrain covered and knowing the geometry of these surfaces, is important to think about the implementation of the noise function. As mentioned before, for this work it will be used the 1983 version of Perlin Noise. This version is already implemented in Processing with the use of the `noise()` function. In the Processing implementation, the Perlin noise function returns values between 0 and 1, in addition, this implementation suffers from a lack of customizable parameters such as the lacunarity or amplitude, because of that there is a couple of considerations that must be pointed out: First, in the Processing implementation the lacunarity will be always 2 making the frequency of each successive octave twice the previous. The amplitude although can not be modified for each octave can be controlled indirectly through the persistence, which will be specified as the parameter of the function `noiseDetail()` along with the number of octaves. Other factors which influence the terrain such as the scale, coordinates, and the mapping functions will be explained later. Finally in order to solve the values range problem, and make the result of the `noise()` function oscillate between -1 and 1 the result will be mapped with the following equation:

$f(\text{noise}) = \text{noise}(x, y, z) * (-2.0) + 1.0$

Here are the quotes to the Processing references of the `noise()` and `noiseDetail()` functions:

#### `noiseDetail()`

*Adjusts the character and level of detail produced by the Perlin noise function. Similar to harmonics in physics, noise is computed over several octaves. Lower octaves contribute more to the output signal and as such define the overall intensity of the noise, whereas higher octaves create finer-grained details in the noise sequence.*

*By default, noise is computed over 4 octaves with each octave contributing exactly half than its predecessor, starting at 50% strength for the first octave. This falloff amount can be changed by adding an additional function parameter. For example, a falloff factor of 0.75 means each octave will now have 75% impact (25% less) of the previous lower octave. While any number between 0.0 and 1.0 is valid, note that values greater than 0.5 may result in `noise()` returning values greater than 1.0.*

*By changing these parameters, the signal created by the `noise()` function can be adapted to fit very specific needs and characteristic.*

#### `noise()`

*In contrast to the `random()` function, Perlin noise is defined in an infinite n-dimensional space, in which each pair of coordinates corresponds to a fixed semi-random value (fixed only for the lifespan of the program). The resulting value will always be between 0.0 and 1.0. Processing can compute 1D, 2D and 3D noise, depending on the number of coordinates given. The noise value can be animated by moving through the noise space, as demonstrated in the first example above. The 2nd and 3rd dimensions can also be interpreted as time.*

*The actual noise structure is similar to that of an audio signal, in respect to the function's use of frequencies. Similar to the concept of harmonics in physics, Perlin noise is computed over several octaves which are added together for the final result.*

*Another way to adjust the character of the resulting sequence is the scale of the input coordinates. As the function works within an infinite space, the value of the coordinates doesn't matter as such; only the distance between successive coordinates is important (such as when using `noise()` within a loop). As a general rule, the smaller the difference between coordinates, the smoother the resulting noise sequence. Steps of 0.005-0.03 work best for most applications, but this will differ depending on use.*

*There have been debates over the accuracy of the implementation of noise in Processing. For clarification, it's an implementation of "classic Perlin noise" from 1983, and not the newer "simplex noise" method from 2001.*

In summary, for the Perlin Noise implementation it has to be defined the following variables:

- The number of octaves, which will variate between 4 and 8 (for a typical range, but it can oscillate between 1 and  $n$ ).
- Scale, to determine the distance traversed in the noise space for each vertex of the plain or the sphere.
- An offset in the x and y coordinates, will make the camera able to move in the plain terrain while the position of the vertices stays the same.
- The persistence or fallout, to determine the loss of impact each octave has compared to the previous one.
- Also for the plain terrain there will be an initial displacement (respect to the noise space) in order to avoid some possible artifacts in the result.

#### 5.2.4 Plain terrain developed methods

Moving into the plain terrain, the functions developed for this work are:

- *preinit()*, In this function some control variables are initialized.
- *initTerrain(det)*, being *det*, the number of vertices along the width and height of the terrain. This function will prepare the terrain disposing of the vertices of the terrain itself, the water, and the starfield. The terrain and water surfaces will have a total of  $det^2$  number of vertices, and the starfield will have a *nstar* number of stars (points). In addition to this other variables such as the initial displacements, x and y offsets, scale, and persistence are initialized.
- *alterHeights()*, this will be the method responsible of making the terrain elevations. It will be called each frame unless the function *createStaticFrame()* has been called. Inside this *alterHeights* function lays the control of the Perlin noise parameters described before and the mapping functions (Billowy noise, Rigid noise, and my personal mapping function which from now on will be abbreviated to mpi). It has a computational complexity of  $\theta(n)$  being  $n$  the total number of vertices of the terrain.
- *renderField()*, this is the method responsible for rendering the terrain. This method is called each frame after the *alterHeights* function has been invoked. It will render the terrain, coloring it according to a set of methods which will be explained in the shaders and textures chapter. The method may change the rendering results according to some variables like if the water needs to be rendered or not, or if the final texture has to be in colors or pure white. It has a computational complexity of  $\theta(n)$  being  $n$  the total number of vertices of the terrain minus *det*.
- *renderSeaField()*, the analog of the *renderField* function but for the water surface, but with the detail that now the Perlin noise function is being used in a 3d context using the z component as time. With this, the water can move and change.



- *generateStars()*, initializes the star field using the Fibonacci sphere algorithm.
- *renderStars()*, this method renders the starfield surrounding the terrain. For that, each frame renders points of different brightness. It has a computational complexity of  $\theta(n)$  being  $n$  the total number of points.
- *renderLightsAndShadows()*, this method provides the lighting and increases the detail of the terrain by adding the shadows.
- *createStaticFrame()*, similar to *renderField*, but instead of rendering each frame the terrain constructing the triangle strips as *renderField* does, this function makes use of the Processing function *createShape* to construct a static shape of the terrain in a particular moment. For this project was very important to accomplish an infinite world in which the user could move and explore. As the number of vertices increases and so the detail of the terrain the performance decreases drastically, to solve this, was developed this method along with the utility of increase or decrease the level of detail. When an interesting landscape is generated, the user could increase the level of detail as much as desired for then “freeze” the terrain constructing this shape and rendering it with the Processing *shape()* function.

As mentioned before, the *alterHeight* function can modify the overall shape of the terrain according to the mapping function such as the billowy, ridged, and mpi. Is important to consider the overall changes that these modification produces in the terrain in order to understand them and know when they are ideal to use.

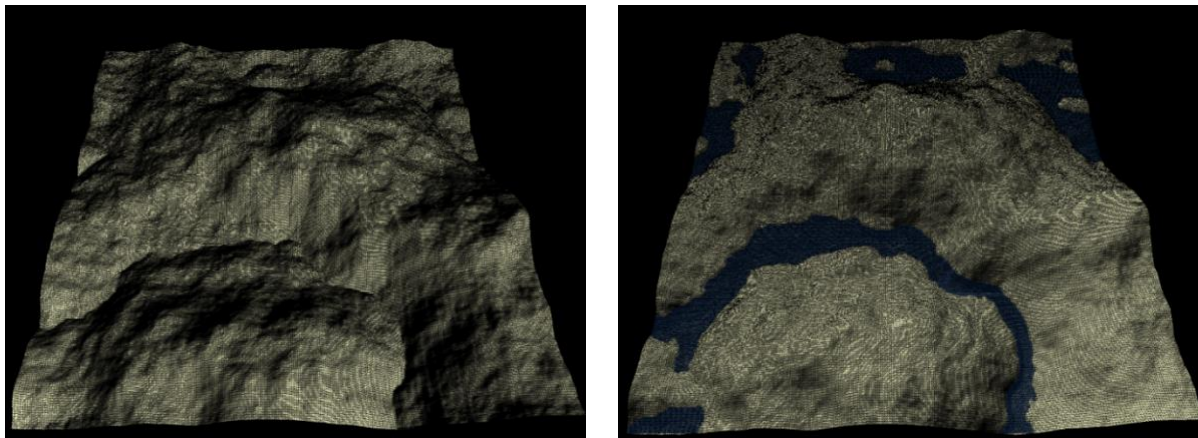


Figure 20 Comparison of the terrain with Billowy noise modification applied with water rendered(right) and without it(left).

As it can be seen in Figure 20, the billowy modification alters the terrain in such a way that the overall shape tends to look less eroded, mountains are less sharp with more continuous changes, and the valleys now fall and are meet in sunken structures. These results can make the surface look a bit strange and may be interpreted as different layers of terrain intersecting with each other (especially as can be seen in Figure 20 left), but, when a water surface is added to the terrain at an arbitrary height (here the water level is calculated with the height of the lowest vertices of the terrain plus a percentage of the maximum height allowed) structures similar to rivers are formed. In the generations chapter other river implementation ideas will be presented and discussed, but, considering that the majority of those ideas brings an extra

computational complexity and the fact that they will be calculated if and only if the terrain is static, means that those ideas, or at least the majority of them couldn't be implemented in this infinite terrain idea. The fact that this modification recreates rivers and lakes kind of structures makes the billowy modification more than interesting. Also, this sunken structure could be interpreted as a representation of oceanic trenches formed by the movement of part of the lithosphere when sinking into a subduction zone because of the collide with other parts of the lithosphere.

One of the benefits of this modification is the fact that these lower vertices which can create rivers and resemble oceanic trenches tend to have noise values of almost 0 (considering that the billowy modification is obtained with the absolute value of the noise function and therefore, it has a range of values between 0 and 1). This produces the situation that when using the billowy noise not as the final collection of noise values, but as a mask, it can be multiplied with other noise modifications using it as a base to form more complex landscapes and to preserve those river structures.

The rigid noise on the other hand is the opposite. It brings those sunken structures up to form more sharp structures. Changes in shape of the mountains are now a bit more pronounced, peaks and mountain ranges emerge from what were the lowers points in the billowy modification as can be seen in Figure 21 left. Valleys can appear, zones where the terrain is flatter. These zones tend to be near the mountains. In a way, this terrain seems to be more eroded. Rivers are still able to spawn but different to the billowy ones if in the billowy modification the rivers were thin, curvier, and in occasions, several ones could join in the luck of affluent, now with the rigid modification, they tend to be more width and less curvy.

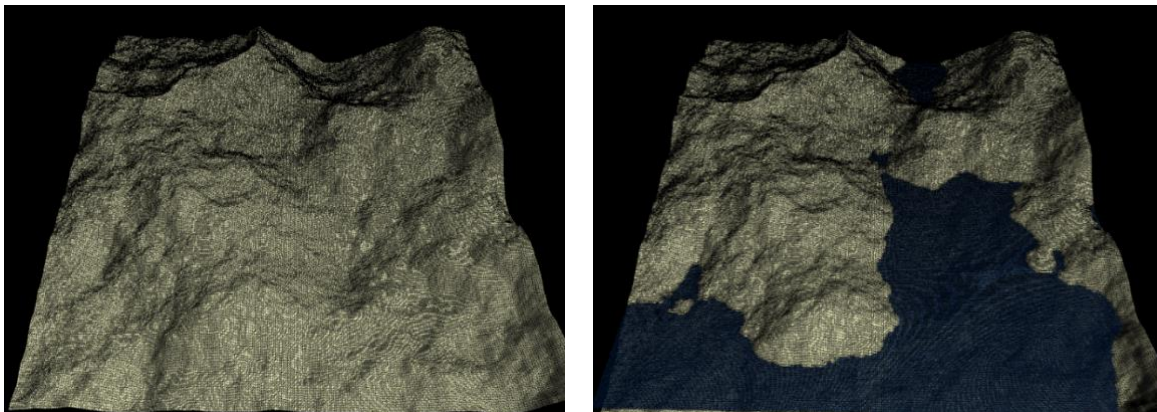


Figure 21 Comparison of the terrain with rigid noise modification applied with water rendered(right) and without it(left).

In addition to the previous, rigid noise take advantage of the valleys, and when the water surface is rendered oceans can appear. Large masses of water in which sometimes could spawn little island or even a system of these to form an archipelago, because of this when rigid noise is implemented or other modifiers which share similar effects coasts, beaches, and river mouths appear as can be seen in Figure 21 right, letting the developers work on shaders and textures which consider sand, very eroded rocks, and other similar components.

Overall rigid noise works just fine for the majority of the applications in which sharp and eroded mountains must appear. It can be used as a mask, like the billowy modification, or even used in a pondered sum of different noise functions to add more roughness to the terrain.

Unfortunately in some cases the level of detail or the imperfections on the terrain could feel missing, getting a terrain that though looks fine, does not look natural and organic enough. In order to solve this, different approaches can be followed: First as mentioned, rigid noise can be combined in several ways with other noise functions or modifications to correct the artifacts and make them more natural. Other alternatives use this kind of modifications as a base to let other algorithms much more complex to work and get a more sophisticated landscape, some examples of these algorithms could be the hydroerosion kind, the downside is that like the rivers implementation algorithms that were introduced before, these hydroerosion algorithms add much more computational complexity and only works for static landscapes. Finally, the *mpi* modification is a mix of all the previous concepts. It was achieved after several tries and numerous modifications. The idea was to get a modification for the Perlin noise values that could be able to create sharp mountains as the rigid noise, with the possibility of having different kind of mountain like the billowy, and valleys near to the mountains which were flat enough to resemble a decent level of erosion. On top of all of this, this modification must provide both rivers and large water masses, and finally being able to be applied on an infinite terrain on the fly just as the billowy and rigid does.

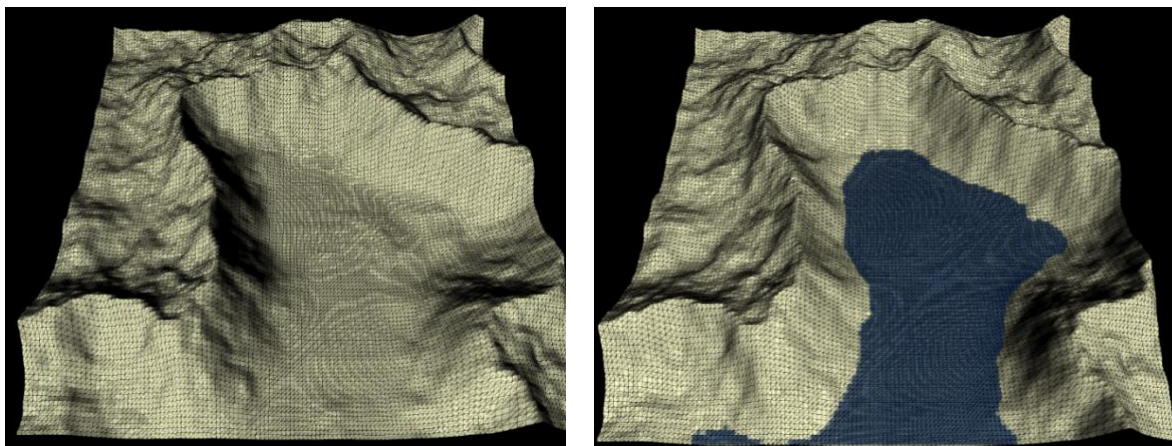


Figure 22 Comparison of the terrain with *mpi* noise modification applied with water rendered(right) and without it(left).

As it can be seen in Figure 22 left, all of the previous requirements were accomplished. This modification not only provides mountain ranges but also produces craters, volcanoes-like structures, and normal mountains. In addition, valleys are flattered, changing the water level they can be appreciated or turned into large masses of water.

To accomplish this result a few steps were required: First, two noise values were needed. One with eight octaves which were called *eightOctavesNoise*, and one with four octaves called *fourOctavesNoise*, both of them used the same scale, persistence, and coordinates in order to obtain coherent values. The *eightOctavesNoise* was converted into an angle just by multiplying its value with two times PI, that angle was then used as an input in the equation shown below.

This equation is indeed the rigid noise one, but with the change that the billowy modification is now the absolute value of the angle's sinus. As can be seen in Figure 14, this function returns values between 0 and 1 which are sharper in the peaks and flatters in the lower points. Finally, the result of this *mpi* modification will be the pondered sum of this function and the *fourOctavesNoise*. The reason behind this is that the *fourOctavesNoise* is being used as a



control with which the final result will have a variety of mountains. Sharper where the angle function values are higher and less eroded where the *fourOctavesNoise* predominates.

$$\theta = \text{eightOctavesNoise} * 2.0 * \pi \qquad 0 \leq \theta \leq 2 \pi$$

$$f(\theta) = 1.0 - | \sin(\theta) | \qquad 0 \leq f(\theta) \leq 1.0$$

$$\text{mpi} = f(\theta) * 0.55 + \text{fourOctavesNoise} * 0.45 \qquad 0 \leq \text{mpi} \leq 1.0$$

To the question is any of these modifications better than the others? The answer is complex. For this particular work, none of these modifications seems better than the others, each one has its pros and cons, perfection is a complicated if not impossible term to achieve when trying to simulate or recreate natural structures. In the end, nature presents different kinds of patterns depending on multiple factors, because of that, the idea of using the three of them depending on different variables like x and y coordinates, biomes, etc. seems to be the most reasonable option. The mix of these different modifications and the gradients needed to make smooth transitions from one to another, among with the conditions with which decide what type of terrain render are let for future updates.

### 5.2.5 Spherical terrain developed methods

Considering that this work is just an implementation of a set of theoretical concepts, the spherical terrain implementation will be simpler and more straightforward. Of course, the complexity of the spherical noise implementation could be much more, but it will be let for future updates. Here is the list of functions developed for the spherical noise implementation for this work:

- *generateCube()*, this method creates and initializes the data structure which contains the 6 faces of the cube, and transforms it into the sphere. Each one is stored in an array similar to the plain terrain. As the cube is indeed the unit cube all the coordinates for each vector will be between -1 and 1. Considering that a cube has 6 faces, each one has  $n^2$  number of points, being  $n$  the width or height of the face ( regular faces ). The total number of vertices or points will be  $6 n^2$ . The computational complexity of this function is  $\theta(n^2)$  as all the faces are filled at the same time.
- *alterCubeMagnitudes()*, similar to *alterHeight* function for the plain terrain. This function applies the Perlin noise algorithm to each vector of the cube sphere. For this implementation, there is a modification to the Perlin noise algorithm similar to the mpi for the plain terrain, but not as detailed and much less complex. The maximum height achievable for any given point has been capped in order to increase the feeling of a planetary scale. This algorithm has a computational complexity of  $\theta(6 n^2)$  being  $n$  the width or height of the faces.
- *render(cube)*, using the position of the camera, this algorithm determines which face of the sphere cube is nearest to it and renders only that face. This is a decision taken after considering the number of points to render, if instead the plain terrain method was applied (rendering the 6 3D meshes correspondents to each face) the performance will be more than insufficient. Another change in respect of the plain

terrain *renderField* algorithm was to use the QUAD\_STRIP parameter for the Processing *beginShape* function, this is due to the fact that using the TRIANGULAR\_STRIP increased the number of artifact in the face's unions. The coloring technique is simpler compared to the plain terrain one. This algorithm has a computational complexity of  $\theta(6 n^2)$  being  $n$  the width or height of the faces.

- *createStaticRender(cube)*, similar to the plain terrain function *createStaticFrame*, this method is able to create a solid shape from the sphere cube for later rendering it using the Processing *shape* function. Taking into account that for the spherical case, there is no need for an infinite terrain, this method increases the overall performance of the application considerably. The downside is that it will not be suitable for a dynamic level of detail. This algorithm has the same computational complexity as the *render* functions but it is called just once instead of each frame.
- *generateStars()*, initializes the star field using the Fibonacci sphere algorithm.
- *renderStars()*, this method renders the starfield surrounding the cube sphere. For that, each frame renders points of different brightness. It has a computational complexity of  $\theta(n)$  being  $n$  the total number of points.
- *renderAtmosphere()*, in order to increase the detail of the planet, these methods render its atmosphere as can be seen in Figure 24, this is composed of 4 different shapes. 3 of these shapes are semitransparent sphere with different colors, simulating the refraction that the atmosphere particles does when interacting with different light wavelength. The last shape corresponds to another sphere with a cloud texture applied with a rotation to simulate a cloud movement around the planet. It is also possible to recreate clouds in a more detailed and complex way using the complex clouds system which will be explained later.

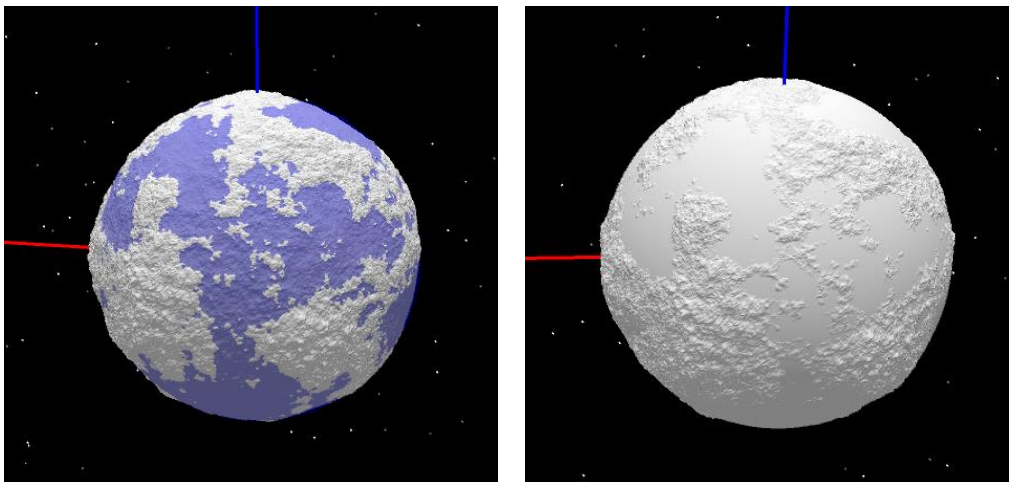


Figure 23 Spherical terrain with oceanic bottom and a water layer(left) and without oceanic bottom and smooth ocean(right).

In summary, the spherical implementation was a bit trickier compared to the plain one. As pointed out before there have been a couple of changes from the plain terrain, for instance: The water could be rendered now in two ways, in the first is rendered as a smooth sphere independent to the planet, with blue color and without using the 3D Perlin noise, compared to the plain terrain is less convincing as water but more efficient. This way, an oceanic bottom

exists and it can be explored. With the second way the water surface is now not an independent blue sphere but the planet terrain itself, for all the vectors which form the planet, if any has a magnitude less than arbitrary quantity (the sea level), then the magnitude of that vector will be forced to be the radius of the planet. With this the ocean is completely smooth, the oceanic bottom disappears and the performance increases even more. Another change is that the Perlin noise modification is simpler and thought to achieve a more planetary scale look, making the terrain look separated in continents. Finally, the textures and shaders although are still procedural are simpler compared to the plain terrain.

Even with those changes, the final result seems pleasant enough to satisfy the needs of this work. The results displayed in Figure 23 shows how an Earth-like planet is obtained with the parameters used, changes in these parameters along with other Perlin noise modifications could lead to the possibility of generating new planets with different biomes and more exotic looks.

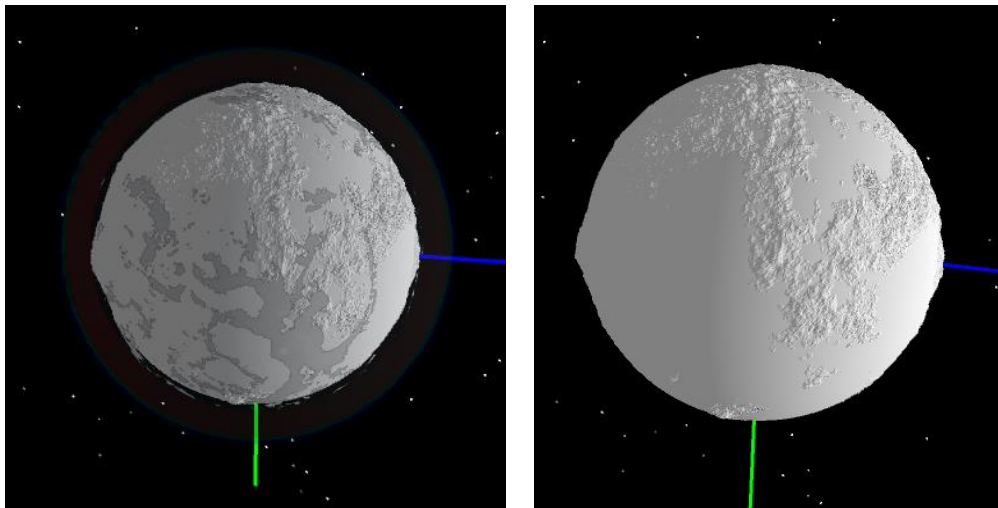


Figure 24 Comparison of the spherical terrain with the atmosphere rendered (left) and without it (right).

Before moving into the data structures and the technical decisions taken for this work, the last implementations did ( in terms of models and noise implementations ) was: First, the complex clouds system. This system is although not perfect, much better than the simple clouds implemented in the *renderAtmosphere* function. Instead of using a simple png, with this method, the clouds are rendered procedurally in real-time, suffering from different deformations as time passes, and being stacked once upon each other simulating the different layers of clouds that are present on the earth. The more clouds are stacked the more opaque's they turned blocking the light as they would do on a planet. The downside of this method is the performance, this method is far from ended and as a direct consequence is not optimized yet. A comparison between the two methods can be seen in Figure 25. The second is the atmosphere rendering, as introduced in the spherical terrain functions list. This method simulates an atmosphere with a set of semitransparent spheres with combinations of red, green, and blue colors simulating the different light wavelengths which refract on a real atmosphere. Both the clouds and the atmosphere are far from accurate, the ideal way of doing this would be through particle systems interacting with simulated light rays with techniques like raycasting or raymarching, these ideas would increment the quality and detail of the final result would be let for future updates.

Notice that the collection of algorithms that are able to render these more sophisticated clouds,

use 3D Perlin noise with the third component as the time, eight octaves, and the typical persistence and lacunarity, in a similar way as how the plain terrain water works, the most tricky and complicated part was how to use the texture generated in a 2D context to wrap a 3D invisible sphere avoiding artifacts in the final image. The key was to try to make the texture symmetrical to avoid the artifacts and then mix different layers of clouds to increase the randomness and create a more complex-looking system. It was not only interesting but very instructive to see how simple algorithms can represent nature or at least, provide similitudes with it.

The list of algorithms to implement this cloud system is:

- *generateTexture(offset)*, creates a 2D texture to apply over a surface. Using *getColorOnPxl* to decide what color is each pixel and Processing functions such as *createImage*, *loadPixel*, *updatePixels* and *color()*.
- *getColorOnPxl(i)*, this algorithm determines the opacity and density of a cloud on a given point based on an index *i*. It uses Perlin noise in 3D, with 8 octaves.

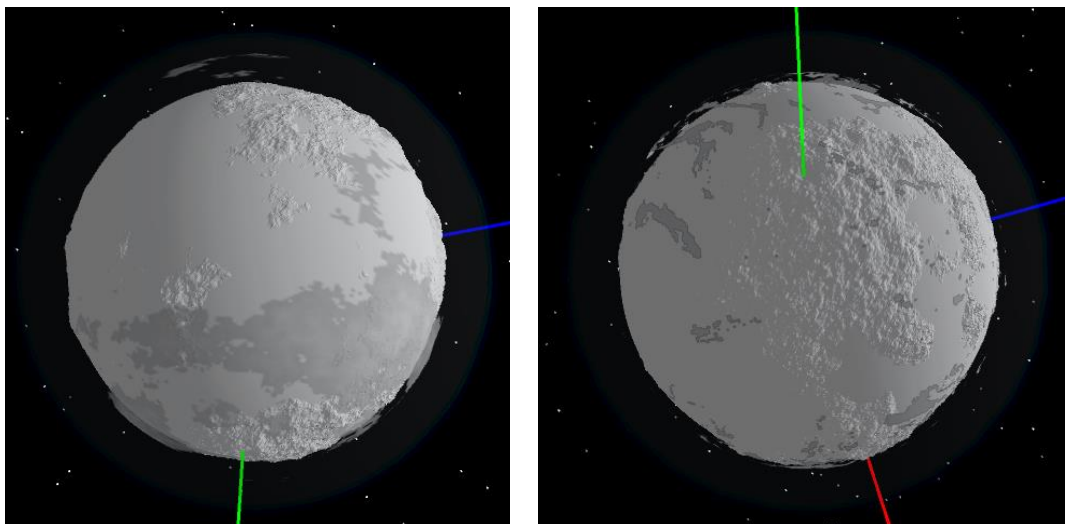


Figure 25 Comparison of the spherical terrain with the complex clouds(left) and simples (right).

### 5.3 Data structures involved

By definition, a data structure is a data organization, management, and storage format that enables efficient access and modification. In addition, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data. In this work, there are four data structures defined: The plain terrain, the spherical one, the clouds, and the heightfields. About the plain terrain there are a couple of structures to defined and to comment on, for instance:

- One-dimensional arrays, these structures store vectors which are used to represent points in a 3D space, the terrain field, sea field, and the star system are stored in separated arrays. As it has been mentioned before, given a width and height of the terrain field defined by a level of detail the length of the terrain and sea field would be

equal to  $detail^2$ . In order to represent and access those points, there has to be a logic to treat the one-dimensional array as if it was a matrix or two-dimensional array, this is because although the one dimensional kind of array allows a more efficient way to store data values, at the moment of rendering the field, joining the points, etc. the two-dimensional arrays are easier to use. To accomplish this and still use the one-dimensional arrays the logic to use is as easy as, to access points to the right or left of a given point just add or subtract one to the index, and to access points above or below add or subtract detail to the index (always considering the boundaries of the structure).

- PShapes, as it has been commented, PShape is the Processing data structure used to create, store and render user custom shapes among the primitive ones. These shapes are incredibly important to increase the performance of the program.

The spherical case takes advantage of the concepts used in the plain terrain and uses it, developing and implementing this spherical terrain turns evident that there is space for improvement in terms of optimization for both memory consumption and computational efficiency, again the scope of this works is a more practical study of some of the theoretical concepts explained in the theoretical part, but this could be an interesting factor to fathom and investigate about in future improvements. Some of the data structures used for the spherical terrain are:

- A one-dimensional arrays, similar to the plain terrain, but taking into account that the sphere which is used to render the planet is formed from a cube and therefore there will be six one dimensional arrays, one for each face of the cube. Also in the spherical case, the one-dimensional arrays are used to store the color palettes which are used to give color to the terrain, in this case, the arrays store not vectors but hexadecimal values.
- HashMap, a HashMap is a particular data structure which stores object based on a key-value relation, where the items stored are not accessed by an index as it could be in a typical array but with a unique key. For instance, in this case, each face of the cube is stored with a string key which stores the axis associated with the face (the most to the right face will be identified with the key "positive\_x" the most to the left with "negative\_x" etc.). In order to store these six faces of the cube, several approaches could have been followed. It could be another array, making the global structure a  $6 * detail^2$  matrix, or a linked list kind of structure like an ArrayList, etc. In this case, for reasons related to the one-face rendering, the HashMap turned to be the more convenient and comfortable data structure to use. This HashMap system is associated with a one-dimensional array of vectors called *facesMainsVectors*, this array will store six vectors (<0,0,1>, <0,0,-1>, <0,1,0>, etc.) it is used to check which of those vectors the camera position is nearest, knowing this the algorithm is able to identify which face of the cube needs to be rendered.



The clouds system is stored in a new data structure which is nothing else than an object-oriented class. This is due to the ease of organizing the variables and methods needed to render and store the clouds separated from the spherical code. The inner structures of the clouds class are:

- PShapes, same as the plain terrain, in this case, the shape is generated with the processing primitive SPHERE, which is based on the polar sphere.
- PImage, PImage is the Processing data structure to store and render images, in this case, the image is created from a heightfield generated with Perlin noise and it is applied as a texture to the sphere shape.

The most evident performance issue of this work is due to the lack of GPU acceleration for part of Processing. This causes that everything has to be calculated and rendered with the use of the CPU, because of this things like rendering a 3D mesh of a million points each frame (without the use of *createStaticRender* or *createStaticField* can be unthinkable (for most of the Figures shown in this work the terrain is formed with 409600 points in the case of the plain and 540000 for the spherical).

Another problem is the noise performance, as mentioned Perlin noise is an expensive algorithm in terms of computational complexity. For the plain terrain, the Perlin noise algorithm is calculated several times with changes in the number of octaves, persistence, etc. ( It is used for the terrain itself, for the shader, and the sea field ), and in the case of the spherical case, although the planet could always be rendered as a static render, the complex clouds system uses a 3D Perlin noise to create an image and apply it as a texture to a shape each frame. Solving these performance issues is not an easy task, and several approaches could be followed:

- Decrease calculations, the easiest way of doing fewer calculations could be to decrease the number of times that the Perlin Noise function is called or decrease the number of points in which the noise function is applied. Although easy this solution will come with a quality downgrade and will not be a technical solution but luck of fix.
- Parallel processing, due to the fact that the sea field of the complex clouds is independent of the plain or spherical terrain an optimization process could be to let those algorithms run in a parallel way in respect to the terrains. Also, the terrain heights calculations could be parallelized as well, but several considerations will be needed and could be not as easy as the previous ideas.
- Data structure optimization, it has already mentioned that several approaches could be used in the way that the values, data, and information is stored and used in the terrains. Other approaches like changing the HashMap for arrays or matrices could increase the overall performance.
- Using less the Perlin noise function, as mentioned Perlin noise is expensive in terms of computations, things like the sea field could be calculated using other algorithms like a Gaussian distribution for water reflections, this could make the overall quality or level of detail be maintained and at the same time save computational complexity.

## 5.4 Shaders and textures

### 5.4.1. Introduction

Processing offers a set of tools to work in the creation and implementation of shaders and textures in 2D ( using the 2D renderer, P2D) and for 3D (using the 3D renderer, P3D ). Although these tools are not as powerful or flexible as others engines could bring (like unity or unreal engine), they are powerful enough for the scope of this work. One thing that is need for consideration is that Processing renders shaders and textures using the CPU, because of this sometimes when the shader needs a lot of calculations ( like the complex clouds ) several performance problems appear. Once again this kind of problems are not considered a very important problem taking into account that this work is just a couple of theoretical concepts put into practice to visualize them. Nonetheless is important to remark that moving the code to other engines like the ones mentioned before could bring both, performance and quality upgrades.

### 5.4.2 Plain terrain shader

For the plain terrain, a more complex and sophisticated shader has been developed, looking for details in the surfaces, vegetation, snow on the top of the mountains, and sand near the water surfaces. For the spherical terrain, the shader is much simpler, it will not be especially difficult to implement a shader similar to the plain terrain but, on one hand this simpler shader doesn't present unpleasant results and it was found ideal for a planetary scale, and in the other hand using both shaders brings the possibility of comparing them, looking for pros and cons, and allows to explain the difficulties or ease on both of them.

The plain terrain shader was as mentioned before designed looking for details and a more organic view. As it has been said, imitate nature is nothing but a very complex task, and because of that usually, simplifications or assumptions are made, this case is not the exception. Here the main concepts are the followings: When a surface tends to have a lot of inclination is because that surface is more eroded and therefore the amount of vegetation that can grow there is less, contrary when a surface is flat enough, much more vegetation is able to grow and extend. The taller is a given point compared to the rest, the more possibilities of snow forming are. Lastly, the sand is formed near the water surfaces because of the erosion that the water produces on the rocks.

With these ideas established the first task was to determine the amount of inclination of a given point in the terrain with respect to the others. Taking into account that the geometry of the terrain allows to given a point (vertices) found its neighbors (the vertices that are surrounding it) instantly, it is possible to calculate an average inclination of those vertices. Then if the difference between the point being evaluated and the average is less or more than a constant ( in this case it was called the flatness constant) is possible to determine if for that point it has to be painted with vegetation tonalities or with rocky ones. To make better gradients and have a smooth transition between vegetation and rocky terrain, the colors are blend together using the value of the difference between the point height and the neighbors average to determine the percentages used in this blend operation. The last consideration was to alter the percentages used in the blend operation according to the height of the point. This is because the higher one

goes in real life, the less vegetation will be found.

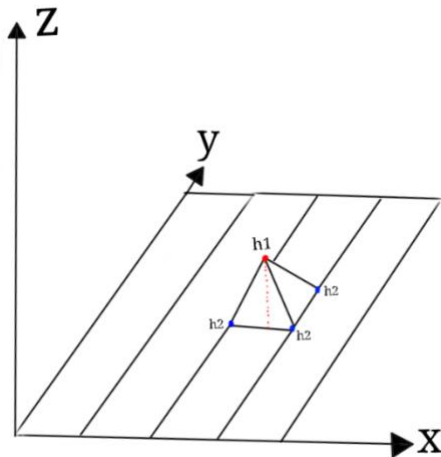


Figure 26 For a given point (red) its height is calculated ( $h1$ ) and compared to its neighbors (blue) points (in this case the neighbors have all the same height ( $h2$ ), the average then will be calculated as  $h1 + h2*3 / 4$ . Finally the difference  $h1 - (h1 + h2*3 / 4)$  will be compared to the *Flatness constant* .

For the others attributes of the terrain ( sand, snow, etc.) similar procedures were used. Blending color between rocks and sand, sand and vegetation, rocks and snow, vegetation and snow, and so on. One important thing to remark is that all of this blending is done as mentioned using a difference between the height of a given point and its neighbors. Happens that when increasing the number of vertices ( without changing the measurements of the terrain ) these vertices are more grouped, and therefore the difference between the height of a given point and its neighbors decreased, for that reason the flatness constant has to change dynamically with the number of vertices, for that the following expression was used:

$$Flatness_{dividend} = level\ of\ detail / 128.0$$

$$Flatness_{constant} = 1.6 / Flatness_{dividend}$$

As it was mentioned in previous points, the detail of the plain terrain can increase or decrease according to the user needs, but it does in jumps of 128 vertices, being the default 256. With the inclination and blending task solved, the next step was to determine how the sand and snow will be formed. As both sand and snow are things that could be described with the position, the approach followed was to first declare an arbitrary height for both of them to spawn and then alter that quantity with the Perlin noise function to increase the randomness and give a more organic view (this noise was calculated with a bigger scale, typical persistence and two octaves). The height for the sand to appear was calculated using a percentage of the lower point of the terrain plus the water level, to make the sand appear very close to the water. For the snow, it was determined that any point above a percentage of the maximum height achieved in the terrain will be a possible spawn for it. Due to the fact that the terrain tends to be flatter near the water and sharper near the top of the mountains, the Flatness constant needs to be adjusted according to it, letting the expression like:

$$Flatness_{constant\ sand} = 0.8 / Flatness_{dividend}$$

$$Flatness_{constant\ snow} = 1.2 / Flatness_{dividend}$$

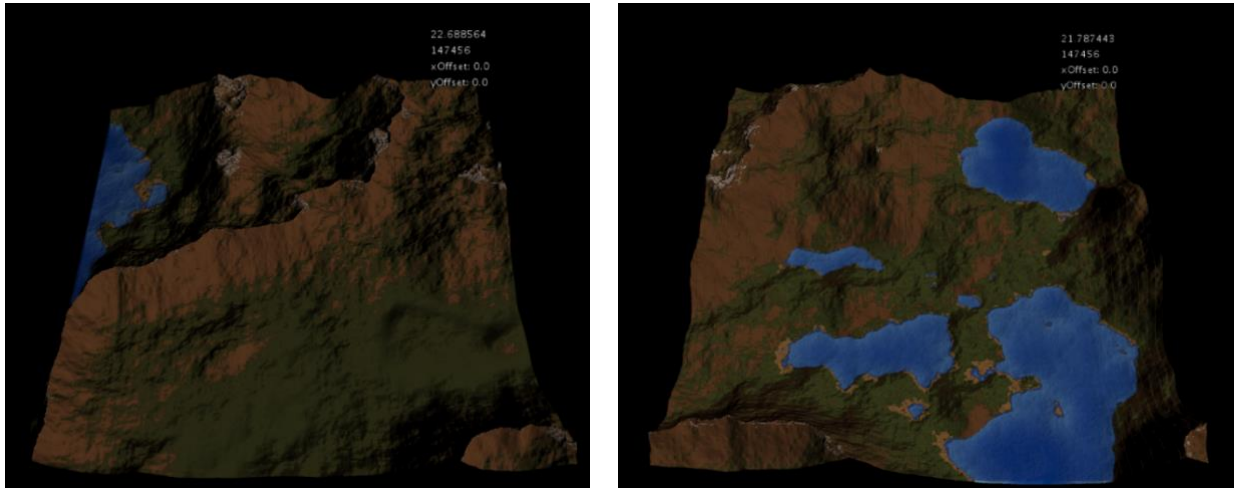


Figure 27 Different captures of the plain terrain surface with the shader applied. In both the terrain is formed with 409600 vertices (640x640) and altered with the *mpi* modification.

As it can be seen in Figure 27 left, the amount and distribution of the sand and snow are unpredictable and natural-looking. At the same time in Figure 27 right is easier to appreciate how depending on the inclination and height there is more or less vegetation.

To conclude the plain terrain shader there are two more things to present: The light system, and the water system. Although the complexity of the main shader or the details achieved on the terrain thanks to the Perlin noise function and the modifications, without lights and shadows these details will not be appreciated. These effects are the ones that bring depth and texture to the terrain. To being able of rendering those lights and shadows the Processing functions *ambientLight()* and *directionalLight()* were used. First, it was necessary to capture the mouse movements in order to map the position of the light source, then use those movements to recreate a 3D position vector (where the z component was fixed to a determined height) allows the lights to have different angles using the *directionalLight()*. Finally, *ambienLight()* was used to have a generic ambient light. In this light system exists 3 different kinds of light: Day, evening, and night mode as it can be seen in Figure 28. Each one modifies in a certain way the color of the light rays produced with *directionalLight()*.

The water system is simple, independent of the plain terrain, exists a sea field that covers all the water masses large or smalls. This sea field is created with very similar properties as the plain terrain does, it is formed with the same number of vertices ( to increase the detail of the sea at the same time that the terrain does). It is joined using the Processing *beginShape()* and *endShape()* functions with the TRIANGULAR\_STRIP attribute. The Perlin noise function used on this water surface has a big scale ( to increase the number of irregularities that the surface will have), using three octaves with a typical persistence and it is a 3D Perlin noise instead of a 2D as the plain terrain one was. Using the 3D Perlin noise, the water receives a third component treated as time, this allows the water to change independently similar to the complex clouds system does. It receives two new parameter called *xOffsetSea* and *yOffsetSea*. With these two new parameters, the user is able to establish the direction of the wind, reproducing the irregularities caused by an airstream into the water surface. This kind of water simulations are expensive in computational complexity terms, other options could be a Gaussian distribution for a refractive water surface.

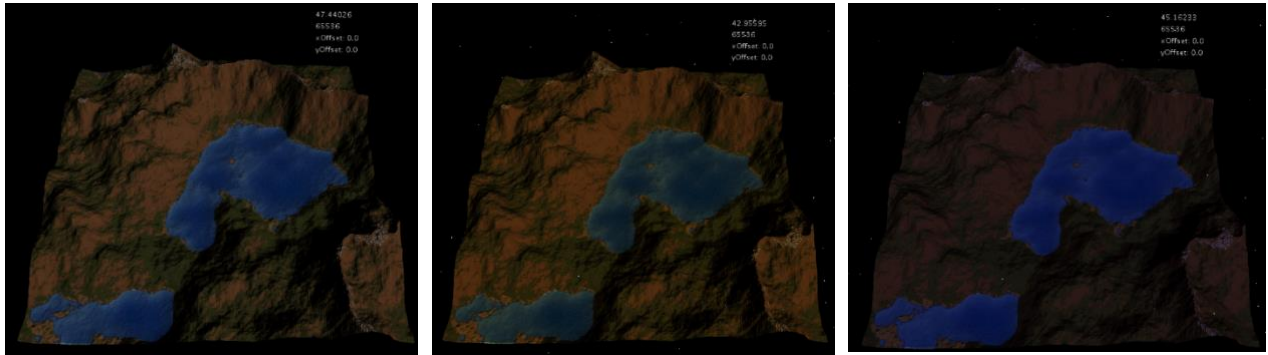


Figure 28 Different captures of the plain terrain surface with the three day modes, from left to right day, evening, night.

### 5.4.3 Spherical terrain shader

The spherical terrain shader is divided into two sections: terrain shader, and atmosphere shader. As it has been said earlier the main spherical terrain shader (terrain shader). It was designed to be simpler and much less sophisticated in order to compare it with the plain terrain one. This terrain shader consists of two colors palettes: Earth palette, which will be applied for all the terrain above the sea level ( it must be remembered that the sea level value is an arbitrary constant determined with the maximum height achieved ), and the sea palette which is applied to the terrain below the sea level (consider that if the planet is being rendered with the option of making the water surface smooth this sea palette will be not applied and instead a plain blue color will be used to paint all the sea surface).

Both palettes are stored in separated arrays, with a set of colors related to each other by a gradient. For instance, the earth palette has the first color a light yellow to represent the sand, then it turns into green for the hills, after brown for the mountains, and finally white for the snow. In order to apply them when a given point is evaluated, its height (which is calculated with the vector magnitude minus the sphere radius) is compared to the maximum and minimum height achieved in the planet, knowing this two values, the shader determines which position (color) of the palette must be applied to that point. This procedure is the same for the sea palette.

This way of determining a color for a given point is much less expensive in computational complexity compared to the plain terrain one, however, two new data structures are needed to store those colors increasing the memory complexity (although this increment is in terms of bytes). Lastly one of the options is to render the water in a similar way as the plain terrain does, without the sea surface being smooth the oceanic bottom is appreciable and the water surface is replaced with a semi-transparent sphere with a radius equal to the planet radius plus the sea level. This way of implementing a sea surface is rather simple, and much more efficient compared to calculating a 3D Perlin noise each frame for a whole new set of points as happens in the plain terrain, however, the complexity and detail are not comparable.

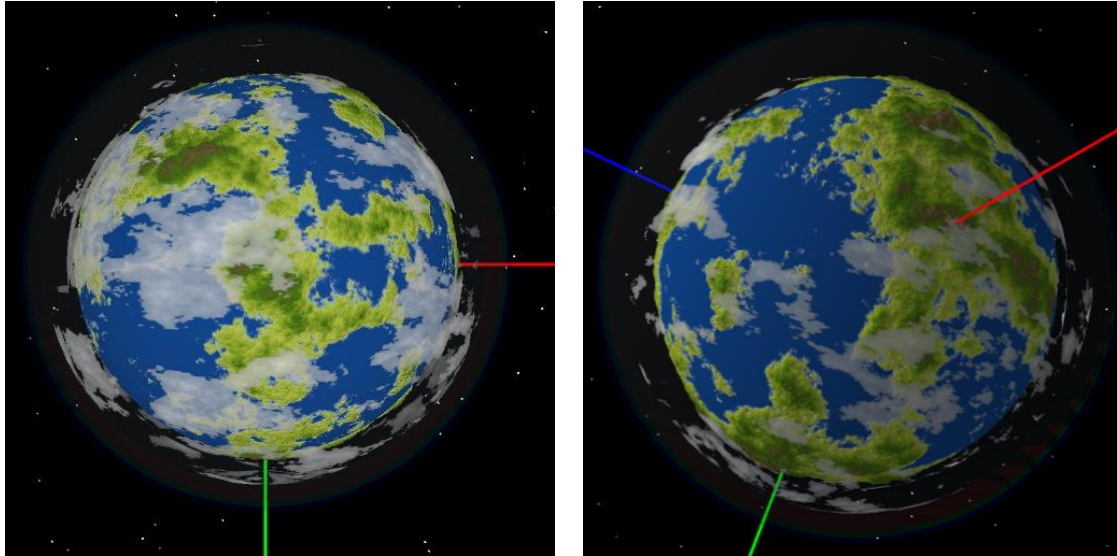


Figure 29 Spherical terrain with shader applied complex clouds and smooth sea.

For the atmosphere shader, a couple of assumptions were made: First, it was acceptable to not simulate the refractions of light rays through a dense particle system, and second, it was admissible for the clouds to don't have depth but to be 2D surfaces. These decisions were made in order to decrease the challenging difficulties associated with those ideas. Independent of the difficulties associated, it must be considered that Processing does not use a GPU acceleration as other IDE's (for instance Unity) could do, this fact has also influenced the simplification decisions.

Then as it has been introduced in previous points, the atmosphere system is composed of the atmosphere and the clouds. For the atmosphere itself, three consecutive semi-transparent spheres are stacked, each one with the colors red, green, and blue so they can simulate how different light wavelengths interact with it. Then the clouds can be simple or complex. The simple is just a png texture generated with climatological data applied over a sphere that surrounds the planet. The complex is made with an image that is created for each frame with a heightmap calculated using 3D Perlin noise with the third component as time. Then the pixels of the heightmap with values less than a threshold are set transparent, this allows the clouds to appear and disappear gradually, to change their shape and simulate clouds with different zones with more or less density.

At the comparison moment, it results evident that the plain terrain shader looks better, more organic, sophisticated, and complex, but also more expensive to render and calculate. It also presents different implementation difficulties which are avoided in the spherical case, as it was said in the spherical geometry comparison there is not a better neither perfect method, but instead each one is suitable for different scenarios. For instance, the spherical shader could be used in space exploring games as the first shader of the planet when it appears in the vision range of the player, for later when the user camera gets closer to the planet, the spherical shader can be replaced with a more sophisticated one, saving computational power.



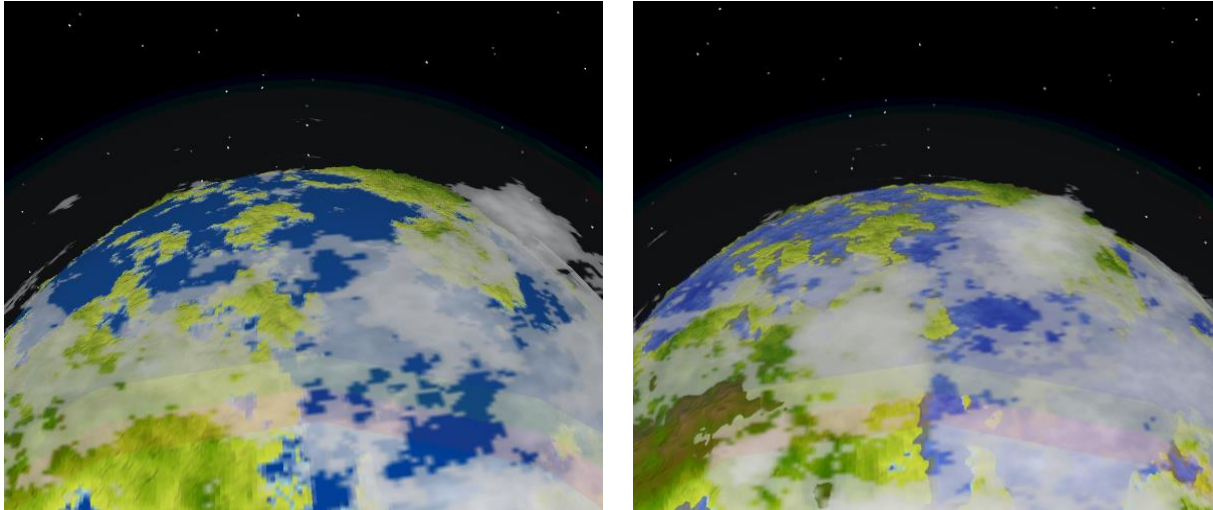


Figure 30 Spherical terrain, shader applied and complex clouds, with smooth sea (left) and with oceanic bottom(right).

## 5.5 Generations

### 5.5.1 Hydroerosion

Hydroerosion is the process where the terrain is eroded by the effect of water movements such as rivers, cascades, waves, rain, etc. The majority of hydroerosion systems simulate in a very decent level of detail these natural effects over the terrain. It usually comes with two major problems: First, it must be considered that this hydroerosion needs to be calculated in a non-real-time system, in other words, it can not be applied in this work. The reason for this is because all the hydroerosion systems are performed in an iterative way, where, given a 3D mesh (the terrain) a set of thousands of droplets fall into it. As the rain would normally do in real life, these simulated droplets drag a certain amount of sediment from the terrain, falling from one point to a lower one, depositing on their way certain sediment quantity's (and increasing the terrain elevation in that point) until they evaporate or a minimum local point is reached. This process is repeated in an iterative way up to tens of thousands of steps. From this short explanation, the second problem could be easily deduced: Computational complexity. These kind of algorithms are incredibly slow and very expensive in terms of power, although the results could be astonishing as it can be seen in Figure 31, the *mpi* noise modification provide somehow similar results (taking into account that the hydroerosion algorithms present much more quality to the final result and the hydroerosion obtained from the *mpi* modification is just luck of erosion) where the slope of the mountains resemble the kind of patterns generated by the water erosion and the valleys tend to be flat. Considering that the *mpi* is applicable to an infinite terrain and is calculated in a real-time system, it has been discarded to implement a proper hydroerosion algorithm for this work. As the plain and spherical terrain systems provide through the *createStaticField* and *createStaticRender* functions a 3D mesh in form of *PShape* objects, it could be a very interesting feature to be added in the future.

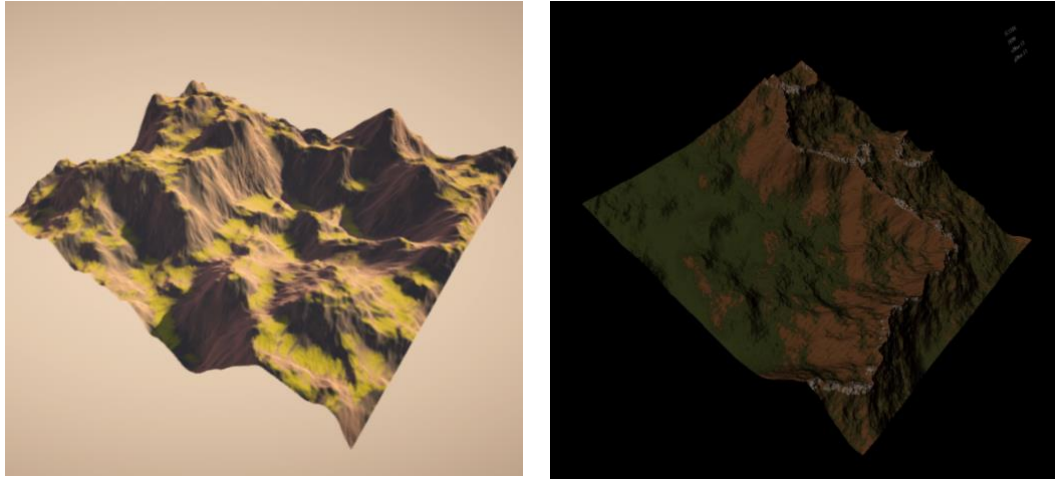


Figure 31 Hydroerosion system (left) by Sebastian Lague (Lague. 2021), and mpi modification (right).

## 5.5.2 Vegetation

One of the most common ways of implementing vegetation is through a shader in a similar way as to how the plain terrain shader does. Evaluating the height of a given point in respect to its neighbors, but in addition, is very common to see some vegetation sprites placed on the zones where the vegetation could spread. These sprites can be divided into two types: The first one are the prerendered ones, a collection of images or 3D models which are placed in determined zones according to some rules. The other are the procedurals or real-time rendered. At first, these were the type of vegetation which were going to be implemented in this work. In order to implement this vegetation, the approach was oriented to an L-System (Lindenmayer system). With this kind of system, different types of plant-like structures could be achieved through a recursive algorithm, which substitutes a text string following a set of rules and axioms. The result is typically a 2D structure as can be seen in Figure 32 (although some L-Systems are able to work in 3D) which then can be used to create a 3D structure through a convolution.

After several reconsiderations, this vegetation system based on procedural or real-time rendered vegetation has been discarded, the reason is that although this structure can have different results when a noise function is applied to the main algorithm and therefore could be another good example of the uses of the Perlin noise function, it adds more computational complexity to the project without contributing too much to it.

For future updates and improvements, this vegetation system could be explored in more detail or even implemented looking for the 3D L-Systems. For now, the shader results present a decent level of detail in the general result of this work which provides a pleasant render.





Figure 32 Examples of L-System structures made for this work.

### 5.5.3 Rivers and lakes

Typically the river generation systems have been divided into cheap and expensive computational methods. The cheap ones don't make a high use of the CPU or GPU and are able to achieve river-like structure through methods like the one presented with the billowy noise. As it has been mentioned and commented on before, these methods use one or several layers of billowy/rigid noise as a mask to alter the terrain making the river channels. They are quick, efficient (in comparison to the expensive group) and provides a decent result. The expensive computational methods on the other hand are, as its name suggests, a set of algorithms with high computational complexity and therefore provides much more rich and detailed results.

In July 2013 SIGGRAPH article 143. "Terrain generation using procedural models based on hydrology" presented by Jean-David G enevaux, Eric Galin, Eric Gu erin, Adrien Peytavie, and Bedrich Benes, it was presented a new way to recreate rivers based on a hyper-graph, which grows in a similar way as the L-Systems does based on a grammar system. Other expensive methods like the ones based on hydrology or hydroerosion systems are also capable of achieving astonishing results.

As it has been common in this work, the majority of these expensive methods that provide richer details are though for non-real-time terrain simulators, heightmap editors, etc. therefore, they are not suitable for an infinite terrain implementation. However, they don't stop being interesting and a future implementation which could improve the overall quality of the work, being implemented similarly as the hydroerosion update, using the `createStaticRender` or `createStaticField` methods to work with a 3D mesh object (the terrain itself).

Considering all said previously about rivers, for this work was considered important the possibility of render rivers dynamically, based on noise functions and modifications such as the billowy noise. The problem with the billowy modification is that it was too simple, and it doesn't present the complexity level desired for this work rivers.

To accomplish a decent amount of detail in the rivers, without using static techniques like the hydroerosion systems, the solution was as follows:

- First, for each point in the terrain, use a similar mpi noise modification mixing four and eight octaves noise (same scale and persistence as the terrain for movement coherence).
- Both the four and eight octaves noise has to be calculated as the complement of the main terrain equations. This way the rivers will pass near to the mountains falling to the valleys.
- With the height map obtained, a white mask was applied. A white mask is a technique used in the noise context to create a threshold. This threshold was used to determine which values of the heightmap use and which others discard. An example of a white mask can be seen in Figure 33.
- Finally, in the getColor function, the value returned from this process will be evaluated deciding if a given point must be painted as a river component or no. The higher the point the more “frozen” will be the river.
- The given point will not have its height modified, it will only be painted with a shade of blue.

This way the terrain can show decent quality rivers as can be seen in Figure 34. Obviously, hydroerosion simulations, or any kind of simulation where particles play a role, and a sediment deposition logic is applied will show much better results. But this shows another of the Perlin noise uses in the procedural textures generation. The rivers equations are:

*eightOctavesNoise = PerlinNoise(x, y); Where noise parameters are 8 octaves 0.45 persistence.*  
*fourOctavesNoise = PerlinNoise(x, y); Where noise parameters are 4 octaves 0.45 persistence.*

$Mask_1 = | \sin ( 2 * \pi * ( eightOctavesNoise * -2.0 + 1.0 ) ) |$   
 $Mask_2 = Mask_1 * 0.35 + fourOctavesNoise * 0.65$

*Paint as river if  $Mask_1 < 0.25$  or  $Mask_2 < 0.2$*



Figure 33. example of white mask. Only the noise value above a determined threshold are shown in white.

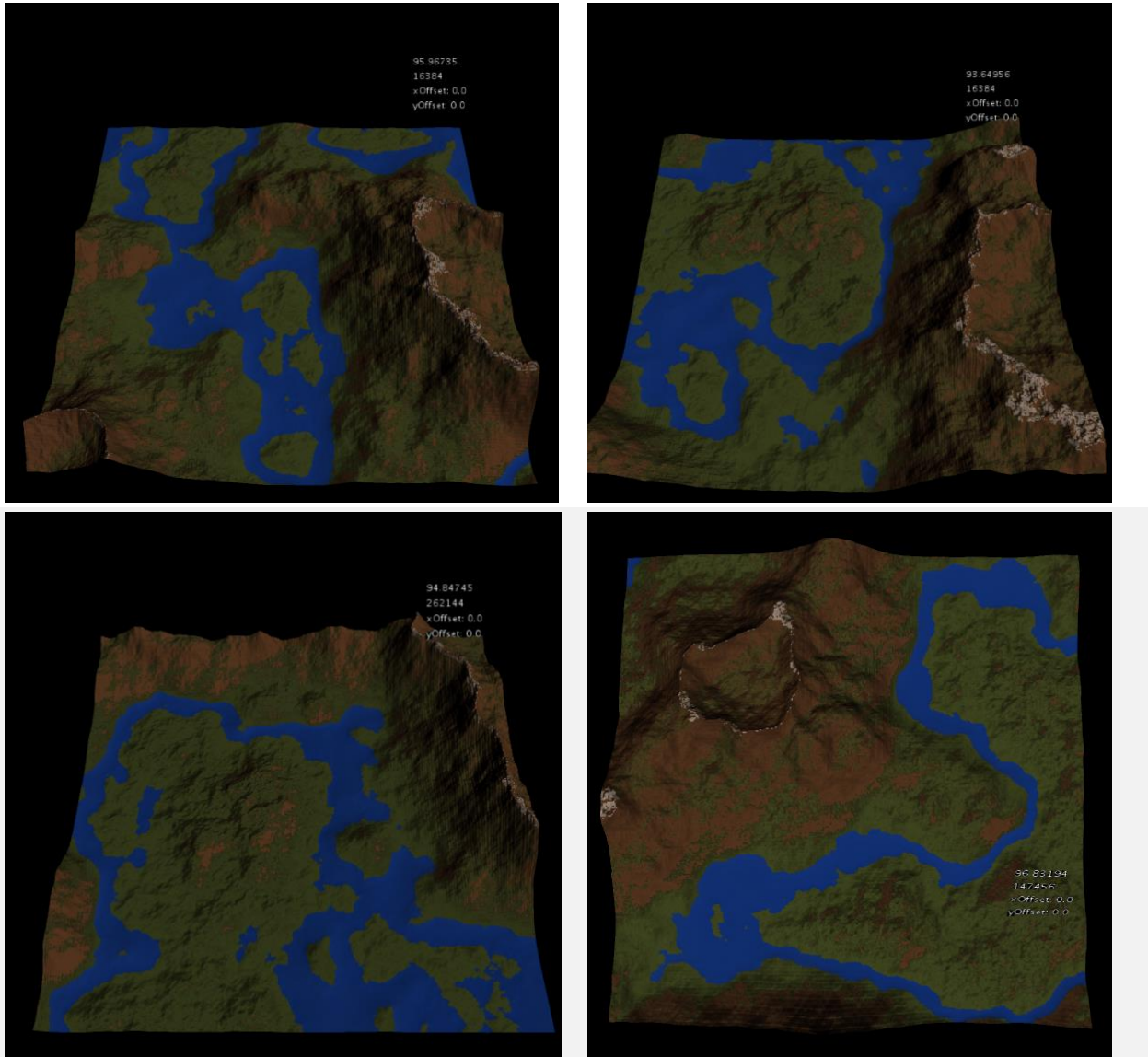


Figure 34 Different rivers generated for the terrain using similar techniques to the mpi modification.

As has been pointed out the results obtained with this method are pleasant enough to be added to the project, but they are not perfect. The main problem of this approach is the intrinsic nature of procedural-generated systems. These systems can (and eventually will) present artifacts, undesired structures which are very hard to detect, predict or fix. An example of these artifacts can be seen in Figure 35. For this work, a river-like structure was considered correct if it was complex enough. In order to determine if a structure was complex enough to be identified as a river, it must have more than two bifurcations, otherwise, it was considered an artifact. The idea behind this distinction is to exclude from the rivers the structures which form an ellipsoid path.

These artifacts among the possibilities for a river to spawn on a given point are, as said, hard to patch or fix since it is something intrinsic to the methodology itself. Nonetheless is important to remark these problems in order to consider them, analyze if the methodology is suitable for a given implementation, or even think about some method to prevent or improve these ideas.

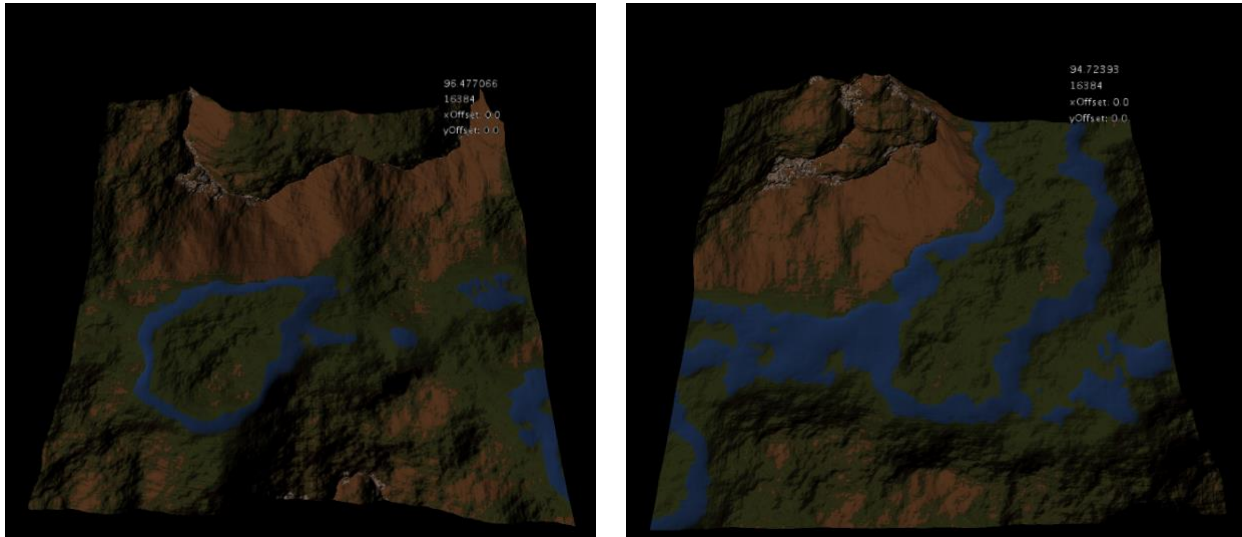


Figure 35 Comparison of an artifact (left) and a complex river structure (right).

## 6 Conclusion

### 6.1 Future updates

The possibilities of different topics to implement in this work and improve it are exciting and challenging at the same time, as it has been pointed out several times, this work was just about the implementation of a set of theoretical concepts such as the Perlin Noise function, its implementation and the different uses it could have, but there is space for better and more complex results, for that here is a list of possible future upgrades:

- For the plain terrain, the mix of these different modifications and color gradients needed to make smooth transitions from one biome to other, among the conditions with which decide what type of terrain to render.
- Although that it has been said that the noise complexity of the spherical terrain is set according to a planetary scale and that's fine, different complexity and quality optimizations could be done, for instance, a dynamical level of detail according to the camera position would be a great implementation to aim for.
- In a similar direction, the atmosphere has more aesthetic than from a simulation, to improve it, a raycasting or raymarching system should be implemented among a particle system, both will create a refractive real-time atmosphere much more interesting than the one currently implemented.
- Finally, the river and vegetation system could be improved by adding functionality that once the terrain has been converted to a static render, generate an hydroerosion procedure, which will both improve the overall terrain and determine the zone for the vegetation to grow. This vegetation will be among the shader vegetation a set of 3D trees and plants, generated with a 3D L-system.

## 6.2 Conclusions

The different aspects and milestones pursued this work had been achieved, from the construction of the 3D meshes to the development of different shaders, passing through the implementation and reviewing of different algorithms and modification techniques to generate a pleasant terrain. As it has been pointed out previously, there is space for upgrading and improving this work, but the overall result obtained is although not perfect, far from mediocrity.

For the plain, terrain the mountains, and valleys present complex structures which in some cases resemble the natural erosion of the stone. The ocean can be explored presenting complexity under the water surface. Rivers and lakes can lead to large water masses presenting natural behavior. The vegetation can grow and spread following logical conditions as the flatness of the terrain or the height. The terrain obtained is virtually infinite, letting the user advance with the certainty of never achieving an end. Multiple variables can be tuned to modify the general view such as scale, width, height, noise modification, day mode, detail, snow, sand, vegetation grow limits, water level, water movement through wind direction, and more.

The spherical terrain is able to create continental landmasses. Presents a decent level of detail for a planetary scale. Present also a good performance, especially with the simple clouds system. Provides pleasant and satisfactory views when rendering with the star system the atmosphere and the complex clouds. It also has tones of variables to be tuned and through this tuning process creates and present different planets with different biomes.

For more than 8 years when I started to study computer science, I've always been fascinated by the procedural generation and always wanted to create or participate in a project where it was used. We live in a moment in time where projects like star citizen, a game that wants to bring the possibility of exploring star systems, or even galaxies is one of the most ambitious projects ever (in videogame and procedural content) but also one of the most desired and supported by the community. This thesis has allowed me to work on these concepts, learning and practicing these ideas into something tangible and if I may say, beautiful. I'm now a bit more satisfied with my career, my work, and myself.

## List of references

- A. Mastin et al, 1987. Gary A. Mastin, Peter A. Watterberg, John F. Mareda. Fourier synthesis of ocean scenes. IEEE Computer Graphics and Applications. Vol 7, Issue 3.
- Ahrens et al, 2019. Jan-Philipp Ahrens, Baris Istipliler, Andrew Isaak, Dennis M. Steininger. Dec 2019. The Star Citizen phenomenon & the "ultimate dream management" technique in crowdfunding. Krcmar, Helmut. AISel, Atlanta, GA.
- Albersmann. et al, 1999. Albersmann, F., Zabel, A., Muller, H., & Weller, F. 1999, February. Efficient direct rendering of digital height fields. In CSI International Conference on Visual Computing (ICVC'99) Goa, India, Febr (pp. 23-26).
- Aran D. Images available at [http://www.arendpeter.com/Perlin\\_Noise.html](http://www.arendpeter.com/Perlin_Noise.html)
- Archer, 2011. Travis Archer. Procedurally Generating Terrain, article for Morningside College, Sioux City, Iowa.
- B. Mandelbrot, 1982. Benoit B. Mandelbrot. Jan 1982. The fractal geometry of nature. Times Books Editorial, New York City, U.S.A.
- Barmore, 2002. Barmore, F. E. 2002. Portraits of the Earth: A Mathematician Looks at Maps. Focus on Geography, 47(2), 36. New York Vol 47, N.º 2.
- Bradbury et al, 2014. Bradbury, G. A., Choi, I., Amati, C., Mitchell, K., & Weyrich, T. November 2014. Frequency-based controls for terrain editing. In Proceedings of the 11th European Conference on Visual Media Production (pp. 1-10).
- C. Duncan, 2019. Sean C. Duncan. October 2019. Minecraft, Beyond Construction and Survival, Journal Contribution posted for Carnegie Mellon University.
- De Carpentier & Bidarra, 2009. De Carpentier, G. J., & Bidarra, R. April 2009. Interactive GPU-based procedural heightfield brushes. In Proceedings of the 4th International Conference on Foundations of Digital Games (pp. 55-62).
- Feldmann & Hinrichs, 2012. Feldmann, D., & Hinrichs, K. (2012). GPU Based single-pass ray casting of large heightfields using clipmaps. In Proceedings of Computer Graphics International (CGI).
- Fournier et al, 1982. A. Fournier, D. Fussel, and L. Carpenter. June 1982. Computer Rendering of Stochastic Models. Communications of the ACM. Vol 25, Issue 6.
- Garland & Heckbert, 1995. Garland, M., & Heckbert, P. S. 1995. Fast polygonal approximation of terrains and height fields. Article for the Carnegie Mellon University.
- Green, 2005. Simon Green. GPU Gems 2, Chapter 26. Implementing Improved Perlin Noise, NVIDIA Corporation. Second printing. U.S. Corporates and Government Sales. Matt Pharr & Randima Fernando.

Gustavson, 2005. Stefan Gustavson. Mar 2005. Simplex noise demystified. Article for the Linköping University, Sweden.

J.P. de Carpentier, 2008. Giliam J.P. de Carpentier. July 2008. Effective GPU-based synthesis and editing of realistic heightfields. Final master Thesis for the Faculty of Electrical Engineering, Mathematics and Computer Science. Delft University of Technology.

Kim et al, 2018. Joon-Seok Kim, Hamdi Kavak, Hamdi Kavak, Andrew T Crooks, Andrew Crooks. November 2018. Procedural city generation beyond game development. SIGSPATIAL Special Vol 10, Issue 2.

Lague, 2021. Sebastian Lague. 2021, web blog available at <https://sebastian.itch.io/>

Libnoise. Libnoise noise module documentation, Perlin class reference available at: [https://hackage.haskell.org/package/Noise-1.0.1/src/libnoise/noise/doc/html/classnoise\\_1\\_1module\\_1\\_1Perlin.html](https://hackage.haskell.org/package/Noise-1.0.1/src/libnoise/noise/doc/html/classnoise_1_1module_1_1Perlin.html)

McDonald, 2021. Nicholas McDonald. 2021, web blog available at <https://weigert.vsos.ethz.ch/about/>

Perlin, 1985. Ken Perlin. July 1985. An image synthesizer. ACM SIGGRAPH Computer Graphics, Vol 19, Issue 3.

Perlin, 2002. Ken Perlin. July 2002. Improving noise. SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques 2002 Pages 681–682.

PlanetSide, 2021. PlanetSide official webpage, available at <https://planetside.co.uk/>

POV-RAY. POV-RAY online documentation, heightfields chapter, available at: <https://www.povray.org/documentation/view/3.6.1/279/>

R Tait & L Nelson, 2021. Emma R Tait, Ingrid L Nelson. March 2021 Non scalability and generating digital outer space natures in No Man's Sky. Article for University of Vermont.

R. KRTEN, 2001. Robert Krten. Jul 2001. Generating Realistic Terrain. In Dr. Dobb's Journal: Software Tools for the Professional Programmer.

Reas & Fry, 2014. Casey Reas and Ben Fry. Dec 2014. Processing: A Programming Handbook for Visual Designers, Second Edition. Published December 2014, The MIT Press, Cambridge, Massachusetts.

S. Ebert et al, 2003. David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley. , July 1998. Texturing & Modeling A Procedural Approach Second Edition. Morgan Kaufmann publishers. Burlington, Massachusetts.

Schneider & Boldte, Rudiger Westermann, 2006. Jens Schneider, Tobias Boldte, Rudiger " Westermann. 2006. Real-Time Editing, Synthesis, and Rendering of Infinite Landscapes on GPUs. Article for Computer Graphics & Visualization Group Technische Universität München.

Smith, 2014. Gillian Smith. April 2014. Understanding procedural content generation: a design-centric analysis of the role of PCG in games. CHI '14: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. Pages 917-926.

StackOverflow.2016. Image available at <https://stackoverflow.com/questions/36628064/perlin-noise-block-grid>.

Tegel & Mengotti, 2013. Rene Tegel and Tiziano Mengotti. May 2013. GPU, Framework Extensions for the Distributed Search Engine and the Terragen Landscape Generator. Pages 2,14.

Tezuka, 2012. Shu Tezuka. Dic 2012. Uniform Random Numbers: Theory and Practice Japan. Springer Science & Business Media. Page 2.

Tracy & Reindell. 2012. Sean Tracy, Paul Reindell. Sept 2012. Cryengine 3 Game Development: Beginner's Guide. Packt Publishing LTD. Page 40.

Wang et al, 2010. Hong-Rui Wang; Wei-Lei Chen; Xiu-Ling Liu; Bin Dong. An improving algorithm for generating real sense terrain and parameter analysis based on fractal. Published in 2010 International Conference On Machine Learning and Cybernetics. Qingdao, China. IEEE.

Wang et al, 2011. Zhen Wang, Meng Yang, Shi Long Xiao. May 2011. The Study on Three-Dimensional Terrain for 3D Game Design. Ran Chen and Wenli Yao. Advanced Materials Research (Volumes 230-232). Pages 798-803.

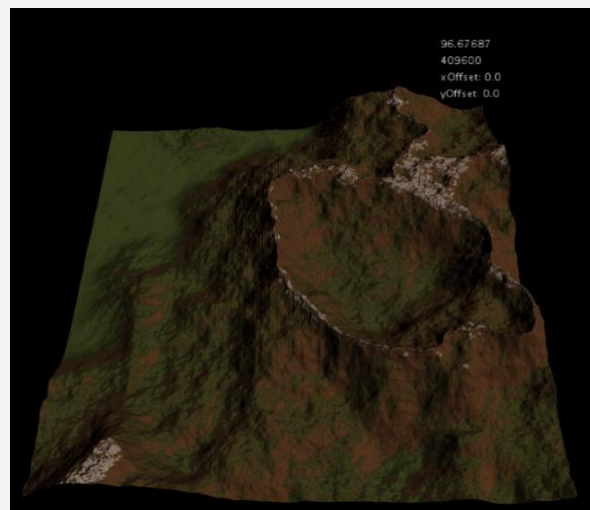
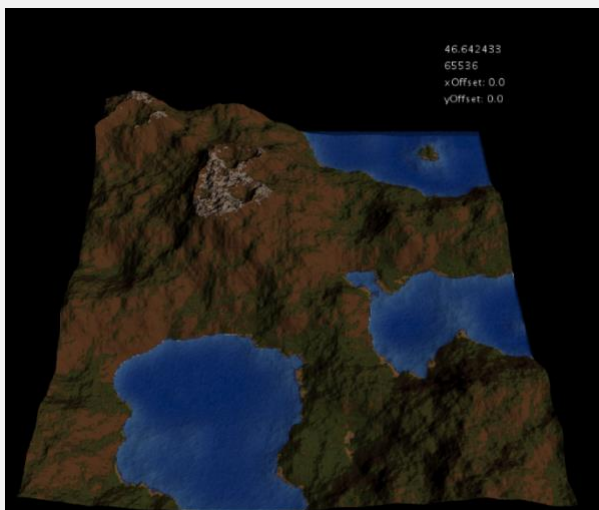
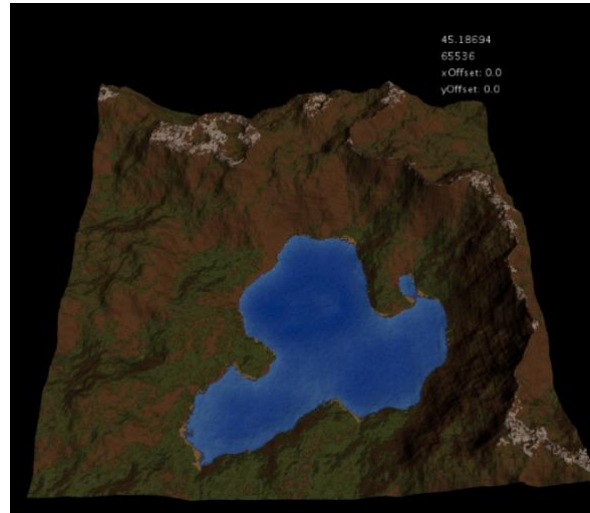
Yalcin, M. A., & Capin, T. K, 2009. M. Adil Yalcin & Tolga K. Capin. September 2009. Editing heightfield using history management and 3D widgets. In 2009 24th International Symposium on Computer and Information Sciences (pp. 442-447). IEEE.

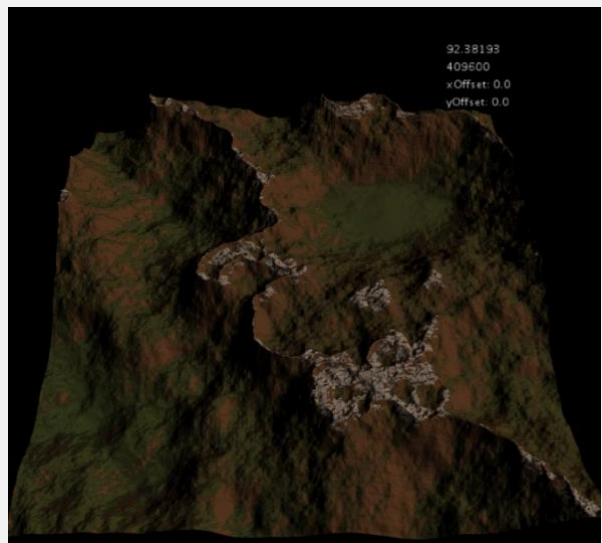
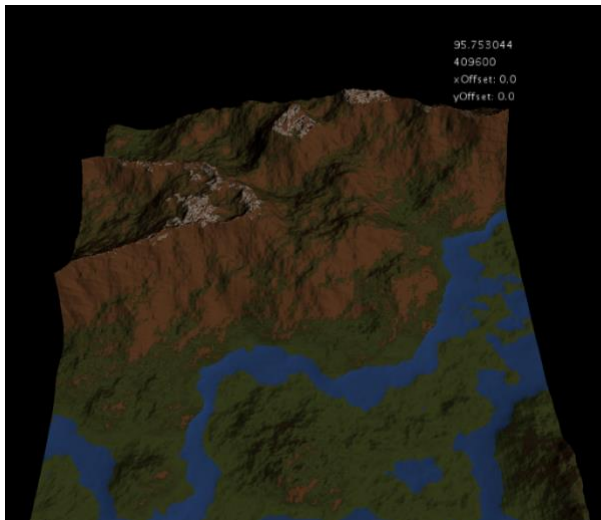
Zábský, 2011. Matěj Zábský. GeoGen – Scriptable generator of terrain height maps. Charles University in Prague Faculty of Mathematics and Physics. Computer Science thesis. Retrieved in 7.9.2011.



## Appendix 1 Plain terrain results

The following images are being rendered with the mpi modification. The Perlin noise settings were 8 and 4 octaves with a persistence value of 0.45. Scale of 0.0019. 409600 vertices ( 640 x 640 ). The day mode was set to day. The water noise parameters was stablish with 3 octaves. A persistence value of 0.5. lastly the water noise scale was set as 0.25.





## Appendix 2 Spherical terrain results

The following images are being rendered with an altered mpi modification. The Perlin noise settings were 8 with a persistence value of 0.65. Scale of 0.025. 540000 vertices ( 300 x 300 x 6 ). The day mode was set to day. The water noise parameters was stablish with 3 octaves. A persistence value of 0.5. lastly the water noise scale was set as 0.25. The complex clouds noise was stablish with 8 octaves. A persistence value of 0.64. The scale was set to 0.0175. The opacity of the clouds was calculated using a threshold of 0.8 (any value less than 0.8 will be interpreted as transparent).

