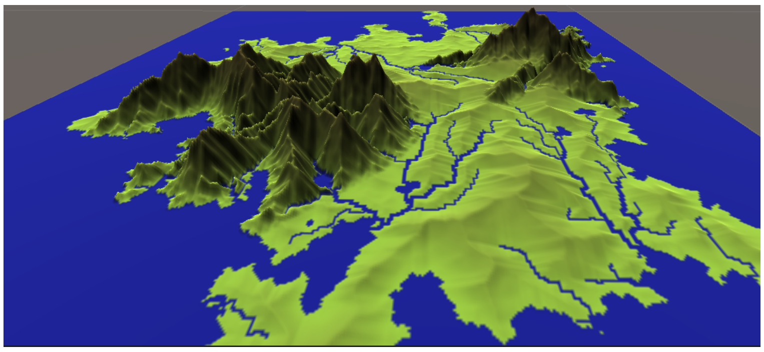
Faculty 3: Mathematics and Computer Science
Master's Program of Computer Science

Realistic 3d Terrain Generation Through Procedural Water Bodies Using Artificial Drainage Basins

Master's Thesis



Judith Boeckers
ju_bo@uni-bremen.de
Enrollment Number: 4218612

December 5, 2021

1st Assessor: Prof. Dr. Gabriel Zachmann
2nd Assessor: Prof. Dr. Rainer Malaka
Supervisor: M.Sc. Roland Fischer

Declaration

I hereby declare that this thesis is my own unaided work and has not been partially or fully submitted ad graded academic work. Only the mentioned sources and references were used. All parts that use sources according to their wording or their meaning are declared as such.

Bremen, December 5, 2021

. . . .

(Judith Boeckers)

Contents

Chapter 1

Introduction

This thesis deals with the generation of realistic landscapes in a semi automatic way. The user can define a set of parameters that control the way the landscape is constructed. The process then builds the landscape automatically based on those parameters. Different approaches were experimented with until the algorithm proposed in this thesis was implemented. The basic idea is to reproduce natural processes for terrain generation in a simplified manner. The best results were achieved when first generating water bodies like rivers, lakes and oceans and then basing the constructions of the terrain on those water bodies. After finishing the generation process, the final landscape can be visualized and exported. The data can then be used in various applications such as the generation of huge landscapes for video games or as test data for simulations.

1.1 Motivation

Procedural generation of landscapes is a topic of great relevance in research. Many studies have been performed to analyse structures of real terrain. (Jenson and Domingue, 1988) and (Costa-Cabral and Burges, 1994) are examples of algorithms proposed to analyse the topological structure and drainage basins of digital elevation models. Most of that research was done on real terrain data. Should it be possible to create data that structurally resembles real landscape then it would be feasible to use that data to run simulations in order to confirm theories without the need of a large pool of real data.
Alongside the application in the field of research, generating landscapes is also a very important focus in the development of computer games. In the past, games were mostly restricted to a linear storyline with relatively small areas and abstracted landscapes to explore. This was mostly due to the restricted power of older hardware. With the increasing calculation capacity of modern computers, the possibility for bigger and more realistic worlds emerged. This materializes in the increasing number of so-called open world games where the player can explore the map freely and in any order. Creating realistic landscapes by hand of a scale

that constantly increases is a very slow and expensive process. Procedural terrain generation could potentially reduce the time and cost needed to create believable landscapes. It could also provide the possibility to create even larger maps without the need for more data storage space, as the terrain itself does not have to be stored. Only the parameters for the algorithm would have to be saved. Even if the results would not be used directly, they could provide a solid base that respects physical factors and that artists can build upon.

1.2 Challenges

Several approaches of procedural terrain generation already exist. They focus on different goals and follow different methods. This often leads to the problem of fulfilling only some aspects of the goals for procedural terrain. While methods based on simulations of natural phenomena produce very realistic results, they lack in the possibility of influence a user can have on the generated results. Besides, simulations are often very demanding computationally and thus take a long time to calculate. (Cordonnier et al., 2016) proposed an algorithm for creating large scale realistic terrain based on tectonics and erosion. This approach shows realistic results but does not give a lot of control over the outcome. Other methods often make heavy use of noise functions. This approach is normally faster and can produce a lot of different outcomes for different parameters. The problem is the abstractness of those parameters as it is difficult to influence the result in a specific way. In (Fischer et al., 2020) multi-biome landscapes are generated by simulating wind, temperature and precipitation. The focus there is more on those simulations and dividing the landscapes into different biomes, so the heightmap generation does not produce very realistic results, as it creates the terrain using only noise functions. The challenge is to develop an algorithm that finds a balance between the different demands. Realism must be balanced against efficiency and automation against control.
A lot of research has been done in the field of river networks and drainage basins, for example in (Rosgen, 1994) who classified river shapes based on terrain slope and terrain type. (Rigon et al., 1993) performed a study on optimal channel networks in river basins. While a lot of work has been done to analyse existing structures, the focus in procedural terrain generation is often less on the generation of realistic river networks and other water bodies and more on the calculation of a heightmap. In this thesis both aspects will be combined to generate realistic terrain result with the focus on water bodies.

1.3 Goals

The goal of this thesis is to develop an algorithm which generates various and divers landscapes that resemble reality in a believable way. While doing that the algorithm should focus on creating realistic water bodies. It should furthermore fulfil a set of requirements. To do so some compromises have to be made.

On the one hand, it is desired to generate landscapes that are as realistic or as believable as possible. On the other hand, the computation time should not take too long. This is to ensure that it is possible to do multiple iterations with different parameters in an acceptable amount of time. As there are various demands for different areas of application, it should be possible to balance these two aspects with the use of parameters the user can set.
Another compromise needs to be made is the balance between the control of the user over the result and the amount of automation in the generation process. While great extend of user control makes it possible for them to achieve exactly the results they want, it also defies the purpose of this thesis. Furthermore, it would lead to large amount of time the user has to invest and also require a great deal of knowledge to create something realistic. A full automation, on the other hand, leads to the problem of results not matching the desired specification for a certain application. For example, in computer games it is often required to divide the map into regions based on different characteristics in a certain layout. This would not be possible if everything was constructed completely without user input.

1.4 Structure

This thesis is divided in six chapters. The second chapter explains the theoretical background of the methods and algorithms used in procedural generation in literature and in this study specifically. Additionally, some physical terminology and processes will be illustrated that play a role in the natural formation of landscapes. In the third chapter, the underlying concept of the developed algorithm will be presented. Furthermore, different possible approaches and methods will be analysed and discussed based on the goals defined in section 1.3. In the fourth chapter the final implementation of the most promising approach will be explained in detail. The fifth chapter begins with a complexity analysis in respect to time and space. After that, the theoretical considerations will be compared with test results. Afterwards an evaluation of how closely a section of a real landscape can be recreated with the developed algorithm will be performed. The thesis then ends with the sixth chapter in which the results are summarised and possible enhancements for future work will be presented.

Chapter 2

Theory and Principles

2.1 Digital Terrain and Procedural Generation

2.1.1 Representation of Digital Terrain

There are different methods to represent digital terrain. The most common one is to store terrain data in form of so-called heightmaps. In the simplest version of this technique, the map is represented by a function

$$f : \mathbb{N}^2 \supset M \to \mathbb{R},$$

where M is a 2-dimensional grid. This means that each cell in M contains the information of the height at the corresponding position. For more complex information, one could extend the right side of the function to a tuplet and thus store as much information as needed (e.g. terrain type, toughness, etc.). An example for terrain representation in games using heightmaps is shown in figure 2.1

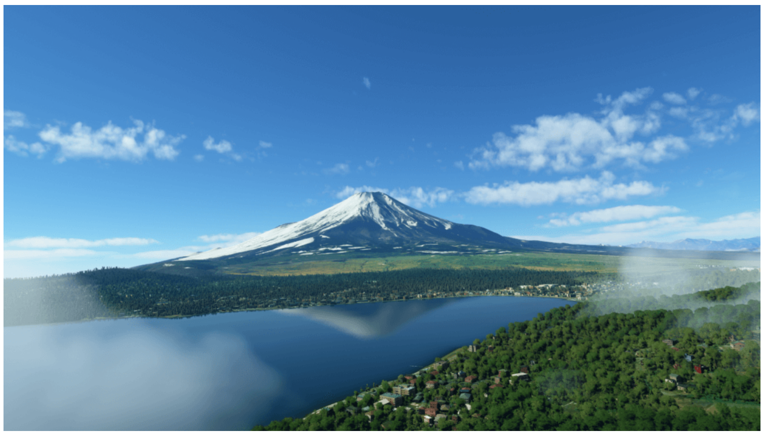


Figure 2.1 Example for heightmap terrain in the Microsoft Flight Simulator[1]

[1]Heightmap is taken from `https://www.flightsimulator.com/media/` on 12/05/2021

Another common possibility to represent terrain is to use voxels. A voxel is the 3-dimensional equivalent of a pixel. Analogously to before, we can describe the terrain through a function

$$f : \mathbb{N}^3 \supset M \to N,$$

where M is a regular 3-dimensional grid and N an arbitrary set. Because the height of the terrain is inscribed in the third dimension of the domain, it leaves the option to store different data in the image of the function. This can include the type of terrain, the softness, the flow direction of a water cell, or every other feature of a terrain voxel that is needed for the application. The Terrain can then be displayed as a 3-dimensional mesh, where each voxel is represented by a cube, connected to the rest of the map. In contrary to the representation with heightmaps, this approach allows for more advanced structures such as caves or cliffs. An example for this kind of terrain representation is displayed in figure 2.2. The information each voxel contains include the type and hardness of the corresponding block. Different types of blocks are represented by different textures on the mesh.

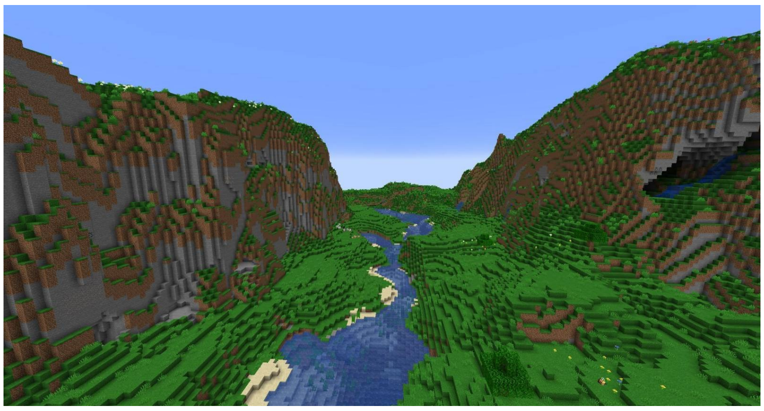


Figure 2.2 Example for voxel terrain generated in the game Minecraft

2.1.2 Voronoi Diagrams

A Voronoi diagram is a method to divide a plane or an area in different regions. To calculate the diagram over the given region one needs to start with a set of so calleed Voronoi sites. These sites will be distributed over the area (randomly or manually). A Voronoi region is a subset of the area and belongs to a Voronoi site. It includes all the points that are closer to its site than to all other sites. If all the regions are calculated, then the area is the union of all the disjoint regions. Borders between two regions are called Voronoi edges and the intersections of three or more regions are named Voronoi points.
The diagram can be calculated by using different metrics which lead to visually different results. Two examples with the standard Euclidean distance and the Manhattan distance are shown in figure 2.3. The choice of the appropriate metric depends always on the application the diagram is used for.

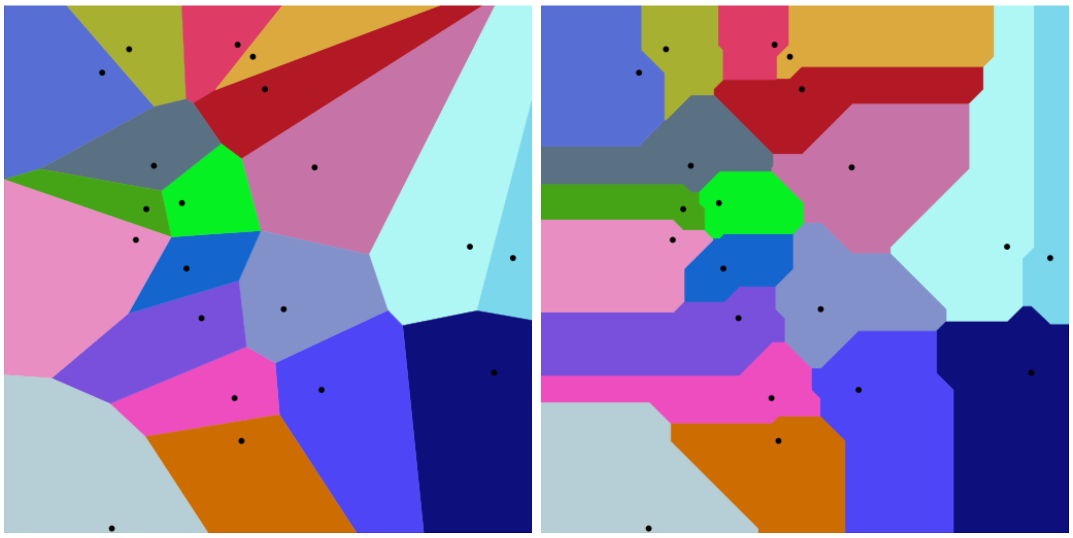


Figure 2.3 Two Voronoi diagrams with the same area and same set of sites but different metrics. The diagram on the left is calculated using the Euclidean distance (Ertl, 2015a) while the one on the right uses the Manhattan distance (Ertl, 2015b).

These diagrams can be used in many different applications. A typical utilisation would be to use it as areas of influence. If, for examples, a chain of supermarkets represents their already existing stores as Voronoi sites, then the best candidates for new market locations would be the Voronoi points, because they represent the points that are farthest from the tree ore more closest markets.
Another interesting application is the procedural generation of textures. For instance, if the distance to the closest Voronoi edge is calculated and mapped to a colour value, it can be used to generate a stylised water or lava texture. Furthermore, the sites could be moved over time to give the illusion of moving waves.

2.1.3 Noise

The term noise describes the random distribution of values over a defined base space. The space can be of any dimension. The most commonly used variation is 2-dimensional noise. Therefore, the focus in the following section will be on only two dimensions, but it can be easily extended to any desired dimension using the same concepts. There exist many kinds of noise. Many references on this topic can be found in literature. For example, a wide range of the most common noise types was discussed by (Lagae et al., 2010). These types including white noise, perlin noise, cellular noise and fractal noise will be presented below.

White Noise For each cell of an $n \times n$-grid, a random value between 0 and 1 will be generated. Thus, every cell stands on its own and has no relation to the neighbouring cells. Besides, each value appears with the same probability. Therefore, the noise is the realisation of n^2 identical and independently distributed random variables. An example of white noise can be seen in figure 2.4.

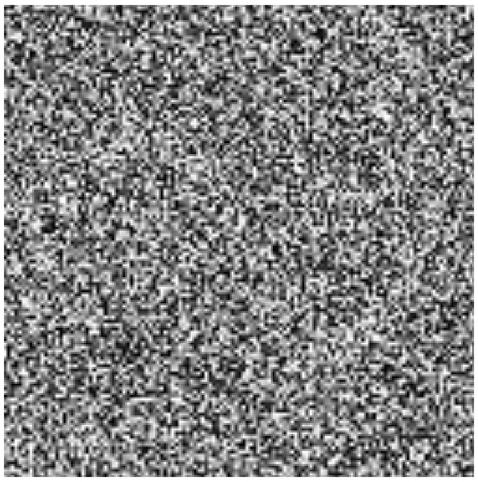


Figure 2.4 Example of 2-dimensional white noise on a 128x128 grid. The image was generated with the software FastNoiseSIMD from (Peck, 2020).

White noise is the most basic form of noise. Because neighbouring cells are not dependent on each other, the difference between adjacent noise values can vary considerably. Moreover, it is almost impossible to get an outcome with smooth transitions along any direction of the grid. Even though this form or noise is very fast to compute, it is normally not used for applications that need smooth noise maps, like the generation of digital terrain.

Perlin Noise Perlin noise is an example of procedural noise that uses adjacency information to generate smooth transitions over neighbouring noise values. The first step in the generation of the noise for a 2-dimensional base set is to overlay it with a grid. The size of the grid determines the frequency of the resulting noise. For every corner point of the grid, a (pseudo) random 2-dimensional normalized vector is generated. To get the noise value for a point p, the grid cell is determined in which p is contained. Then, for each corner of that cell, the dot product between the corresponding random vector and the offset of p to that corner is calculated. By linearly interpolating these four values regarding the relative position of p in the cell, the final noise value is calculated. An example result for this procedure is shown in figure 2.5.
Because of the way this kind of noise is calculated, it produces a smooth result. Therefore, and because of its relatively simple calculation steps, it is a popular choice for procedural terrain, texture or surface generation.

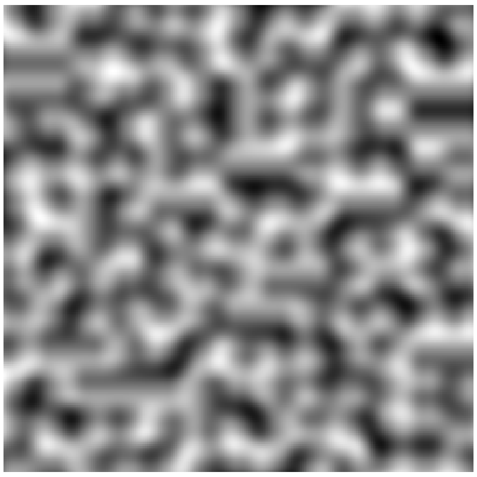


Figure 2.5 Example of 2-dimensional perlin noise on a 256x256 grid with a frequency of 0.05. This corresponds to a size of $256 \cdot 0.05 = 12.8$ pixels. The image was generated with the software FastNoiseSIMD from (Peck, 2020).

Fractal Noise Fractals are structures where the whole structure can be found in subsets of themselves. Fractal noise takes this key concept to generate noise based on other noise generation procedures. Different versions of the same noise are stacked on top of each other. Each layer adds noise with increased frequency but lower amplitude. Therefore, the first layer determines the global structures, while each subsequent layer adds smaller, local details. These layers are called octaves. The higher the number of octaves, the closer the result gets to an actual fractal. Examples with different numbers of octaves based on the perlin noise are shown in figure 2.6.

Figure 2.6 Example of 2-dimensional fractal noise on a 256x256 grid. The noise function used is perlin noise with a base frequency of 0.05. The number of octaves from left to right are 1, 2 and 3. The images were generated with the software FastNoiseSIMD from (Peck, 2020).

Fractals are structures which are found in different parts of nature, because construction rules can be both applied globally and locally. Therefore, using fractal noise in the generation of natural objects can lead to more realistic and natural results.

Cellular Noise Cellular Noise, as the name suggests, is a form of procedural noise that calculates its values by dividing the area into smaller cells. This can be done in multiple ways, but the most common way is to use Voronoi diagrams, which are explained in detail in section 2.1.2. Depending on the used distance function, the cells are generated in a different way. There are multiple possibilities of assigning values to the cells. Each point of a cell can have the same value. This value can be chosen either randomly or by using another noise function like perlin noise. By looking up the noise value of the generating points of the Voronoi regions and then assigning that value to the whole region, a less arbitrary result is produced. It is not smooth like perlin noise, but neighbouring cells follow a kind of gradient. This is only the case if the frequency of the underlying noise function is set appropriately. If the frequency is too high, the results are very random again. This form of cellular noise can be used, for example, to divide a map into different territories. Figure 2.7 shows examples of both options.

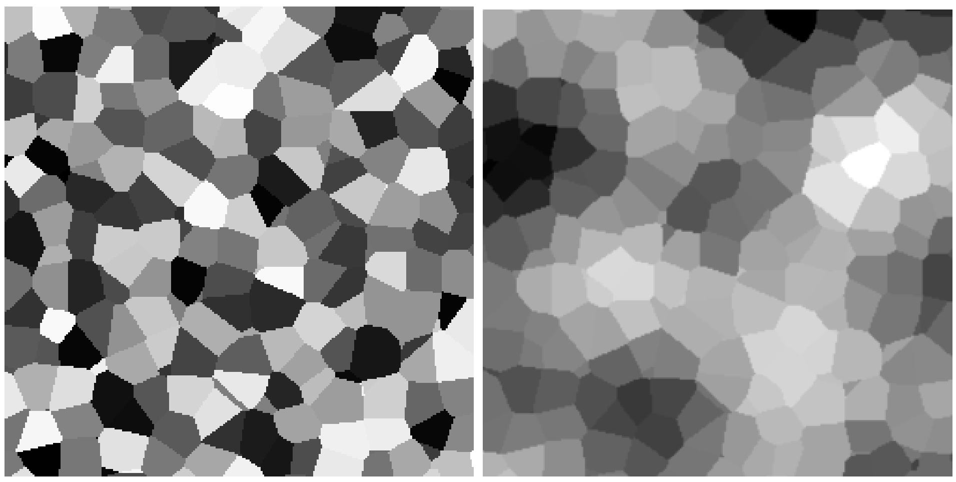


Figure 2.7 Example of 2-dimensional cellular noise on a 256x256 grid with a base frequency of 0.05. Each cell has a fixed value. On the left, the values are chosen randomly, while on the right the values are defined by an underlying perlin noise texture. The frequency of the perlin noise is 0.2. The images were generated with the software FastNoiseSIMD from (Peck, 2020).

Instead of assigning a fixed value to each cell, it is also possible to use a value that depends on the distance of a point to the corresponding cell border. The distance function chosen for that again has an influence on the appearance of the result. Examples for those options are displayed in figure 2.8.
By interpolating between different colours based on the values, this type of noise could potentially be used for water textures in games with a more stylized graphic style. Instead of using a static texture, an animated one can be produced by moving the generating Voronoi points over time. This can create the illusion of moving waves.

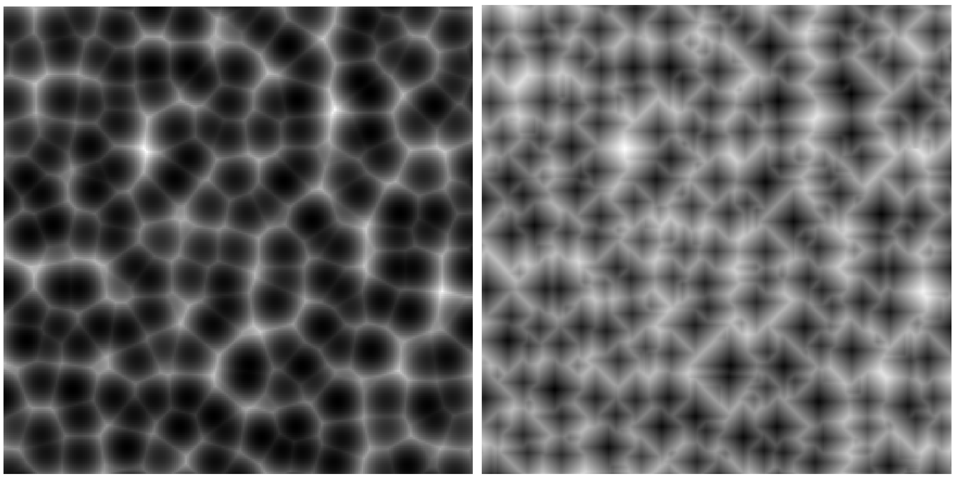


Figure 2.8 Example of 2-dimensional cellular noise on a 256x256 grid with a frequency of 0.05. The distance to the border of each cell is used to calculate the values. The image on the left uses the Euclidean distance, while the image on the right uses the Manhattan distance. The images were generated with the software FastNoiseSIMD from (Peck, 2020).

2.1.4 Pathfinding Algorithms (A*)

Pathfinding algorithms are a form of search algorithm on graphs. Beginning from a starting state, the algorithm searches until it finds a state which is included in a set of target states. Search algorithms can be separated into two different categories, informed and uninformed search. A search is called informed when it uses information about the current state to its advantage. If it does not use additional information, it is called uninformed. A typical example of an uninformed search is the so-called Dijkstra search algorithm, which operates on weighted graphs and only takes the cost of the path to the next node in consideration. The A* search algorithm (often seen as an extension to the Dijkstra algorithm) can be classified as informed. It makes use of a heuristic function to not only evaluate the cost to the adjacent node of the graph, but also an approximation of the final cost. A* first evaluates the node n which minimizes

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the cost of the path from the start to node n and $h(n)$ is the value of the heuristic, which approximates the cost from n to the target.
A heuristic function h is called admissible if it fulfils

$$h(n) \leq h^*(n) \text{ for all } n,$$

with $h^*(n)$ being the optimal cost for a path from n to the goal. If the A* algorithm uses an admissible heuristic, it is guaranteed that the result of the search is optimal. If h is monotone, meaning that it satisfies the condition

$$h(m) \leq c(m,n) + h(n) \text{ for all } m, n,$$

the algorithm also finds the solution without evaluating a node twice. Hereby $c(m,n)$ represents the cost from node m to node n.
The choice of the heuristic depends on the application. A heuristic that is admissible in one scenario can behave differently in another one. For example, in the case of fining the shortest route on a grid from one point to another, the Manhattan distance between a point and the goal is admissible. On the other hand, in the application of finding routes on a map, the Euclidean distance between two points is always shorter or equal to the Manhattan distance. In this case. the same heuristic would not be admissible.
Pathfinding algorithms can be used for a lot of applications, like navigation systems. With the right cost and heuristic functions, it could also be a good candidate for calculating rivers paths.

2.2 Physical Terms and Processes

2.2.1 Drainage Basin

An area that collects precipitation and channels it to a single outlet is called drainage basin. Multiple terms exist for the same mechanism. These include catchment areas and water sheds. The different terms are used interchangeably in the following. The water collected can be from rain, melting snow or other sources like rivers or subsurface sources. The outlet which the precipitation connects to can be a river, a lake, the ocean or a point where the water dissolves into the earth. Each basin can be divided into smaller basins in that there exists a hierarchical structure of catchment areas where one is included in multiple ones of higher order. Individual basins are separated by higher elevations like mountains or hills forming a natural border. Figure 2.9 shows a simplified structure of a drainage basin.

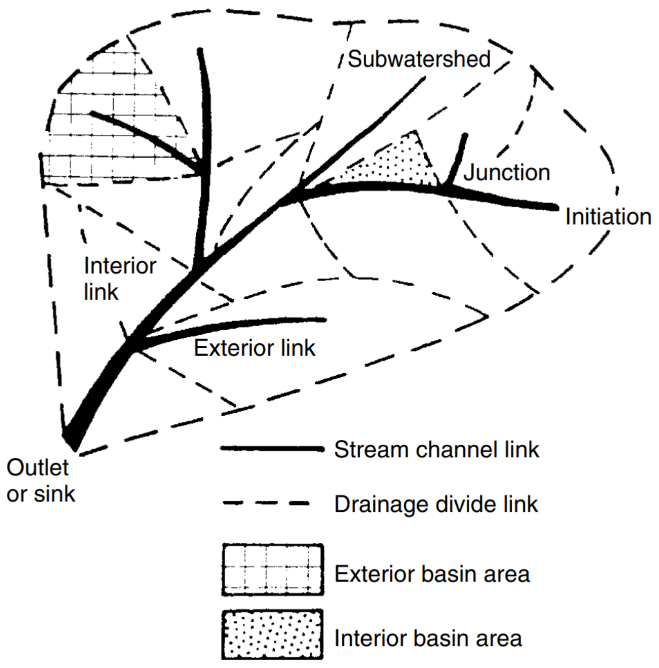


Figure 2.9 A schematic showing the general structure of a drainage basin (Lin et al., 2006)

A lot of different approaches were made to extract catchment areas from existing terrain data based on the heightmap. The most common data representation hereby is a regular grid with a height value for each cell of the grid. Most algorithms on those grids work by calculating the amount of water flowing through a cell and then assume that cells above a certain threshold belong to a water stream. A problem that arises in most cases is the existence of local minima in the elevation data. For example (Jenson and Domingue, 1988) tried to work around this by making use of a depression filling algorithm. Another challenge is that different areas may produce the optimal result with different thresholds. (Lin et al., 2006) proposes a method to extract the optimal thresholds automatically by means of using a fitness index.

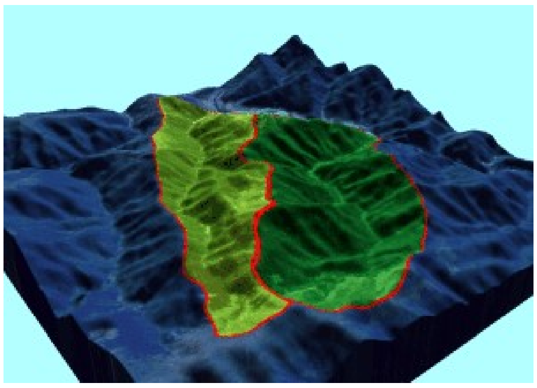


Figure 2.10 A 3-dimensional model of an area of terrain divided into drainage basins (Asybaris01, 2011)

2.2.2 Fluid Dynamics

Fluid dynamics refer to the description of the flow of fluids. The term fluid refers to liquids, such as water, or gases, e.g. air. It is based on sets of equations which describe the physical rules that control the flow. Those equations build on the assumption that energy, mass and momentum are conserved. Different factors need to be considered when analysing and calculating fluid behaviour. In addition to properties of the fluid itself, such as the viscosity, density and its momentum, also outside forces have to be included. As the calculation of fluid motion can be very complex, simplifications of the model are often made. A common practice is to assume the fluid is incompressible. (Harris, 2003) bases their approach on this assumption and uses the commonly used *Navier-Stokes equations*

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} - \frac{1}{\rho}\nabla p + v\nabla^2\vec{u} + \vec{F}$$
$$\nabla \cdot \vec{u} = 0,$$

where ρ is the fluid density, v the kinematic viscosity and $\vec{F}$ the vector describing all external forces acting on the fluid. The whole model is based on a velocity vector field with vectors $\vec{u}$. These equations can simulate the movement of fluids over time

2.2.3 Erosion

In geology the term erosion refers to the process of removing material like stone or soil by breaking it into smaller pieces through movement. It is a process that appears on the surface and is caused, for example, by rainfall, movement of water bodies (such as oceans or rivers) and wind. The material is removed in one place and transported to another. These different types of erosion have a different effect on the landscape they interact with. Flowing water, for example, forms riverbeds and rain and wind can cause terrain masses to break and slide down a hill or form a cliff. Different types of material behave differently under the influence of the same movement. In the case of flowing water, softer materials like soil are removed much easier than, for example, stone. If a river traverses from hard terrain to soft terrain, the softer one is eroded a lot faster which, over time, can lead to gaps and as a consequence to waterfalls. These differences in soil composition together with the slope of the terrain also have an effect on the shape of riverbeds. (Rosgen, 1994) proposed a classification of stream types based on those factors. This is shown in figure 2.11.

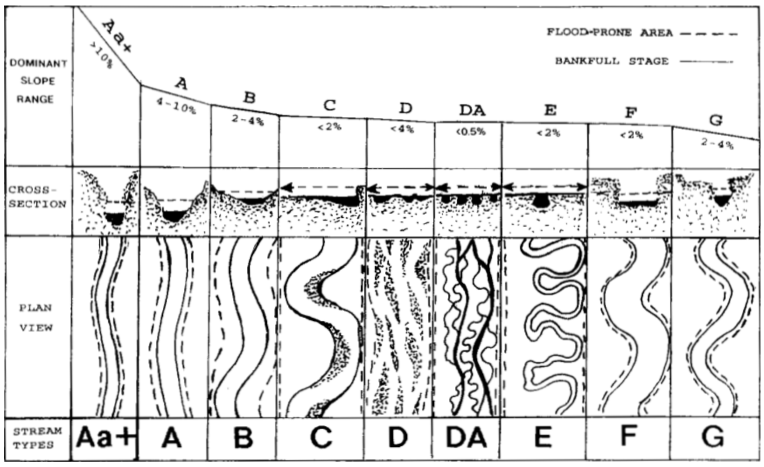


Figure 2.11 Classification of rivers by cross section and plane view (Rosgen, 1994)

A lot of different algorithms were developed to simulate the process of erosion. A common approach is to begin by distributing water randomly on an existing terrain. The flow of the water is then simulated and that simulation is then used to calculate the movement of material. Several different approaches to perform this simulation were proposed. (Musgrave et al., 1989) were the first in this area. They proposed an algorithm that works on a mesh of vertices with a fractal approach. They were followed by more studies, including (Mei et al., 2007), who chose a more physical approach by using a shallow-water model for the flow simulation which is based on the Navier-Strokes equations (see section 2.2.2).

2.2.4 Plate Tectonics

Plate tectonics have been the subject of many studies. (Viitanen, 2012), (Le Pichon et al., 2013) and (Cox and Hart, 2009) explain the process in more detail. The earth consists of multiple different layers. The outermost layers are called the lithosphere and are responsible for the process that is called plate tectonics. Figure 2.12 shows this concept of the layers in a cutout of the earth. The lithosphere contains different plates which move very slowly over a very long time. This movement shapes the terrain in different ways and can also have some other influences.

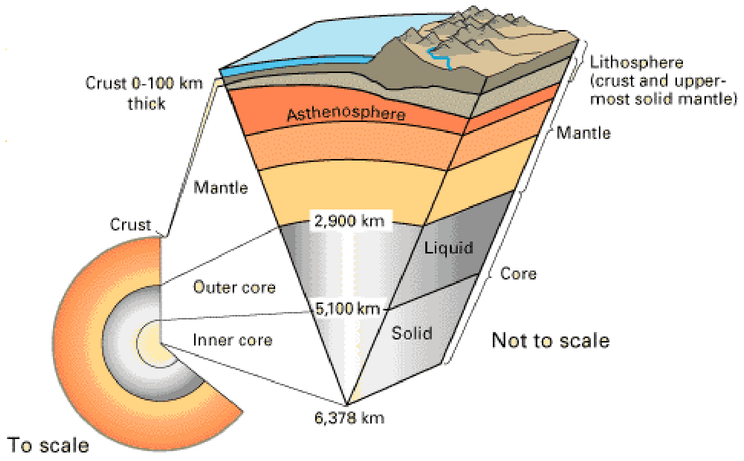


Figure 2.12 A schematic illustrating the different layer of the earth (Viitanen, 2012)

The relative movement of two plates is categorized into different types. Plate boundaries that move towards each other are called *convergent*, while those which move away from each other are called *divergent*. The type of movement where two plates slide alongside each other is called *transform*. This is illustrated in figure 2.13. These different types shape the terrain in various ways. Mountains or ridges can be constructed over time. Earthquakes or volcanic activities are another possible effect of this movement. The influence of plate tectonics normally happens on a larger scale than other geological terrain shaping effects such as erosion.

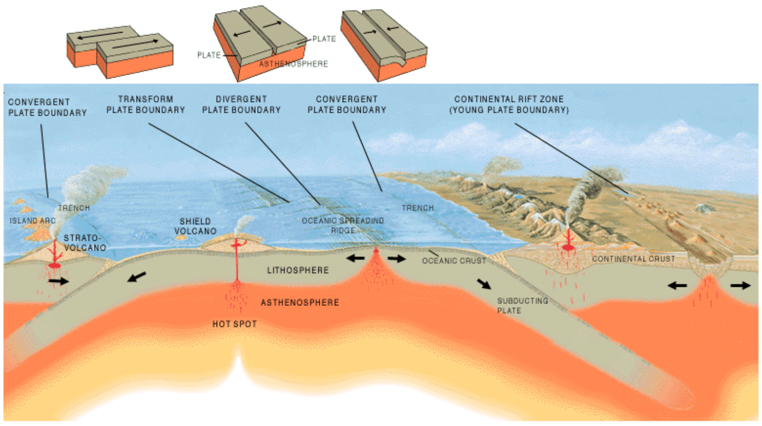


Figure 2.13 A schematic showing the different types of plate boundary interactions (Vigil., 1997)

Plate tectonics have also been researched for the use of physically based procedural terrain generation. In (Viitanen, 2012), an algorithm is proposed to simulate the movement of plates and generate terrain using the collision information. Another example for the use of plate tectonics is (Cordonnier et al., 2016) who used plate tectonics as well as erosion to create realistic landscapes based on a vector representation of the map.

Chapter 3

Concept

This chapter is dedicated to developing a concept for the most optimal algorithm giving the specifications made in section 1.3. First different general options like the representation of data will be discussed. This is followed by a section evaluating different methods for the actual algorithm based on the decisions made in the first section. Finally, the strategy pursued in the following implementation will be revealed and the reasons leading to these decisions will be explained.

3.1 Possible Methods and Techniques

3.1.1 General Considerations

Software The algorithm that will be developed is a tool to generate landscapes so it is natural to have a way to display the produced results in a graphical way. It would be possible to write all the code to do that, but the logical step to take is to use a tool that is already designed to do this, because it is not the aim of this thesis to develop a rendering algorithm. Since an important application for the results of this paper is in the field of game development, a game engine as the development environment will be used. 3d rendering is included there so the main focus will be on the relevant parts. Furthermore, this yields the important advantage of a lot of useful tools like already implemented vector math and mesh generation tools.
There are a lot of Game Engines capable of accomplishing this task. In this project the Unity 3d Engine will be used. It is one of the most used engines in the game industry and also includes a free licence. It supports the programming language C# in which this algorithm will be programmed.

Online/Offline The decision whether the algorithm should be implemented to be used on-line or offline affects the way it can be used in applications. Online implies that all calculation

are done in realtime. For this, the generation of the terrain would then be split into separate sections which can be calculated independently of one another and then be connected. This leads to the advantage of infinite landscapes that can be generated live when a player explores a game. This approach is pursued for example in the game Minecraft. There the map is split into chunks which will be generated when the player comes close to them. Offline algorithms, on the other hand, are executed asynchronously without life information. This means that in the case of terrain generation the terrain would be generated completely in advance. With this strategy it is not possible to construct infinite landscapes. The advantage for this approach is that more time can be spend on the generation and different parts of the terrain can share information about each other.
In this thesis, the approach of an offline algorithm will be followed. One reason for that is the generation of river networks which are connected globally and would have to share information to be created in a believable way. Furthermore, the developed algorithm should be designed to not produce completely procedurally terrain, but also give a lot of control to the user about the global layout and other features. Using an online approach would make more sense if the goal were to generate infinite terrain. This does not work well with terrain that should be controlled by the user. The last important point is, that offline methods allow for the opportunity to use more time for complex calculations which are probably needed to follow the idea of creating realistic terrain.

Terrain Representation An important decision that has to be made before the actual algorithm can be worked out is how to represent the digital terrain. Two different methods commonly used will be discussed here. Both are explained in more detail in chapter 2.1.1.
The first method is to use so-called heightmaps. This method uses a 2-dimensional grid and stores the height information for each cell. This is relatively simple compared to other methods which use a higher dimension for the base set. This also means that calculations concerning neighbouring cells are faster because there are only neighbours in two dimensions. Furthermore the data can be stored easily as image data with grey levels in each pixel for the height information in the corresponding cell. This yields the advantage of simple portability of data between different applications without one knowing about the implementation details and data structure of the other. Additionally heightmaps are the method that is most commonly used in game development and is supported by most of the game engines.
The second possibility of storing terrain data to be discussed here is the method of using voxel representation. Voxels are 3-dimensional pixels which means that the terrain is stored in a 3-dimension grid instead of a 2-dimensional one as in the heightmap approach. Each voxel can then store the type of terrain that is represented. The most simple way would be to store just a boolean for each voxel to indicate whether there is land or air. For a more complex approach one could store the exact type of terrain like for example rock, sand or water. This method offers more possibilities for different shapes of terrain because of the third dimension. This allows the generation of structures like caves or cliffs that would not

be possible otherwise. On the other, hand this increases the complexity and time that is needed to perform calculations.
After considering advantages and disadvantages of both options, it was decided to base the algorithm developed in this thesis on the use of heightmaps. The most important reason for that is the compatibility with game engines. The most important application for this kind of algorithms is to generate terrain for games ,therefore, it is beneficial to have it integrated well. With this approach the possibility to generate cliffs and similar structures is lost but it provides the opportunity for faster runtime which leads to more iterations to get to desired results.

3.1.2 Potential Approaches

Creating digital landscapes can be realised with different approaches. These can be divided into three categories. One option is to generate water bodies and terrain simultaneously. The other possibility is to generate both parts individually. This leads to a second approach to first generate terrain and then water bodies on top of it. The third option would be to generate the landscape in reverse order. In this section, these different strategies will be presented, and their advantages and disadvantages will be discussed.
The first possibility to create everything simultaneously is normally realised with the use of simulations. This means that, instead of defining several steps that are performed one after another, the program defines a set of rules and then in a certain amount of time each part of the simulation acts according to those rules. These rules can represent a wide variety of phenomena and depend on the context of the simulation. In the context of terrain generation, this would probably include simulations of plate tectonics (chapter **??**) or erosion (chapter 2.2.3). With the use of fluid simulation (refer to chapter **??**), water, soil and stone can be represented by particles and the rules define how each particle moves depending on its state and how it interacts with other parts of the environment. This includes, for example, how easily water particles can remove and carry different kinds of terrain particles to simulate erosion (refer to chapter 2.2.3). In this manner, it is possible to emulate natural processes very closely. The major advantage is the realism that can be achieved with this approach. This requires a lot of complex rules, a large amount of particles and many calculation steps, all of which lead to the problem of long computation times. Depending on the number of simulated systems and the desired degree of realism, this can sometimes take several days. Another issue is the lack of control over the result of the simulation. This can be a problem when the generated landscape has to meet, for example, specific layout criteria. To fulfil those criteria, either more systems have to be implemented or the simulation has to be controlled in other ways.
The other option would be to generate different parts of the terrain separately. This can also be divided into different possible strategies. The main difference between them is the order of actions and how to perform each step. Because this thesis puts an important focus on procedural water bodies, the generation of those items can be classified as one important step.

Another one is the generation of the heightmap. Each of those steps can then be partitioned even more, but the main question is in which order to perform those main steps. Since there are two steps, two possibilities arise for the order of them. The first option would be to generate the terrain and then place the rivers on top of it. The terrain could be generated for example with the utilization of noise functions introduced in chapter 2.1.3. Most types of 2-dimensional noise produce a map with smooth transitions that can be used directly as a heightmap. Different types of noises can be combined with different settings to create a vast amount of shapes for the terrain. After constructing a heightmap the next step would be to generate water bodies on top of it. Because the water level of the ocean is always at the same height, the easiest way to generate it would be to classify each cell as ocean, that is located under a specific height threshold. This requires a good set-up of terrain height distribution from the previous step to generate convincing coast lines. Without that, it can result in a large quantity of separate ocean regions such as big lakes.
Creating river networks can be accomplished with different approaches. First, there is the possibility to place river sources (either manually or by means of randomization) on the map and then create the river by calculating the flow directions using path finding algorithms as described in chapter 2.1.4. Path finding algorithms can not only be used to calculate the shortest or fastest route to a target position. With the right cost functions, it can be applied to many problems. Rivers, for example, generally flow along the path which takes the least amount of energy. This does not always mean the shortest path. Consequently, finding a cost function that leads to that behaviour is important. It includes the distance travelled but also the steepness of the path. The advantage of this approach is that lakes can be generated along the way by including both the result of the algorithm but also the searched area. For that purpose, the A* algorithm explained in chapter 2.1.4 could be used with a fitting heuristic. A result of this procedure is depicted in figure 3.1.

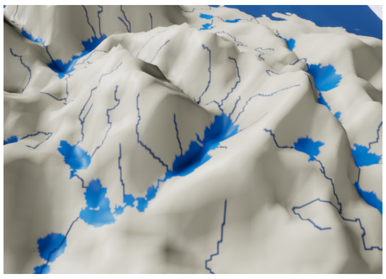


Figure 3.1 Section of a river network with lakes generated with A*. Dark blue depicts river cells resulting from the shortest path calculated. The lighter blue areas represent the areas searched by the path finding resulting in lakes.

Another way to define river networks would be to analyse the catchment areas of the terrain. This was done, for example, in (O'Callaghan and Mark, 1984). By following the steepest slopes, for each cell the amount of water flowing through it is calculated. Based on that, every cell that lies over a specified threshold can then be classified as cell. This approach holds the advantage of not generating river sources in unrealistic location like the middle of

a valley or the immediate vicinity of an ocean. The issue is that while it works well on real terrain it can face major problems when dealing with randomly generated terrain. Because rivers merely flow the steepest slope downhill, a river stops if it runs into a local minimum. This means, that either a depression filling algorithm has to be performed after the random heightmap generation or the rivers would have to be connected afterwards. A possibility for the latter would be to fill the depression with a lake until it overflows. The only problem then is to find another river afterwards to connect to. Also, because every cell is categorized separately, there is no information about neighbouring cells. This means an extra step has to be done to find those dead ends. Additionally, when placing rivers in this fashion, the way they are generated is determined almost completely by the previous step. This reduces the possibilities for the user to take control over the process and influence it to their liking.
A problem that both approaches have in common is that they work well with real terrain, but can produce unrealistic results when the terrain is generated in an arbitrary way without respecting natural procedures. This means finding a way to create realistic terrain in the first step would be necessary.
The last option which should be discussed here is to also generate heightmaps and water bodies separately but in reverse order in comparison to the last approach. Creating the water bodies would be the first step and based on that, the heightmap would be created. Even if this is not a simulation, the natural process of erosion, described in chapter 2.2.3, is approximated slightly more. Erosion means, that over time rivers take parts of the terrain with them when they flow. This leads to lower heights where there are or have been river beds. Here a kind of opposite process would be performed. Instead of taking height away in water regions, height is added between them. With the appropriate strategy, this could potentially create realistic looking hills and mountains between river networks. This would have to include a random factor, so that the mountains would not look to similar to each other. Furthermore, the terrain should not be created in the same way everywhere. Other geographical phenomena like plate tectonics (see chapter 2.2.4) shape the landscape in a different way for different regions. There have to be regions that are flat and regions that have bigger elevations up to very high mountain ranges. This could either be produced by another random step or could be potentially controlled by the user.
Before the terrain between the rivers is lifted, there first have to be river networks. These also have to be generated.

3.2 Final Concept

Even though this algorithm does not aim for a generation in realtime, short generation times would be ideal. The goal of this project is to develop a procedure that allows multiple quick iterations with different settings for prototyping until the desired results are achieved. This is problematic when deciding on the physical simulation approach. As discussed previously, simulations normally are significantly slower than other procedural generation approaches.

Even if one run takes just a few minutes, this will be a very long waiting time, if the user is not sure what result he wants exactly or how to achieve them. Furthermore, fluid simulations are done best with the use of particle systems. Because the algorithm should be designed to work well for game development, it would be necessary to transform the results in a heightmap so it can be used with most terrain tools in game engines. These considerations have led to the decision to not employ this approach. Instead, different parts of the terrain will be generated separately in the form of a pipeline. The relevant questions here are the steps in which the generation will be split into and the order of those steps. The parts that have to be implemented are the different types of water bodies and the height of the terrain. There are different factors that can influence this decision. One the one hand, there are the results that can be achieved with each possibility and, on the other hand, the kind of control the user could have over each part. One important thing the user should be able to define is the general shape of the landmass. This means that everything else has to depend on this. Consequently the separation of ocean and landmass (and with that the coastline) has to be the first step.

The other area that the user should have a lot of influence on is the decision about what the general structure of the landmass should be. This should not be too specific but provide the possibility to decide roughly on where to generate mountains or other types of terrain. These different terrain types can then have an influence on how later steps perform. For example, different regions can have different height distributions or different ways of generating rivers. Using cellular noise introduced in chapter 2.1.3 is an option to divide the map into different kinds of regions. The problem with this is that it leaves not enough control over the distribution of terrain types. Hence it was decided to apply a more manual approach, by means of which the user can specify exactly where to place different regions. Randomization can then be performed in the calculation of the border between different regions. Different region types should, for example, enable the option of dryer types like deserts, where no rivers are generated. Also flat landscapes and mountainous areas should be a possibility.

The next step in the pipeline should then be the generation of river networks. This should be a mostly procedural process. It ought to be possible to specify some parameters that have an influence on the general shape or number of rivers, but it is not desired to set, for example, every single river source or outlet. Parameters for the user should also take in consideration the different regions generated in the previous step. The density of the networks or the amount of branching could be possible options that could be also set differently for different region types. Rivers should grow in strength and width over time as they do in reality. This can then also affect other properties like the size of lakes they flow in and out of.

Although lakes rarely exist without a connected river, it is a rather rare phenomenon. This lead to the decision to not have an extra step for the lake generation but instead to include it in the process of river generation. Lakes should generate randomly along the river courses and vary in size and shape. The user should have an influence over the amount, size and general shape of lakes, without having to specify this for single lakes.

The last part of the actual terrain generation would have to be the calculation of the heightmap. As discussed in the previous chapter for this approach, the terrain should grow between the water bodies. The amount of growth, and with that the final height, should be something the user can control. Also it should be possible to have different settings for different terrain types. A problem that has to be addressed is the interpolation between the different regions. Having a hard transition of steepness at the borders between terrain types would lead to unrealistic results. Furthermore, growing the terrain straight from the rivers would make every hill and mountain look the same and would result in an unrealistic look. For that reason some kind of randomization should be employed to give the heightmap a more natural flow. Using noise textures would be a good first idea on tackling this problem. By setting different parameters of the noise functions, this can also give the user another way of controlling the outcome.

Finally, there has to be a visualization step. Even though this is not part of the actual terrain generation, it serves an important purpose. Without some form of graphical representation, the user has no possibility to verify if the results they achieved with the parameters set is what was intended or not. There has to be an option to evaluate the outcome. This can be done in form of showing textures, like the heightmap, but a 3-dimensional representation would be more advantageous. Therefore, a 3-dimensional mesh should be generated to give the possibility to view the generated terrain from different angles. To improve the visualization, textures should be generated for the mesh to outline different properties of the terrain in a better way. This should include information about height and terrain types. These textures ought to be exported and store all the relevant information to recreate the terrain with other programs such as game engines or other terrain creation tools.

Figure 3.2 illustrates the order of the steps that were discussed in this chapter.

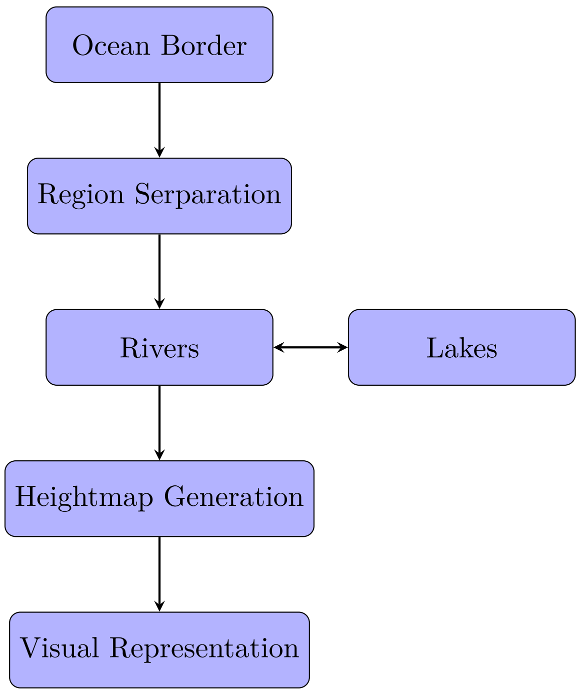


Figure 3.2 The concept of the pipeline model for the algorithm

Chapter 4

Implementation

This chapter provides a detailed explanation about how the final algorithm is structured. Firstly, it begins with an overview over the general structure of the pipeline with a short summary of the purpose of each step. This will be followed by the discussion of the data structures that are important for calculations and exporting of information. After that, each step of the pipeline will be explained in detail with pseudo code. Additionally, the parameters and settings the user can control will be listed and their influence on the calculation elaborated on.

4.1 General Structure

As described in chapter 3.2 the algorithm follows a pipeline approach. This means that it is divided into different steps where each step builds on top of the previous one and uses the generated data up to this point. An overview of the order of the methods in this process is provided in algorithm 1.

Algorithm 1 General Structure

```
function DefineOceanRegion
    generate the coastline and the underwater terrain in the ocean region
function DefineRegions
    divide the land mass into different regions depending on user input
function GenerateRivers
    generate river network with lakes based on regions
function GenerateHeightMap
    calculate land mass between water bodies
function Visualize
    visualize generated terrain inside the Unity editor
```

In section 3.2, the decision was made to base the terrain generation on water bodies first. As a result these items are generated before calculating the heightmap. This process begins with the definition of the coastline where the landmass is separated from the ocean region based on different user settings. After the creation of the general shape of the map, the landmass has to be divided into different regions which are responsible for the way rivers and lakes will originate there afterwards. This gives the user control over the general placement of mountains and deserts. Afterwards the river networks can then be generated in a third step. Here a map containing the flow directions of each cell is generated and then river sources are placed depending on regional settings. The rivers can then be followed downwards from the source until the ocean is reached, meanwhile generating lakes on the way. Here the information from the previous step is used to avoid generating rivers in dry regions and also to shape the rivers depending on the regions. Rivers will also grow in strength when they combine and along the distance they flow. To finish the terrain a heightmap is generated by growing mountains and hills between the water bodies in a fourth step. This is also based on the regions from step two. The last step is then the visualization of the produced terrain. This does not change the generated information anymore but is used to represent the results of the algorithm for the user. It generates a grid mesh and displaces the points in y-direction corresponding to the heightmap generated from the algorithm. Finally, a texture can be applied to display the different information and to illustrate the terrain to the user. All the different steps are explained in more detail in 4.2.

4.1.1 Data Structures

As previously discussed, the way of representing the terrain for this algorithm is to use heightmaps. To do this the height information is stored in a 2-dimensional float array. On top of the height the algorithm needs to store some additional information per cell. This includes the information of the type of terrain or the strength of a river both of which represent data that may be interesting to be exported with the heightmap for further uses, but also some information that is only relevant for the calculation process, for instance, the distance to a region border. This supplementary data is also stored in 2-dimensional array of different data types. Information that should be exported will be stored as image data.

4.2 Implementation Details

There are a few parameters that do not belong to a specific step. Those are presented in table 4.1. These parameters are treated separately because they are used in multiple steps or represent more global settings that are not tied to a specific step.

Parameter	Data Type	Description
size	Integer	Size of the calculation grid in each dimension
seed	Integer	Seed for the pseudo random number generator
worldWidth	Integer	Real world scale of border of the generated terrain in meters
worldHeight	Range	The minimal and maximal real world height of the generated terrain in meters

Table 4.1 Overview of the parameters for the general parameters used in all of the steps.

The size of the calculation grid does not influence the size of the generated mesh but only the resolution of the generated data. Because it refers to the number of cells in the grid on each dimension, the computational cost increases quadratically. Therefore it is recommendable to do multiple runs on smaller grids for testing parameter settings and to do only the final generation on a very large grid. More details about the complexity and efficiency will be presented in chapter 5.1.1. The seed of the pseudo random function influences all the steps, because randomness is an important factor in the whole procedure and is mostly used in the form of noise maps (refer to chapter 2.1.3). The world width and height do not influence the size of the mesh representing the terrain in the visualization step, but exist to give the user the possibility to define exactly what the measurements of the produced terrain should be. The effect can be seen in the ratio between width and height of the mesh.

4.2.1 Ocean Border

To begin the procedure of generating the terrain, the coastline between ocean and landmass has to be defined first. To give the user the option to shape the terrain for their needs, this step deliberately provides a lot of manual control. The user can place markers on the map and the program then generates a closed curve out of the location and rotation information of these points. The markers are game objects inside Unity which contain a component that collects the data about the markers and sends it to the main object with the terrain generation component. The position of the objects is read in relation to the main object. The order of the marker objects in the hierarchy also defines the order in which the points are evaluated to calculate the curve. It also contains a strength value for each point, which affects the influence of the rotation of the marker. An example of this effect is displayed in figure 4.2. This strength value is not a global parameter but is set individually for each marker.

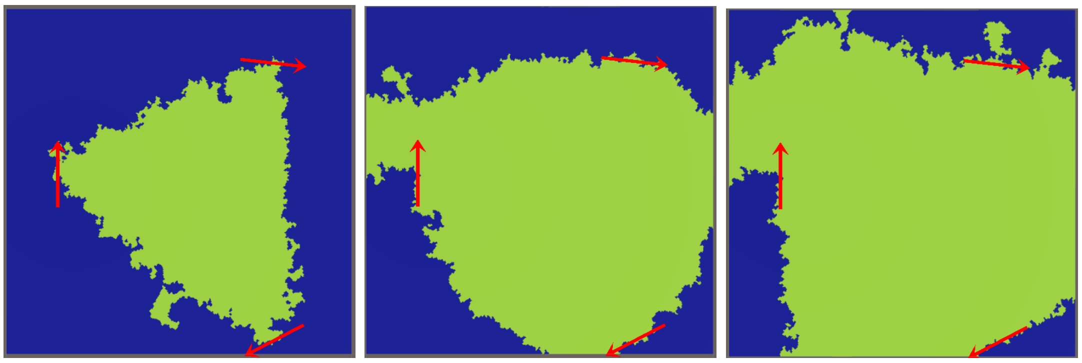


Figure 4.2 Effect of the influence parameter of the ocean spline points (represented by the arrows) on the general shape of the ocean border. The influence values from left to right are 100, 500, 1000.

The inside of the curve will then be classified as landmass, while the rest is set to be ocean. The curve is calculated as polynomials of third degree between the marker points.

Algorithm 2 Ocean border

```
function DefineOceanBorder
    find ocean point markers and generate curve from them
    place $voronoiPointsFirstIteration$ random points on grid and add them to queue $q$
    for all points $p$ do
        if $p$ is inside curve then
            set cell as ocean
        else
            set cell as land
    while $q$ not empty do
        point $p \leftarrow q$.pop()
        for all not searched neighbors $n$ of $p$ do
            classify $n$ the same as $p$
            add $n$ to $q$
            mark $n$ as searched
    mark points at border between ocean and land as coast
```

The issue is that a smooth curve does not represent a realistic coastline especially for a large map. To improve the borders, they are randomised afterwards in two refinement steps. This refinement is done with the use of Voronoi diagrams as described in 2.1.2. A number of random Voronoi sites are placed on the map and then the Voronoi regions are calculated for each site by assigning to it all points that are closer to that site than to any other. Because operations are done on a grid, the metric used here is the Manhattan distance. Then the whole region is classified to be either ocean or land depending on whether the corresponding site is inside or outside the curve. This step ensures that the general layout defined by the curve is retained but the actual border is randomised. The number of sites that are placed initially is defined by a parameter set by the user. The more points are generated the closer the shape resembles the curve. The effect of different amounts of points placed for this step can be seen in figure 4.3.

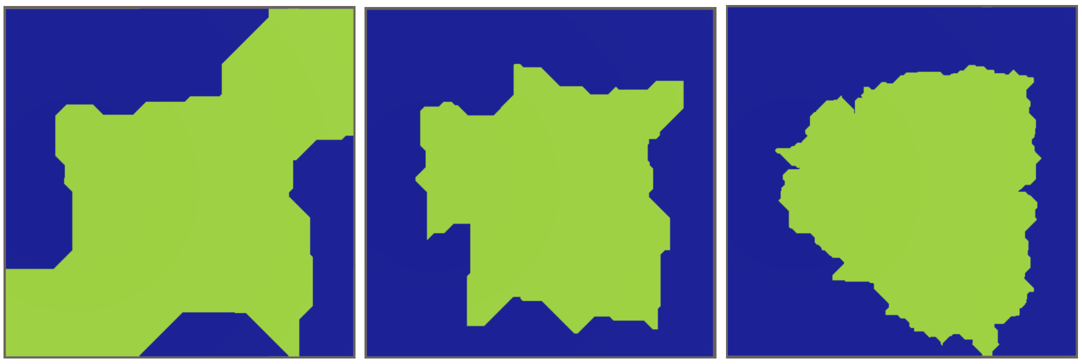


Figure 4.3 Effect of the number of Voronoi points in the first iteration of the ocean border. Here the second iteration was not performed. Number of Voronoi points from left to right are 20, 50, 500. The higher the number of points the closer the border resembles a smooth curve, but the less variation is produced.

This first refinement is a good way to define a rough outline but can produce some straight lines and is also not enough for finer structures. Because of that, an optional second refinement can be performed to generate a more realistic coastline. There are two different options the user can choose from. The first is to repeat the first refinement with sites placed around the previously defined border and using the generated classification instead of the curve. The second option is to also place the sites around the border but use a priority queue based on noise for the growth of the regions. The advantage of the second possibility is that it produces more variations and can also lead to small islands around the coast. Generating islands is not possible with the first approach. The parameters for the noise function used have an influence on how the border will look in the end. A higher frequency makes finer structures, while increasing the number of octaves allows for a differentiation between larger and smaller features. The disadvantage of this procedure is that the process is slower due to the use of a priority queue. A decision here must be made between performance and realism. To allow for more optimization in speed for different runs of the algorithm and for viewing the larger structure of the coastline, the second iteration can be disabled completely. A comparison between the different options for the second iteration is presented in figure 4.4. A summary of the parameters available to the user and their functionality can be viewed in table 4.5.

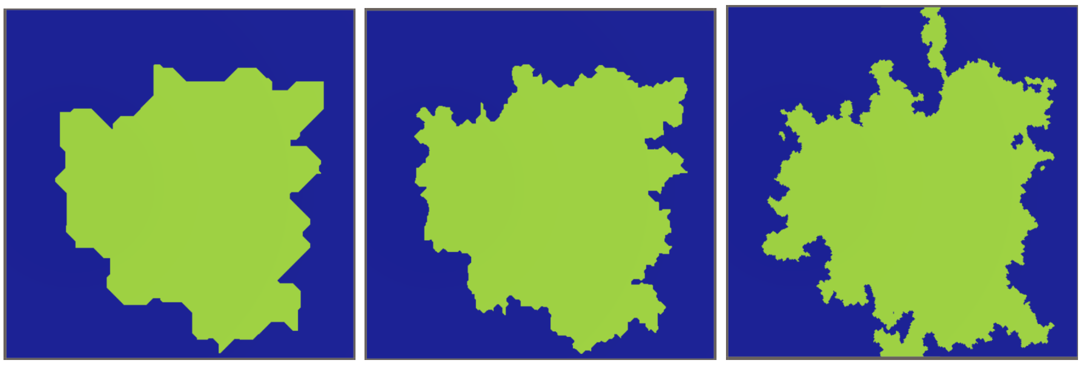


Figure 4.4 Effect on the generation of the ocean border depending on settings for second iteration. Left: Ocean border generation without second iteration, middle: Ocean border generation with second iteration without using noise, right: Ocean border generation with second iteration using noise.

Parameter	Data Type	Description
voronoiPointsFirstIteration	Integer	Number of Voronoi sites placed for the first refinement iteration
doSecondIteration	Boolean	Toggle for the execution of the second iteration
voronoiPointsSecondIteration	Integer	Number of Voronoi sites placed for the second refinement iteration
maxDistFromBorder	Integer	Range around the border of first iteration in which points for second iteration are placed
useNoiseInSecondIteration	Boolean	Toggles the mode in which the second iteration is performed
NoiseOctaves	Integer	Number of octaves for noise map of second iteration if noise is used

Table 4.5 Overview of the parameters for the generation of the ocean border which are available for the user

4.2.2 Regions

After the separation of ocean and land is completed, it is followed by the second important part of the pipeline. The landmass will now be divided up into regions of different types of terrain. This step is rather a preparation for what is coming later than actually generating terrain itself. The types of terrain that are available are flatland, mountain and desert. The distribution is also a manual process done by the user with markers on the map as before for the ocean border. These markers again are game objects in Unity with a component for the region parameter. Those parameters and the position of the object in relation to the terrain generator object are used in this step. For each marker the user has the possibility to set the type of terrain (mountain and desert), the range of the marker and parameters for the randomness of the region border. Every land cell that is not contained in a mountain or desert region is automatically assigned to flatland. The priority of overlapping regions depends on the order of markers in the Unity scene hierarchy. After setting the points, the algorithm will then proceed to iterate through the map to assign the corresponding region to each cell while skipping cells that have already been classified as ocean or shore. These regions are the base for generating rivers and terrain later in the process. The pseudo code for this step is shown in algorithm 3. Figure 4.6 shows an example for the regions generated with different settings for the markers.

Algorithm 3 Regions Separation

```
function DefineRegions
    user input
    for all regions do                ▷ order depends on sibling order inside Unity Editor
        calculate region border based on set radius and noise parameters
    for all cells marked as land do
        for all regions do
            if cell is in region then
                set cell as corresponding region
                break
```

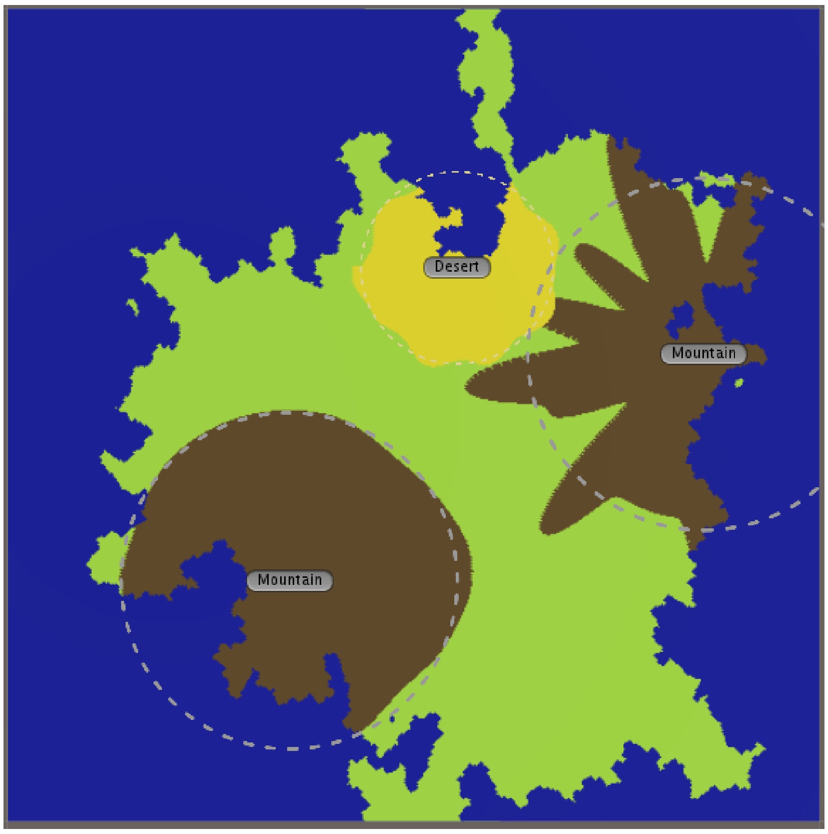


Figure 4.6 Effect of region point parameters region type, range, edge noise strength and edge noise frequency. Region points are represented by dotted circles, where the size of the circle shows its range.
Left: Type = mountain, range = 0.2, noise strength = 0.05, noise frequency = 0.2,
middle: Type = desert, range = 0.1, noise strength = 0.05, noise frequency = 0.3,
right: Type = mountain, range = 0.2, noise strength = 0.5, noise frequency = 0.5.

4.2.3 River Networks and Lakes

The next step in the pipeline is the generation of the river network. This is one of the most important steps in the whole process, as it is the base of the heightmap generation and a big focus of this thesis. When analysing naturally generated terrain, the landscape can be divided in catchment areas (refer to chapter 2.2.1). These are the areas where water (for example from rain) is collected. They are often divided by mountains or hills because the water flows in different directions from the peak. This means that by artificially generating those catchment areas, we know afterwards where we can place hills and mountains. The procedure starts by placing down a several possible outlets around the coast and also around mountain regions. The separation between those outlets happens so that the rivers can be generated differently based on terrain type. The number of outlets can be set by the user separately for each region type. An overview of all the different parameters used in this step is shown in table 4.7.

Parameter	Data Type	Description
flatlandOutlets	Integer	Number of outlets placed around the flatland coast
mountainOutlands	Integer	Number of outlets placed around mountain regions
flatlandSources	Integer	Number of river sources placed in flatland regions
mountainSources	Integer	Number of river sources placed in mountain regions
riverGrowth	Float	The strength a river gains in each step
calcRiverWidth	Boolean	Decides if width of rivers should be calculated based on their strength
riverMaxWidth	Integer	The maximal width of rivers in pixels
driedRiverChance	float	Chance for each river to only generate dry riverbed

Table 4.7 Overview of the parameters for the river generation available to the user

Placing outlets around desert regions is excluded in this process. To generate the catchment area, the algorithm then proceeds to calculate the flow directions of the water for each cell randomly and stores them in a flow map. By doing this, it procedurally connects all cells from mountain and flatland cells in a randomly connected flow network. This process also avoids desert regions to assure that they stay dry when placing the rivers. The number of outlets influences the shape of the river networks. The fewer outlets are generated, the higher the branching factor in the networks will be. This effect can be observed exemplarily in figure 4.8.

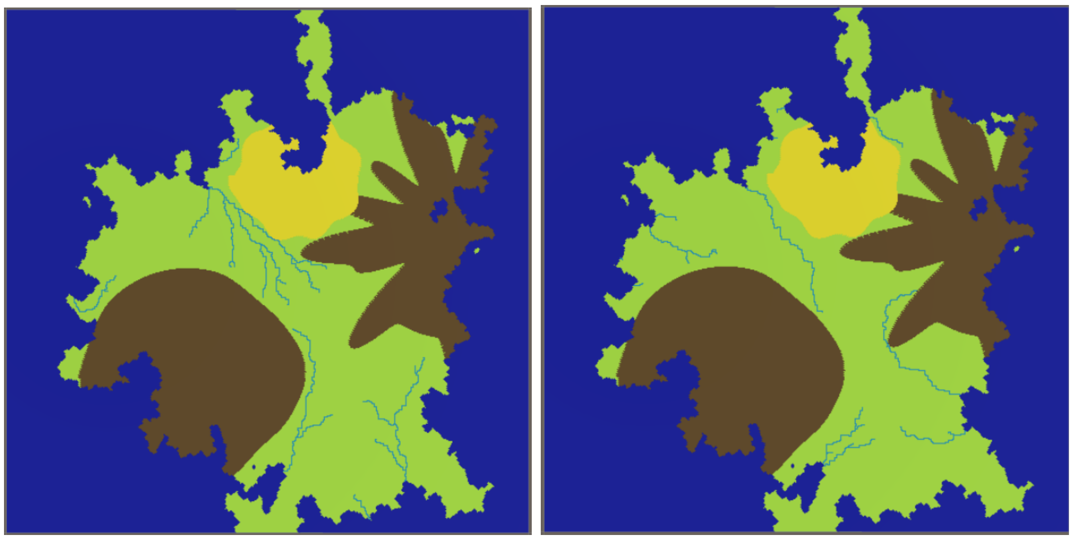


Figure 4.8 Effect of number of outlets in the catchment generation on the river network generation. For clarification purposes, rivers are only generated in the flatland region. The more outlets are generated the lower is the number of connected rivers. The picture on the left has 10 outlets and 20 sources, the picture on the right has 1000 outlets and 20 rivers. Using a certain number of outlets for the catchment generation does not necessarily result in the same number of actual outlets on the map.

Following these preparations, the algorithm can now proceed to generate actual rivers. A possible approach for this would be to calculate the amount of water that would flow through each cell as mentioned in section 3.1.2. This would be done by placing water evenly on the grid and then following each water unit down in the previously generated flow directions and adding up how many water units traverse each cell. Rivers can then be placed in every cell that has an amount of water higher than a specified threshold. This method works but leads to a relatively even distribution of river sources. The method which was used instead in this algorithm is to place the river sources randomly on the map and then follow the flow direction map until the ocean is reached. This yields the advantage of a more random distribution of sources. Furthermore, it is possible to calculate the strength of the river and save it to another map. The amount of river sources placed is also controlled by the user and can be set separately for mountain and flatland. They can also set the chance for a river to generate as a dried riverbed instead of a normal river. Each river adds a certain amount of strength to a cell when flowing through it which leads to combining the strength of two river arms when they join. Depending on the strength it is then possible to calculate the width of the river and assign more cells to it. This width calculation can be skipped by setting a parameter in the user setting. It is also possible to set a maximum width for rivers. This is useful in the case of calculation grids with lower resolution where one cell cover a larger area of the map. For each step of the river generation, there is a chance to generate a lake. In order to avoid overfilling the map with too many lakes and also to give control to the user, it is possible to set a maximum amount of lakes generated for mountain and flatland regions. In addition, the lake generation is limited by one lake per river source.

Algorithm 4 River Generation

```
function DefineRivers
    set $flatlandOutlets$ random points at border between flatland and ocean and add them to queue $q$
    while $q$ not empty do
        point $p \leftarrow q$.pop()
        connect a random not yet connected neighbour $n$ to $p$ by setting the flow direction from $n$ to $p$
        $heightMap(n) \leftarrow heightMap(p)$ + growth factor
        add $n$ to $q$
    set $mountainOutlets$ random points at border between mountain and flatland/ocean and add them to $q$
    for all $mountainOutlets$ $o$ do
        if $o$ is next to flatland then
            connect $o$ to random flatland neighbour $n$
            $heightMap(o) \leftarrow heightMap(n)$ + growth factor
        else
            $heightMap(o) \leftarrow 0$
    while not all mountain cells are connected do
        point $p \leftarrow q$.pop()
        connect a random not yet connected neighbour $n$ to $p$ by setting the flow direction from $n$ to $p$
        $heightMap(n) \leftarrow heightMap(p)$ + growth factor
        add $n$ to $q$
    set $mountainSources$ random sources points on mountain cells
    set $riverSources$ random sources points on mountain cells
    for all sources $s$ do
        roll if this river should be generated as dried
        point $p \leftarrow s$
        while $p$ is not ocean cell do
            mark $p$ as (dried) river
            $riverStrengthMap(p) \leftarrow riverStrengthMap(p) + riverGrowthFactor$
            $p \leftarrow flowDirectionMap(p)$
            if number of lakes<$flatlandLakes/mountainLakes$ and random condition then
                GenerateLake($p$, $riverStrengthMap(p)$)
    for all sources $s$ do
        follow $flowdirectionMap$ and extend width for each cell respecting $riverMaxWidth$
```

The function which produces lakes takes the strength of the river cell it is generated from into account. The higher the strength the more likely it becomes for the lake to be bigger. In addition, the lake size is controlled by a maximum and minimum size that can be set by the user. The pseudo code for the lake generation is shown in algorithm 5.

Algorithm 5 Lake Generation

```
function DefineLake(position $p$, river strength $s$)
    define lake border by $s$, $lakeNoiseStrength$, $lakeNoiseFrequency$, $lakeNoiseOctaves$
    add $p$ to queue $q$
    while $q$ is not empty do
        point $p \leftarrow q$.pop()
        for all neighbours $n$ of $p$ do
            if $n$ has not been checked yet and inside lake borders then
                $q$.push($n$)
                mark $p$ as lake
                mark $n$ as checked
```

To produce lakes that vary in shape, a noise function is used on the border of the lake. The user can set the frequency, the strength and the number of octaves of the noise function. An overview of the parameters for the lake generation can be seen in table 4.9. All the cells inside the border are then classified as lake. After generating a lake, the river generation continues.

Parameter	**Description**	
flatlandLakes	Integer	Maximum number of lakes generated in flatland
mountainLakes	Integer	Maximum number of lakes generated in mountains
lakeNoiseStrength	Float	Amplitude of the noise function used for lake outline variation
lakeNoiseFrequency	Float	The frequency of the noise function used for lake outline variation
lakeNoiseOctaves	Integer	Number of octaves for the noise function used for lake outline variation
lakeDimensions	Range	The minimal and maximal possible radius of lakes

Table 4.9 Overview of the parameters for the lake generation available to the user

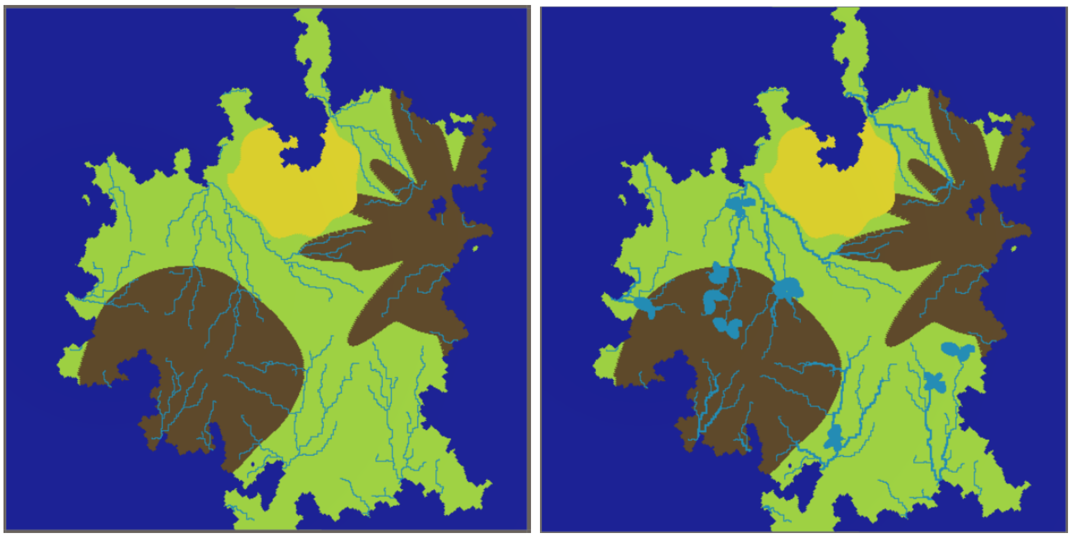


Figure 4.10 Effect of calculating river width and the placement of lakes. Left: no river width, no lakes. Right: max river width = 2, max. lake number = 10.

4.2.4 Heightmap

After finishing the river and lake generation, the next step is to generate the heightmap. This is the last part of the actual terrain generation process. For the latter, all the information that was produced in the steps before is needed. The terrain will be grown beginning from the ocean, rivers and lakes. To do this, all those cells are added to a priority queue with the priority being the height of the cells generated as a preparation in the river network generation. When a point is removed from the queue, all the neighbours that have not been traversed yet will receive a new height value calculated by adding a growth factor to the height of the point. This process is illustrated in algorithm 6.

Algorithm 6 Heightmap Generation

```
function GenerateHeightMap
    add all coast, river and lake cells to priority queue $q$
    while $q$ is not empty do
        point $p \leftarrow q$.pop()
        for all neighbours $n$ of $p$ do
            if $n$ has not been checked yet and not been classified as ocean then
                $heightMap(n) \leftarrow heightMap(current)$ + growth factor
                $q$.push($n$)
                mark $n$ as checked
```

The growth factor there is calculated by taking in account the terrain growth factor of the region the cell is in, and a random value obtained from noise functions also depending on the regions. The factor is also interpolated between the different region types to allow for a smoother transition between them. The parameters available to the user can be seen in table 4.11.

Parameter	Data Type	Description
riverSlope	Float	The rate at which the terrain grows in river paths
mountainSlope	Float	The rate at which the terrain grows in mountain regions
flatLandSlope	Float	The rate at which the terrain grows in flatland regions
mountainNoiseFrequency	Float	The frequency of the noise function used in the mountain region heightmap generation
mountainNoiseStrength	Float	The strength of the noise function used in the mountain region heightmap generation
mountainNoiseOctaves	Integer	The number of octaves of the noise function used in the mountain region heightmap generation
flatlandNoiseFrequency	Float	The frequency of the noise function used in the flatland region heightmap generation
flatlandNoiseStrength	Float	The strength of the noise function used in the flatland region heightmap generation
flatlandNoiseOctaves	Integer	The number of octaves of the noise function used in the flatland region heightmap generation

Table 4.11 Overview of the parameters for the generation of the ocean heightmap which are available for the user

The effect of different growth factors for the regions are displayed in figure 4.12. The growth factor itself does not change the final height of the terrain as it is difficult to know exactly what the result would be. To guarantee the final height is controllable the heightmap is scaled to fit the global world width and height parameters set by the user.

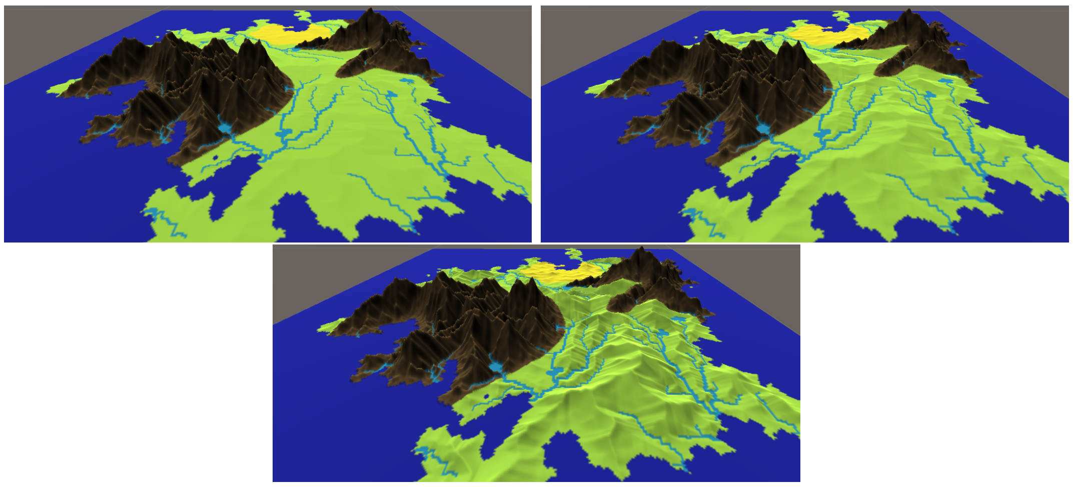


Figure 4.12 Effect of flatland slope on height generation of the map. The lower the flatland slope in comparison to the mountain slope, the higher the hills between rivers.
Top left: Flatland slope = 0.05, mountain slope = 2.
Top right: Flatland slope = 0.2, mountain slope = 2.
Bottom: Flatland slope = 0.5, mountain slope = 2

4.2.5 Visual Representation and Output

The last step of the pipeline is the visual representation of the generated terrain. This is not part of the actual terrain generation but serves an important role in giving the user information about the results of the previous steps. This allows for evaluating the results and changing the parameters if needed. The visualization includes two steps. The first is to generate a mesh to represent the heightmap generated before. Each cell in the calculation grid is represented by one vertex in the mesh. The mesh needs to be generated by calculating the triangles between the vertices and the uv coordinates to be able to place a texture on it. The vertices are placed at the corresponding height of the heightmap. It is also possible to generate a completely flat mesh. The latter slightly speeds up the last part if the map has a really high resolution. This can be useful, if the focus is on one of the earlier pipeline steps, such as the ocean border generation, where the height information has not become relevant yet.

Parameter	**Description**
textureMode	List of texturing options to put on the terrain mesh
colors	Colour collection used for generating the textures

Table 4.13 Overview of the parameters for the visualization process

The procedure is then completed by generating textures which can be placed on the mesh. There are different texture options the user can choose from. The normal texture displays a gradient on terrain points that is dependent on the height. Other options include a region mode, where a separate colour for each region is displayed and a height mode where the normalized heightmap is chosen as a texture. Examples of the different texturing options are shown in figure 4.14. There exists also an option that lets the user choose the colours for the different region types. Furthermore, the region and the heightmap texture are exported to allow for using the generated terrain in other applications.

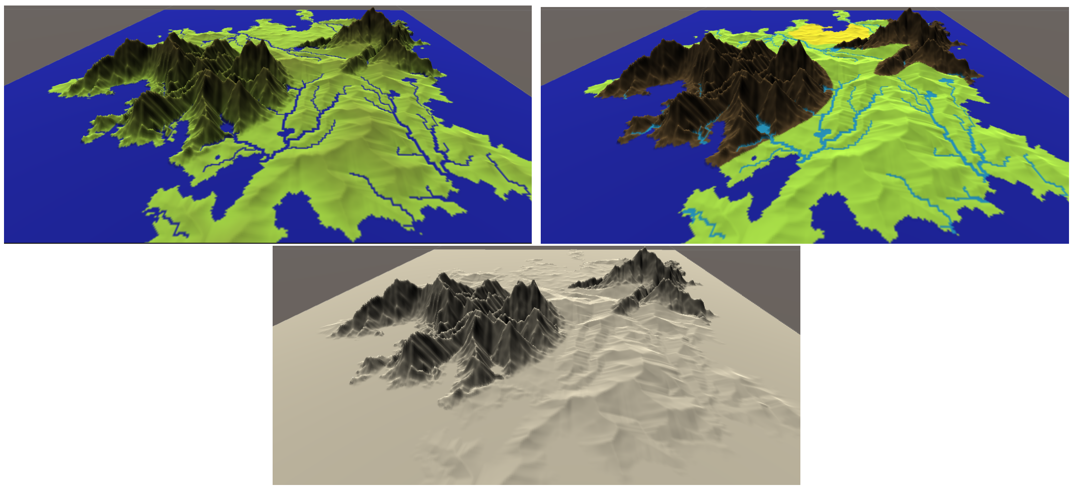


Figure 4.14 Highlighting different aspects of the algorithm using different texture modes. Top-left: Normal mode (blue for all types of water bodies, gradient for landmass based on height), top-right: Region mode (separate colour for each region: Dark blue for ocean, light blue for rivers and lakes, brown for mountains, yellow for desert, green for flatland), bottom: Height mode (black and white gradient based on the normalized height value).

Chapter 5

Evaluation

This chapter deals with the evaluation of the implemented algorithm. First, a theoretical complexity analysis will be conducted. This is divided in two parts, the time and space complexity. Afterwards, the results of runtime measurements of the implementation will be presented and analysed. In another section, the quality and realism of landscapes generated with the algorithm will be evaluated. The chapter then ends with a last section in which visual results will be compared to reality. A comparison to other terrain generation algorithms is difficult because the field of procedural terrain generation is very large. Most algorithms differ in the approaches they use and thus, comparing them to this approach sensible.

5.1 Complexity Analysis

5.1.1 Time Complexity

In this section, the algorithm described in chapter 4 will be analysed for its runtime complexity CX. This will be divided into the different steps of the pipeline analogously to the last chapter. For each step, a table introducing the relevant variables will be presented. Based on those variables, the complexity for the corresponding step will then be calculated.

Ocean Border

Variable	Description
n_c	Number of cells in the grid per dimension
n_m	Number of coastline markers placed on the map
n_{v1}	Number of Voronoi sites in the first iteration
n_{v2}	Number of Voronoi sites in the second iteration
C_{spline}	Cost for one spline calculation
C_{testP}	Cost for testing the position of a point in relation to curve
C_{testR}	Cost for testing the position of a point in second iteration
C_{class}	Cost to classify a cell
CX_c	Complexity of the coastline calculation

Table 5.1 Variables for the ocean border time complexity

With these variables the complexity of the coastline generation can be calculated by

$$\begin{aligned} CX_c &= n_m \cdot C_{spline} + n_v \cdot C_{testP} + n_c^2 \cdot C_{classify} \\ &= O(n_c^2). \end{aligned}$$

In the worst case, the number of Voronoi sites equals the number of cells in the grid, which means that then n_v is the same as n_c^2. The same is true for the second iteration. Even when using the option that uses the priority queue, the complexity for inserting and removing elements from that is in $O(\log n_c^2)$ and thus can be neglected.

Regions

Variable	Description
n_c	Number of cells in the grid per dimension
n_r	Number of region points placed
C_d	Cost to check the distance
CX_r	Complexity of the region separation

Table 5.2 Variables for the region time complexity

In the region separation, each cell of the grid needs to be checked once. In the worst case, all cells are non-ocean cells so that for each cell the distance to the region points must be checked. This leads to a complexity of

$$\begin{aligned} CX_r &= n_c^2 \cdot n_r \cdot C_d \\ &= O(n_c^2). \end{aligned}$$

Rivers

Variable	Description
n_c	Number of cells in the grid per dimension
n_s	Number of river sources
w	Maximal river width
n_l	Number of lakes
d_l	Maximum lake diameter
$CX_{catchment}$	Complexity of the catchment generation
CX_{rivers}	Complexity of the river generation
CX_{lakes}	Complexity of the lake generation
CX_r	Complexity of the river generation

Table 5.3 Variables for the rivers and lakes time complexity

In the worst case with reference to river and lake generation, the whole map is classified as flatland or mountain. In this case the catchment must be calculated for the whole grid. A second issue in the worst case scenario is that each river flows through each cell of the grid. In normal scenarios the number of river sources should be negligible in comparison to the size of the grid, but it would be possible to raise it to the number of cells. Sharing the same argument for the lake generation, this leads to a complexity of

$$\begin{aligned} CX_r &= CX_{catchment} + CX_{rivers} + CX_{lakes} \\ &= n_c^2 + n_s \cdot n_c^2 \cdot w + n_l \cdot d_l^2 \\ &= O(n_c^2 \cdot n_s \cdot n_l). \end{aligned}$$

Heightmap

Variable	Description
n_c	Number of cells in the grid per dimension
C_h	Cost for calculating the height of a cell
C_a	Cost for the height adjustment of a cell to fit real life world height span
CX_T	Complexity of the terrain generation

Table 5.4 Variables for the heightmap time complexity

The complexity for the terrain height calculation is $O(n_c^2)$. In the worst case, the algorithm must visit each cell of the grid once. For each cell it is required to calculate the height based on the height of the neighbour cell and a random factor. In addition, the grid must be traversed again to assure that the maximum and minimum height fit the user specification. Thus, the complexity can be calculated as following

$$\begin{aligned} CX_T &= n_c^2 \cdot C_h + n_c^2 \cdot C_a \\ &= O(n_c^2). \end{aligned}$$

Visuals

Variable	Description
n_c	Number of cells in the grid per dimension
C_h	Cost for calculating coordinates of a vertex of the mesh
C_t	Cost for calculating a triangle of the mesh
CX_M	Complexity of the mesh generation

Table 5.5 Variables for the visualization time complexity

To allow Unity to visualize the terrain a 3-dimensional mesh must be constructed from the results of the previous steps. Each cell of the grid is represented by one vertex of the visualization mesh. For each vertex the coordinates must be determined to show the whole scenery in a 1 by 1 square with the corresponding heights. In addition, a square grid mesh of side length n consists of $(n-1)^2$ squares. This translates to two times the number of triangles. Therefore, it follows that the complexity of the mesh generation can be described as

$$\begin{aligned} CX_M &= n_c^2 \cdot C_h + 2 \cdot (n_c - 1)^2 \cdot C_t \\ &= O(n_c^2). \end{aligned}$$

Finally, the time complexity of the whole pipeline CX can be calculated by combining the complexities of all the different steps by

$$\begin{aligned} CX &= CX_C + CX_R + CX_R + CX_T + CX_M \\ &= O(n_c^2 \cdot n_s \cdot n_l). \end{aligned}$$

In conclusion, the time complexity depends linearly on the number of river sources and the number of lakes and quadratically on the number of cells in the grid per dimension.

5.1.2 Space Complexity

Variable	Description
n_c	Number of cells in the grid per dimension
c	Constant space of parameters
CX_s	Space complexity of the algorithm

Table 5.6 Variables for the space complexity

To run the algorithm some data must be stored on the system. The size of the data can vary depending on the users settings. The only parameter that has an influence on this is the number n_c of cells in the calculation grid per dimension. Some data has to be stored permanently while other data is only kept for specific parts of the algorithm's runtime. The permanent data includes two dimensional arrays of size n_c^2 to store terrain information. This includes maps for storing the terrain type, water flow direction, river strength, terrain height and interpolation distance. Moreover a texture must be saved.
While the algorithm is running, multiple queues of a maximum size of n_c^2 have to be saved temporarily. At any time of the algorithm there exist at most four of those queues. Furthermore, each step of the algorithm that traverses the map must save information about which cells have been visited already. This is done in a 2-dimensional array of size n_c^2. The parameters that have an influence on how the algorithm is performed need a fixed amount of space no matter what the size of the grid. This means that they are negligible for the space complexity calculation.
Considering all these facts, the space complexity CX_s is

$$\begin{aligned} CX_s &= 5 \cdot n_c^2 + n_c^2 + 4 \cdot n_c^2 + n_c^2 + c \\ &= O(n_c^2). \end{aligned}$$

5.2 Performance Evaluation

After the theoretical considerations, it is important to measure the performance of the algorithm. As discussed in chapter 5.1.1, the runtime of the algorithm depends mostly only on the size of the grid n_c. Therefore, for $n_c = 256, 512, 1024, 2048$ the times for each part of the algorithm are measured in milliseconds over 20 runs. To give more detailed information for the resulting data besides the average, also the minimum, maximum and median value are stated. Because the section of the river and lake generation also depends on the parameters of the number of lakes and river sources, additional data are acquired where those variables

were changed and the grid size was kept constant.
All measurement were taken on a 64-bit Windows 10 operating system with an AMD FX(tm)-6300 Six-Core processor with 3.5GHz, 16Gb of RAM and a GeForce GTX 960 graphics processor. Times are measured in Unity with stopwatches from the System.Diagnostic namespace.

5.2.1 Ocean Borders

Table 5.7 shows the calculation times of the ocean border calculation. The times are split into the calculation times for the first and the second iteration. In addition, the table provides the information for both variants of the ocean border calculation (with or without using noise). Values in parentheses show the results for the option where noise is not used. The average times for both variants are plotted in figures 5.8 and 5.9. As expected after the complexity analysis, the graphs show a quadratic growth. The iteration of the variant which uses noise takes approximately double the time of the other variant. This is not caused by the noise, but by the use of the priority queue. The noise is only used as the priority for the elements of the queue. In the other variant, a normal queue is sufficient, as it only implements a classic Voronoi diagram. The second iteration takes approximately three times as long as the first iteration. This is caused by the number of Voronoi points that are used. Because the second iteration is intended to be a refinement of the first, a high number of points is recommended to gain better results.

n_c		**Minimum**	**Maximum**	**Average**	**Median**
256	First iteration	19	30	21	21
	Second iteration	133(72)	154(87)	140(82)	138(83)
512	First iteration	80	93	86	86
	Second iteration	619(289)	678(398)	636(336)	631(335)
1024	First iteration	326	356	337	335
	Second iteration	2713(1145)	2915(1340)	2839(1298)	2850(1306)
2048	First iteration	1335	1496	1405	1398
	Second iteration	11878(4690)	13258(5892)	12389(4848)	12504(4740)

Table 5.7 Calculated minimum, maximum, average and median times in milliseconds measured for grid sizes n_c = 256, 512, 1024, 2048. For each size, the times are separated into the time for the first and second iteration. Numbers in parentheses are the times for the second iteration without using noise. Times are taken from 20 runs with the same settings.

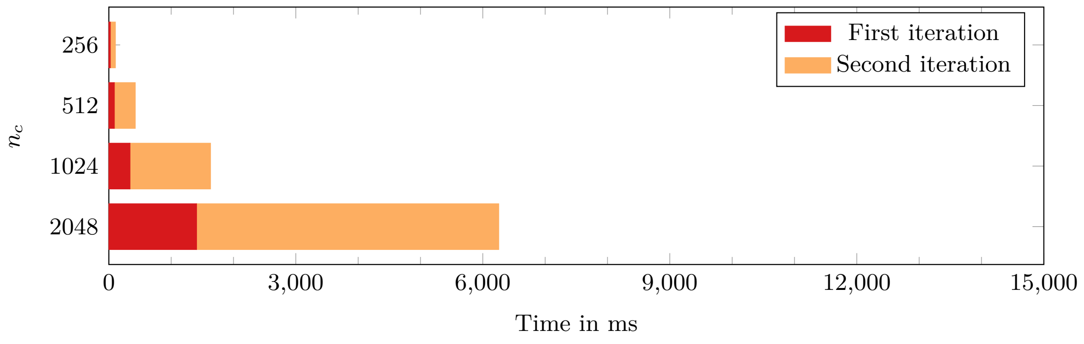


Figure 5.8 Plotted average times for first and second iteration of the ocean border generation against grid side length n_c = 256, 512, 1024, 2048. The second iteration is done without using noise. The data is taken from table 5.7.

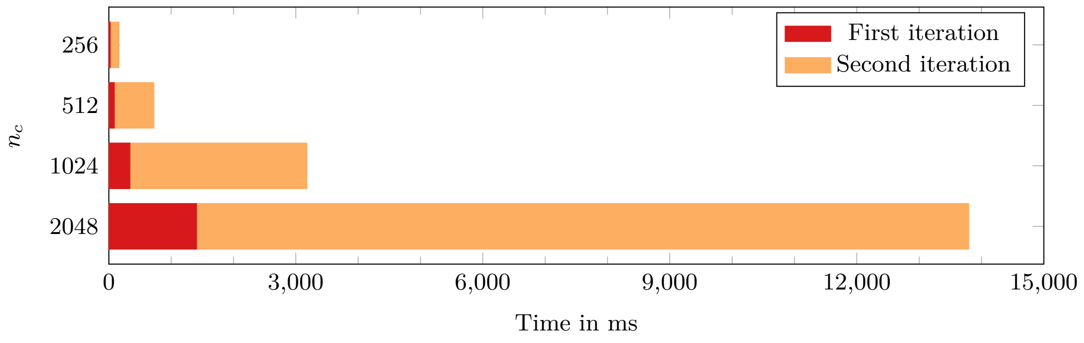


Figure 5.9 Plotted average times for first and second iteration of the ocean border generation against grid side length n_c = 256, 512, 1024, 2048. The second iteration is done with using noise. The data is taken from table 5.7.

5.2.2 Regions

The evaluation of the calculation times for the region separation is divided into two different parts. In the first part, the map is flooded with the different region types. In the second part the borders where different regions touch each other are extracted. The measured times for the steps are shown in table 5.10. Both steps are relatively fast as the map must be traversed only once and no complex calculations are needed per cells. As can be seen in figure 5.11 for resolutions up to 1024 cells per dimension, the times fit the expected quadratic curve. For 2048 cells per dimension the map sometimes generates a lot of small islands instead of one connected land mass. This leads to fewer cells being classified in the region filling, as all ocean cells are skipped. The border extraction is not affected by this because it traverses the whole map regardless of what region the cells belong to.

n_c		**Minimum**	**Maximum**	**Average**	**Median**
256	Fill regions	13	15	14	14
	Region borders	3	4	3	3
512	Fill regions	52	57	44	44
	Region borders	14	17	15	15
1024	Fill regions	130	161	138	136
	Region borders	38	43	40	40
2048	Fill regions	124	145	132	133
	Region borders	56	62	58	59

Table 5.10 Calculated minimum, maximum, average and median times in milliseconds measured for grid sizes $n_c = 256, 512, 1024, 2048$. For each size, the times are separated into the time for region filling and region border calculation. Times are taken from 20 runs with the same settings.

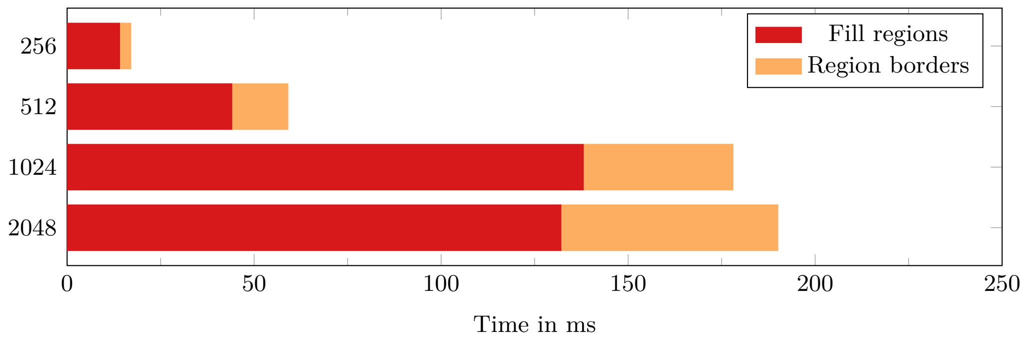


Figure 5.11 Plotted average times for region filling and calculating of region borders against grid side length $n_c = 256, 512, 1024, 2048$. The data is taken from table 5.10. Note that the time does not increase as expected. This is due to less landmass being generated for $n_c = 2048$ in comparison to lower values of n_c. This leads to lower calculation time for region separation.

5.2.3 Rivers and Lakes

The results of the river and lake generation pipeline step are displayed in table 5.12. This step is separated into three substeps. The first step involves the calculation of the flow direction map. The other two parts concern the river and lake generation. In figure 5.13 the results are plotted in a graph. It shows a quadratic growth in time when the grid length increases. Furthermore, it can be observed that the flow direction generation takes up almost all the time. This is because that part must traverse every cell that is not ocean or desert. The river and lake generation operates only on the cells that are classified as those types and thus has to traverse a significantly fewer number of cells. To be able to examine those two parts more closely, a separate measurement of calculation times was performed, where the grid size is fixed, and the number of total river sources and maximum number of lakes are varied. The results for this are displayed in table 5.14. It can be observed that the times do not increase when the maximum number of lakes exceeds the number of river sources. This is due to the fact that the implementation limits the amount of lakes generated by one per river in order not to overfill the map with lakes and to guarantee a better distribution.

n_c		**Minimum**	**Maximum**	**Average**	**Median**
256	Flow directions	29	33	31	31
	River generation	9	59	12	10
	Lake generation	5	6	5	5
512	Flow directions	171	193	178	177
	River generation	17	22	19	19
	Lake generation	5	8	6	6
1024	Flow directions	695	752	708	705
	River generation	23	33	26	24
	Lake generation	7	19	10	8
2048	Flow directions	13003	14226	13320	13178
	River generation	83	176	94	88
	Lake generation	8	33	11	8

Table 5.12 Calculated minimum, maximum, average and median times in milliseconds measured for grid sizes $n_c = 256, 512, 1024, 2048$. For each size the times are separated into the time for flow direction calculation, river generation and lake generation. Times are taken from 20 runs with the same settings.

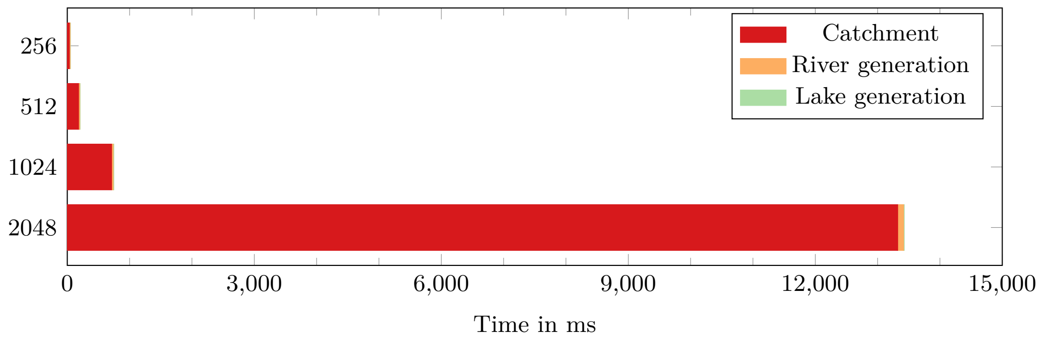


Figure 5.13 Plotted average times for flow direction calculation, river and lake generation against grid side length $n_c = 256, 512, 1024, 2048$. The data is taken from table 5.12.

n_r	n_l	**Minimum**	**Maximum**	**Average**	**Median**
10	10	192	3632	1871	1843
	100	168	3537	1861	1953
	1000	169	3542	1849	1823
100	10	189	3748	1992	2066
	100	251	5090	2628	2838
	1000	245	4995	2463	2736
1000	10	306	6686	3480	3649
	100	371	8053	4182	4377
	1000	715	15233	7364	8368

Table 5.14 Calculated minimum, maximum, average and median times in milliseconds measured for all combinations of number of river sources n_r = 10, 100, 1000 and maximum lake amounts n_l = 10, 100, 1000.

5.2.4 Heightmap

The process of the height generation is divided into two parts for this evaluation. The first one is the calculation of the distance map as a preparation step which is used for the interpolation between the different regions. The second part covers the actual heightmap generation. The times for both steps are shown in table 5.15. As can be seen in the visualization of these results in figure 5.16, the times resemble a quadratic curve as expected. The graphic also shows that most of the total calculation time for this step is used to generate the heightmap. The distance map calculation takes up only a small amount of the total time. For creating the distance map, the grid is traversed only once and only the parts of the landmass. The heightmap generation uses a priority queue to grow the terrain from the lowest point to the highest. This slows down the process considerably.

n_c		**Minimum**	**Maximum**	**Average**	**Median**
256	Distance map	9	10	10	10
	Heightmap generation	147	164	153	153
512	Distance map	36	40	37	37
	Heightmap generation	574	649	592	589
1024	Distance map	107	115	111	110
	Heightmap generation	1413	1540	1462	1454
2048	Distance map	623	705	660	651
	Heightmap generation	7687	8756	7980	7896

Table 5.15 Calculated minimum, maximum, average and median times in milliseconds measured for grid sizes n_c = 256, 512, 1024, 2048. For each size the times are separated into the time for the distance map calculation and heightmap generation. Times are taken from 20 runs with the same settings.

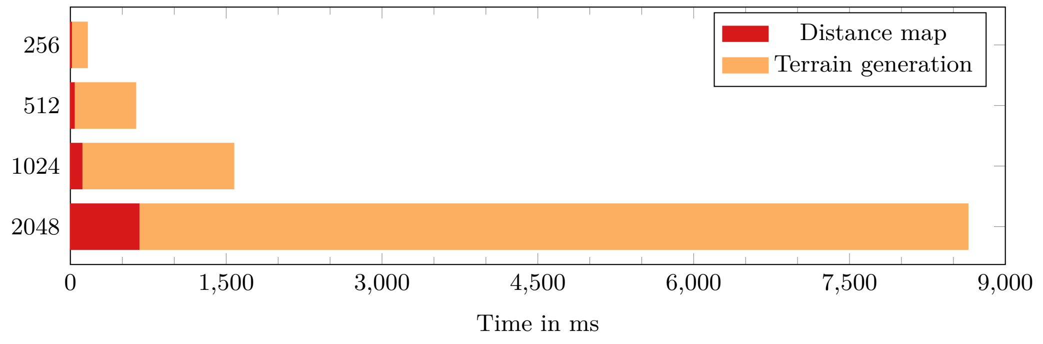


Figure 5.16 Plotted average times for distance map and terrain generation against grid side length $n_c = 256, 512, 1024, 2048$. The data is taken from table 5.15.

5.2.5 Visualization

The calculation times for the visualization divided into the mesh and texture generation are displayed in table 5.17. The times are plotted in figure 5.18. It shows a quadratic growth. The generation of the textures takes slightly longer than the mesh generation. Only one mesh must be calculated but more than one texture needs to be generated for exportation.

n_c		**Minimum**	**Maximum**	**Average**	**Median**
256	Mesh generation	18	41	23	22
	Texture generation	57	128	69	65
512	Mesh generation	79	100	85	83
	Texture generation	175	238	185	181
1024	Mesh generation	365	419	381	375
	Texture generation	630	813	662	653
2048	Mesh generation	1417	1872	1549	1542
	Texture generation	2697	3720	2885	2879

Table 5.17 Calculated minimum, maximum, average and median times in milliseconds measured for grid sizes $n_c = 256, 512, 1024, 2048$. For each size, the times are separated into the time for the mesh and texture generation. Times are taken from 20 runs with the same settings.

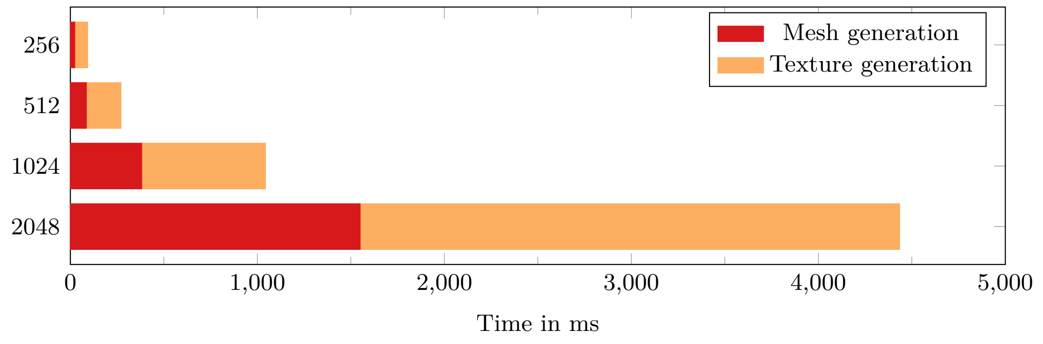


Figure 5.18 Plotted average times for mesh and texture generation against grid side length $n_c = 256, 512, 1024, 2048$. The data are taken from table 5.17.

5.2.6 Complete Algorithm

To complete the section of evaluating the calculation time results, the total times are presented in table 5.19. For each grid size, the times for the different steps are listed to provide the possibility to compare the different steps. These results are also plotted in figure 5.20. The graphic shows a quadratic curve as expected after the complexity analysis and the evaluation of the individual steps. It can be seen that the region separation takes a negligible amount of time in comparison to the other steps. The ocean border generation takes approximately a third to half of the total time. In these measurements the ocean border variant with noise in the second iteration was used to give a better representation of the more demanding options. Faster times can be achieved when disabling the use of noise or the second iteration completely as seen in section 5.2.1. The visualisation time could also be reduced by approximately half when activating the option of generating a flat mesh, as only the textures must be generated.

n_c		**Minimum**	**Maximum**	**Average**	**Median**
256	Ocean borders	138	154	144	139
	Regions	18	19	18	18
	Rivers and lakes	44	49	46	45
	Terrain	141	162	149	150
	Visualization	85	133	92	99
512	Ocean borders	622	706	644	641
	Regions	69	74	71	69
	Rivers and lakes	191	245	199	195
	Terrain	534	579	557	544
	Visualization	267	313	279	267
1024	Ocean borders	2757	2966	2832	2798
	Regions	177	193	182	177
	Rivers and lakes	697	755	714	704
	Terrain	1348	1478	1407	1374
	Visualization	1027	1136	1059	1029
2048	Ocean borders	12804	17555	13506	14230
	Regions	194	227	204	214
	Rivers and lakes	13050	16050	13549	14195
	Terrain	8251	10944	8662	8763
	Visualization	4486	5785	4779	5320

Table 5.19 Calculated minimum, maximum, average and median times in milliseconds measured for grid sizes $n_c = 256$, 512, 1024, 2048. For each size the times are separated into the time for the separate steps ocean border generation, region calculation, river and lake generation, terrain generation and visualization. Times are taken from 20 runs with the same settings.

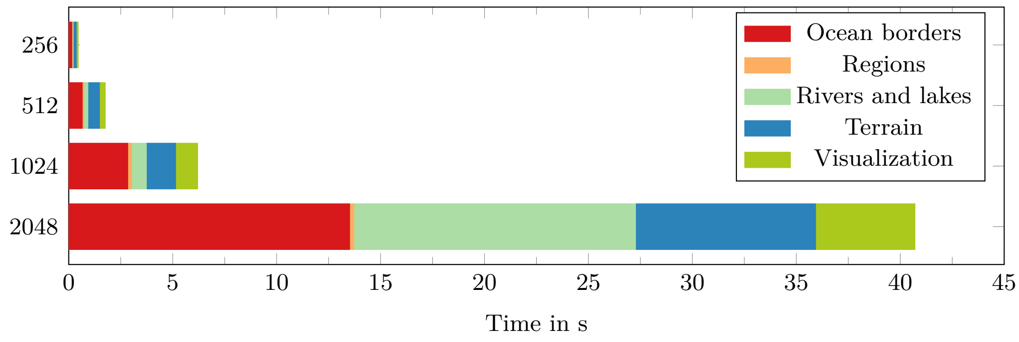


Figure 5.20 Plotted average times for the complete algorithm against grid side length $n_c = 256$, 512, 1024, 2048. The time is subdivided in the sections ocean border generation, region filling, lake and river generation, terrain generation and visualization. The data is taken from table 5.19.

5.3 Visual Evaluation

5.3.1 Results

In order to evaluate the quality of the terrain generated with the proposed algorithm, several landscapes were created with various settings. Some examples are shown in figure 5.21. As seen there, it is possible to produce a vast variety of shapes for the coastline. Single continents, as well as island groups, can be created by varying the number and position of ocean points. Furthermore, several different height profiles can be generated. It is possible to generate large scale maps, such as the examples in figure 5.21 but also results at smaller scales are an option as shown in figure 5.22.
In general, the generated results look realistic. Presumably, the majority of different demands on the produced landscapes can be satisfied.

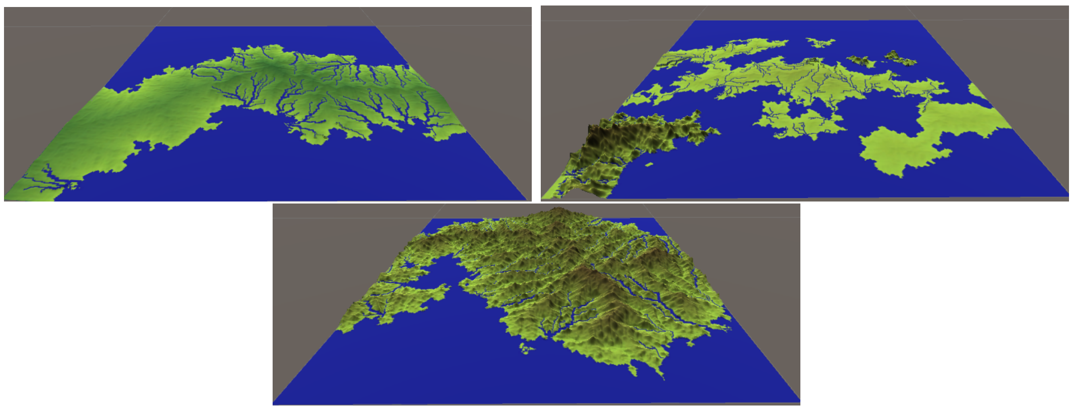


Figure 5.21 Examples for terrain generated with the algorithm of this thesis

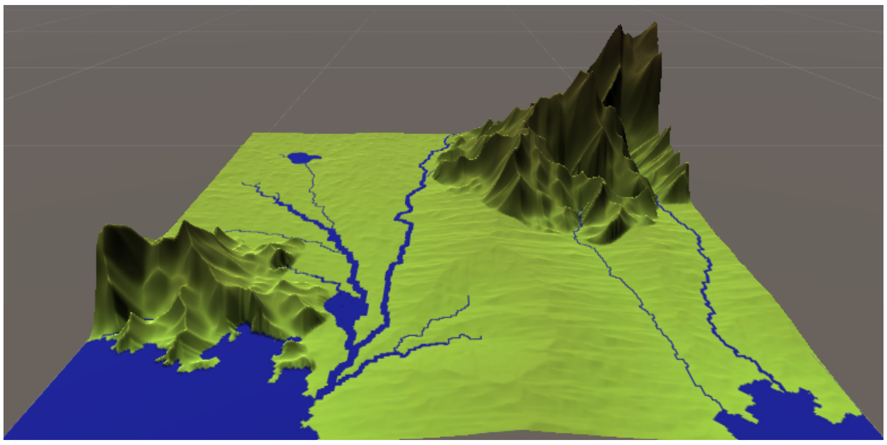


Figure 5.22 Example for a landscape at smaller scale

5.3.2 Comparison with Reality

After analysing the algorithm for its complexity and measuring the runtime efficiency, this section is now dedicated to considering the realism of the results. For that purpose a heightmap image from real terrain was taken and interpreted in the program to generate a 3-dimensional representation. The chosen terrain is a section of the Severo-Evensky District in Magadab Obkast in Russia (61.217039, 160.218362). Based on that, an attempt was made to replicate the real terrain as closely as possible with the algorithm of this thesis. Figure 5.23 shows the comparison between the mesh representations of both landscapes. As seen, it is possible to recreate a similar general shape of the coastline. Because the generation is heavily based on random components, it is impossible to generate a coastline that matches exactly. The mountain ranges are distributed with a good approximation of the reality. It has not been possible to find real world reference with satisfactory information about river networks. Therefore, the map was generated only using dried riverbeds. It is not possible to perfectly match the behaviour where terrain touches the world border. The real map is a part of a larger landscape and thus, rivers flow through the border. The terrain generation algorithm does not have information about the terrain outside the grid borders and thus cannot replicate this behaviour. However, the inland parts of the river networks were generated in a believable way. Even though the pathways of the riverbeds differ from the original directions, the individual parts of river networks have similar overall shapes. This can be observed, for instance, in the northern parts of the mountains in figure 5.23. In figure 5.24 the heightmaps of both terrains are displayed. Figure 5.25 shows the setting used for this attempt to recreate the real terrain. In general, it was possible to create a good approximation of the the real terrain.

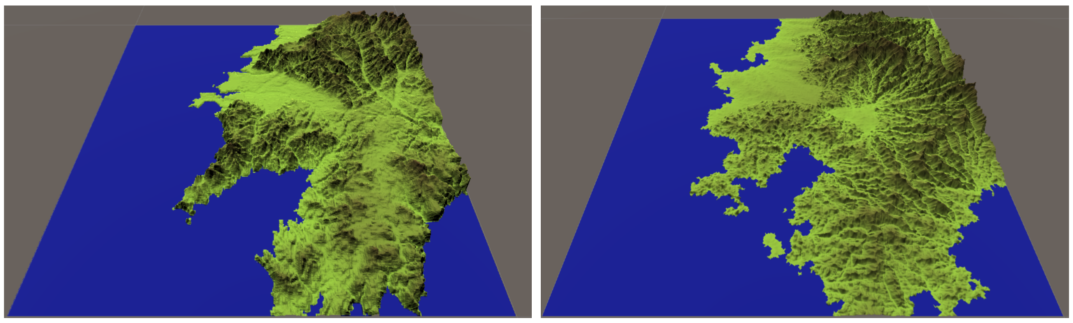


Figure 5.23 Comparison between the 3d-representation of real and generated terrain. The image on the left shows a section of the Severo-Evensky District in Magadan Oblast in Russia. The image on the right shows the terrain generated with this algorithm.

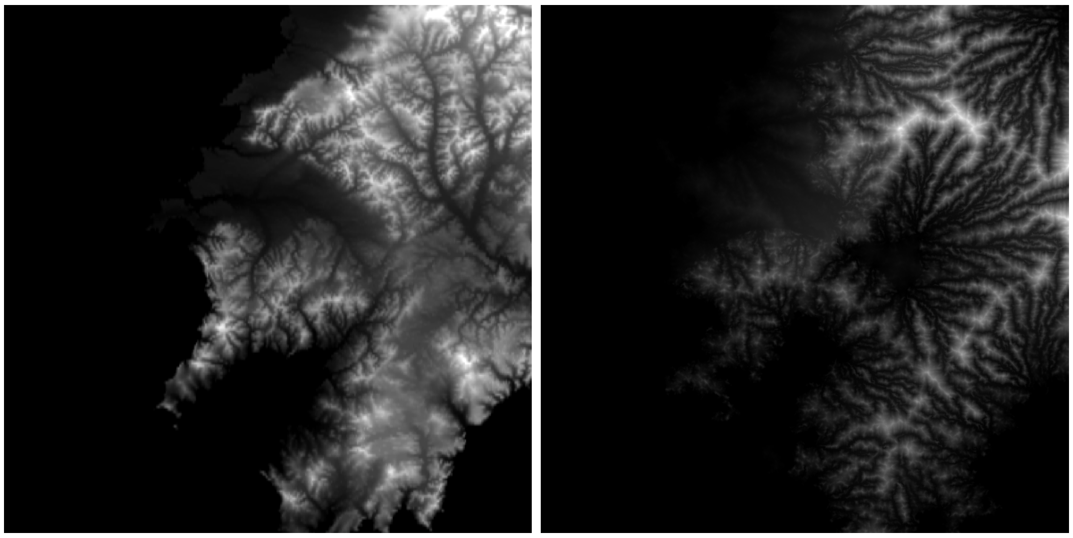
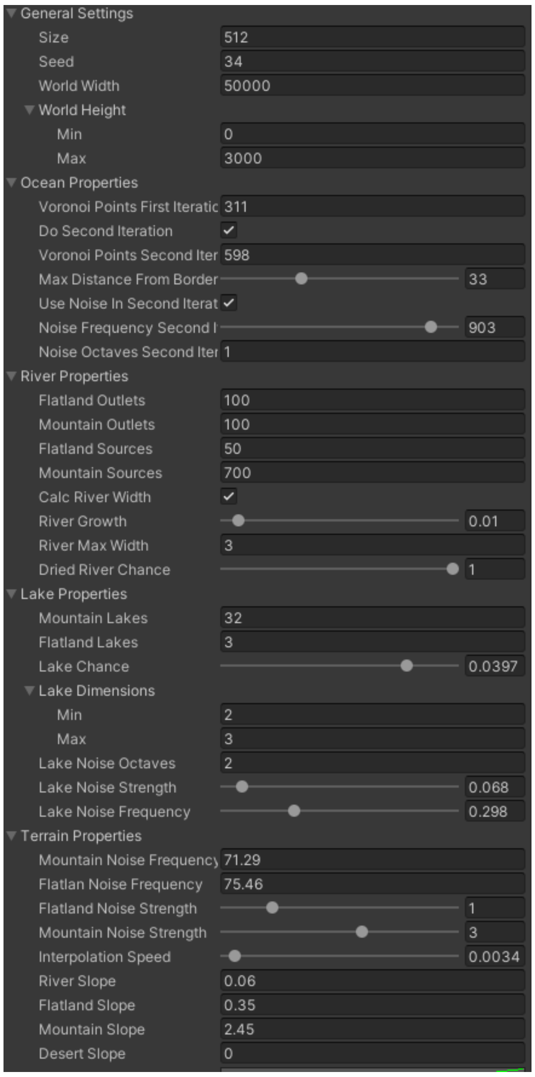


Figure 5.24 Comparison between a real life heightmap[1] (on the left) and a heightmap generated with this algorithm (on the right).

[1]Heightmap is taken from `https://heightmap.skydark.pl` on 11/28/2021



▼ General Settings	
Size	512
Seed	34
World Width	50000
▼ World Height	
Min	0
Max	3000
▼ Ocean Properties	
Voronoi Points First Iteratic	311
Do Second Iteration	✓
Voronoi Points Second Iter	598
Max Distance From Border	33
Use Noise In Second Iterat	✓
Noise Frequency Second I	903
Noise Octaves Second Iter	1
▼ River Properties	
Flatland Outlets	100
Mountain Outlets	100
Flatland Sources	50
Mountain Sources	700
Calc River Width	✓
River Growth	0.01
River Max Width	3
Dried River Chance	1
▼ Lake Properties	
Mountain Lakes	32
Flatland Lakes	3
Lake Chance	0.0397
▼ Lake Dimensions	
Min	2
Max	3
Lake Noise Octaves	2
Lake Noise Strength	0.068
Lake Noise Frequency	0.298
▼ Terrain Properties	
Mountain Noise Frequency	71.29
Flatlan Noise Frequency	75.46
Flatland Noise Strength	1
Mountain Noise Strength	3
Interpolation Speed	0.0034
River Slope	0.06
Flatland Slope	0.35
Mountain Slope	2.45
Desert Slope	0

Figure 5.25 Settings for the algorithm that were used to create the terrain presented in figure 5.24 and figure 5.23.

Chapter 6

Conclusion

The goal of this thesis was to create an algorithm to generate realistic procedural terrain with a focus on the generation of different kinds of water bodies. This approach should ensure that the result looks realistic while still balancing the performance. At the same time, the algorithm should aim for a good balance between automation and flexibility. Different methods, such as physically based simulation or noise based approaches, to reach those goals were analysed and evaluated in this thesis. The final result meets the expectations as were seen in chapter 5. A large variety of realistic looking landscapes can be created using the algorithm. The user has the possibility to change a lot of parameters for different steps of the pipeline. With these options they can control the reasonable parts of the algorithm while the other parts work in an automated fashion. The calculation times are also low enough to provide the possibility of running multiple iterations so as to try different parameter sets until the generated landscapes meets the requests. For example, ten iterations of the generation of a landscape with a resolution of 1024x1024 can be accomplished in under a minute. Several of those parameters control the way different water bodies are created. The user can control, for example, the shape of the ocean border, the denseness and form of the river networks and the shape and size of lakes. The results look realistic, and it is possible to recreate real landscapes to a certain degree. The size and height of the created terrain is hereby variable, and the parameters can be adapted to fit a landscape sized up to hundreds of square kilometres. The result is also usable in multiple applications as it can be presented in a form of a heightmap, which is the most common form of digital terrain in the game industry. It is conceivable that landscapes generated with the algorithm proposed in this thesis will be used in the map generation of open world games. In addition, other exported textures can, for example, transport the information of precipitation to other applications.
Even though the implementation shows good results, it could be optimized in multiple ways. One possible optimization could be to detect the changes in the parameters. In that case the pipeline can be started from a later step. On doing so, some difficult steps concerning the performance of e.g., the ocean border calculation with noise could be skipped. This would lead to a significant improvement in the runtime.

The approach of this thesis to base the height map generation on the previously calculated river networks, could be enhanced by alternating multiple times between those two steps. The landscape could be calculated more roughly in the first steps and refine the results in each step. This would resemble the real procedures of terrain generation more closely but would increase the computational effort. To make the created terrain even more realistic, more features could be added. This could include, for example, water features such as river deltas. Another water feature could be the addition of waterfalls although this is difficult with respect to the use of heightmaps as they do not provide the possibility to generate overhangs in the terrain. The selection of terrain types to place on the maps could be extended to allow for more variety of the respective areas. Furthermore, extending the settings that can be changed for every individual terrain point could influence the possibilities of different landscapes being produced.
Another possible extension to this thesis could be to include the results of terrain generation in other algorithms to add features that this thesis does not deal with. A good option for this idea could be the algorithm from (Fischer et al., 2020). There, the focus lies more on the climate simulation and the biome classification based on that. The basis for those approaches is rough terrain data generated with noise maps. Replacing those terrain generating steps with the terrain generation from this thesis and performing the simulation and classification on top of that could lead to realistic results that also include climate and biome information.
To sum up, good results were achieved with this thesis. A fast and customizable terrain generation system was created which leads to realistic results in a decent time. It is also extensible as multiple features could be added in future.

Bibliography

Alcaraz, S. A., Sannier, C., Vitorino, A. C., and Daniel, O. (2009). Comparison of methodologies for automatic generation of limits and drainage networks for hidrographic basins. *Revista Brasileira de Engenharia Agrícola e Ambiental*, 13:369–375.

Asybaris01 (2011). Modelare 3d pentru bazinul raului curpanul si cotii, afluenti ai oltului, licenced under th public domain `https://commons.wikimedia.org/wiki/File:Modelare_3D_pentru_Bazinul_Raului_Curpanul_si_Cotii,_afluenti_ai_Oltului.gif`.

Balister, P., Balogh, J., Bertuzzo, E., Bollobás, B., Caldarelli, G., Maritan, A., Mastrandrea, R., Morris, R., and Rinaldo, A. (2018). River landscapes and optimal channel networks. *Proceedings of the National Academy of Sciences*, 115(26):6548–6553.

Barnes, R., Lehman, C., and Mulla, D. (2014). Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. *Computers & Geosciences*, 62:117–127.

Berends, C. J. and Van De Wal, R. S. (2016). A computationally efficient depression-filling algorithm for digital elevation models, applied to proglacial lake drainage. *Geoscientific Model Development*, 9(12):4451–4460.

Bertuzzo, E., Maritan, A., Gatto, M., Rodriguez-Iturbe, I., and Rinaldo, A. (2007). River networks and ecological corridors: Reactive transport on fractals, migration fronts, hydrochory. *Water Resources Research*, 43(4).

Birkinshaw, S. (2010). Automatic river network generation for a physically-based river catchment model. *Hydrology and Earth System Sciences*, 14(9):1767–1771.

Chorowicz, J., Ichoku, C., Riazanoff, S., Kim, Y.-J., and Cervelle, B. (1992). A combined algorithm for automated drainage network extraction. *Water Resources Research*, 28(5):1293–1302.

Cordonnier, G., Braun, J., Cani, M.-P., Benes, B., Galin, E., Peytavie, A., and Guérin, E. (2016). Large scale terrain generation from tectonic uplift and fluvial erosion. In *Computer Graphics Forum*, volume 35, pages 165–175. Wiley Online Library.

Cortial, Y., Peytavie, A., Galin, E., and Guérin, É. (2020). Real-time hyper-amplification of planets. *The Visual Computer*, 36(10):2273–2284.

Costa-Cabral, M. C. and Burges, S. J. (1994). Digital elevation model networks (demon): A model of flow over hillslopes for computation of contributing and dispersal areas. *Water resources research*, 30(6):1681–1692.

Cox, A. and Hart, R. B. (2009). *Plate tectonics: How it works*. John Wiley & Sons.

Dai, Z., Huang, Y., and Xu, Q. (2019). A hydraulic soil erosion model based on a weakly compressible smoothed particle hydrodynamics method. *Bulletin of Engineering Geology and the Environment*, 78(8):5853–5864.

Doyle, P. and McMullen, C. T. (1989). Solving the quintic by iteration. *Acta Mathematica-Stockholm-*.

Erskine, R. H., Green, T. R., Ramirez, J. A., and MacDonald, L. H. (2006). Comparison of grid-based algorithms for computing upslope contributing area. *Water Resources Research*, 42(9).

Ertl, B. (2015a). Euclidean vornonoi diagram, licenced under creative commons attribution-share alike 4.0 international `https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg`.

Ertl, B. (2015b). Manhattan vornonoi diagram, licenced under creative commons attribution-share alike 4.0 international `https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg`.

Ewen, J., Parkin, G., and O'Connell, P. E. (2000). Shetran: distributed river basin flow and transport modeling system. *Journal of hydrologic engineering*, 5(3):250–258.

Fischer, R., Dittmann, P., Weller, R., and Zachmann, G. (2020). Autobiomes: procedural generation of multi-biome landscapes. *The Visual Computer*, 36(10):2263–2272.

Freeman, T. G. (1991). Calculating catchment area with divergent flow based on a regular grid. *Computers & geosciences*, 17(3):413–422.

Galin, E., Guérin, E., Peytavie, A., Cordonnier, G., Cani, M.-P., Benes, B., and Gain, J. (2019). A review of digital terrain modeling. In *Computer Graphics Forum*, volume 38, pages 553–577. Wiley Online Library.

Génevaux, J.-D., Galin, É., Guérin, E., Peytavie, A., and Benes, B. (2013). Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics (TOG)*, 32(4):1–13.

Grimaldi, S., Nardi, F., Di Benedetto, F., Istanbulluoglu, E., and Bras, R. L. (2007). A physically-based method for removing pits in digital elevation models. *Advances in water Resources*, 30(10):2151–2158.

Grimaldi, S., Petroselli, A., Alonso, G., and Nardi, F. (2010). Flow time estimation with spatially variable hillslope velocity in ungauged basins. *Advances in Water Resources*, 33(10):1216–1223.

Harris, M. J. (2003). *Real-time cloud simulation and rendering.* The University of North Carolina at Chapel Hill.

Jenson, S. K. and Domingue, J. O. (1988). Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric engineering and remote sensing*, 54(11):1593–1600.

Kurowski, M. (2012). Procedural generation of meandering rivers inspired by erosion. In *The 20th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision.* Citeseer.

Kweon, I. S. and Kanade, T. (1994). Extracting topographic terrain features from elevation maps. *CVGIP: image understanding*, 59(2):171–182.

Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D. S., Lewis, J. P., Perlin, K., and Zwicker, M. (2010). A survey of procedural noise functions. In *Computer Graphics Forum*, volume 29, pages 2579–2600. Wiley Online Library.

Le Pichon, X., Francheteau, J., and Bonnin, J. (2013). *Plate tectonics*, volume 6. Elsevier.

Leopold, L. B. and Maddock, T. (1953). *The hydraulic geometry of stream channels and some physiographic implications*, volume 252. US Government Printing Office.

Lin, W.-T., Chou, W.-C., Lin, C.-Y., Huang, P.-H., and Tsai, J.-S. (2006). Automated suitable drainage network extraction from digital elevation models in taiwan's upstream watersheds. *Hydrological Processes: An International Journal*, 20(2):289–306.

Martz, L. W. and Garbrecht, J. (1998). The treatment of flat areas and depressions in automated drainage analysis of raster digital elevation models. *Hydrological processes*, 12(6):843–855.

Mei, X., Decaudin, P., and Hu, B.-G. (2007). Fast hydraulic erosion simulation and visualization on gpu. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pages 47–56. IEEE.

Montgomery, D. R. and Foufoula-Georgiou, E. (1993). Channel network source representation using digital elevation models. *Water Resources Research*, 29(12):3925–3934.

Musgrave, F. K., Kolb, C. E., and Mace, R. S. (1989). The synthesis and rendering of eroded fractal terrains. *ACM Siggraph Computer Graphics*, 23(3):41–50.

Nardi, F., Grimaldi, S., Santini, M., Petroselli, A., and Ubertini, L. (2008). Hydrogeomorphic properties of simulated drainage patterns using digital elevation models: the flat area issue/propriétés hydro-géomorphologiques de réseaux de drainage simulés à partir de modèles numériques de terrain: la question des zones planes. *Hydrological sciences journal*, 53(6):1176–1193.

O'Callaghan, J. F. and Mark, D. M. (1984). The extraction of drainage networks from digital elevation data. *Computer vision, graphics, and image processing*, 28(3):323–344.

Pan, J., Wang, M., Li, D., and Li, J. (2009). Automatic generation of seamline network using area voronoi diagrams with overlap. *IEEE Transactions on Geoscience and Remote Sensing*, 47(6):1737–1744.

Peck, J. (2020). Fast noise simd `https://github.com/Auburn/FastNoiseSIMD`.

Peytavie, A., Dupont, T., Guérin, E., Cortial, Y., Benes, B., Gain, J., and Galin, E. (2019). Procedural riverscapes. In *Computer Graphics Forum*, volume 38, pages 35–46. Wiley Online Library.

Quinn, P., Beven, K., Chevallier, P., and Planchon, O. (1991). The prediction of hillslope flow paths for distributed hydrological modelling using digital terrain models. *Hydrological processes*, 5(1):59–79.

Rigon, R., Rinaldo, A., Rodriguez-Iturbe, I., Bras, R. L., and Ijjasz-Vasquez, E. (1993). Optimal channel networks: a framework for the study of river basin morphology. *Water Resources Research*, 29(6):1635–1646.

Rinaldo, A., Rigon, R., Banavar, J. R., Maritan, A., and Rodriguez-Iturbe, I. (2014). Evolution and selection of river networks: Statics, dynamics, and complexity. *Proceedings of the National Academy of Sciences*, 111(7):2417–2424.

Rinaldo, A., Rodriguez-Iturbe, I., and Rigon, R. (1998). Channel networks. *Annual review of earth and planetary sciences*, 26(1):289–327.

Rosgen, D. L. (1994). A classification of natural rivers. *Catena*, 22(3):169–199.

Seibert, J. and McGlynn, B. L. (2007). A new triangular multiple flow direction algorithm for computing upslope areas from gridded digital elevation models. *Water resources research*, 43(4).

Šťava, O., Beneš, B., Brisbin, M., and Křivánek, J. (2008). Interactive terrain modeling using hydraulic erosion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 201–210.

Tarboton, D. G., Bras, R. L., and Rodriguez-Iturbe, I. (1988). The fractal nature of river networks. *Water Resources Research*, 24(8):1317–1322.

Teoh, S. T. (2008). River and coastal action in automatic terrain generation. In *CGVR*, pages 3–9. Citeseer.

Vigil., J. F. (1997). Tectonic plate boundaries, licenced under th public domain `https://commons.wikimedia.org/wiki/File:Tectonic_plate_boundaries.png`.

Viitanen, L. (2012). Physically based terrain generation: Procedural heightmap generation using plate tectonics.

Woodward, F. I., Lomas, M. R., and Kelly, C. K. (2004). Global climate and the distribution of plant biomes. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 359(1450):1465–1476.

Zhang, H., Qu, D., Hou, Y., Gao, F., and Huang, F. (2016). Synthetic modeling method for large scale terrain based on hydrology. *IEEE Access*, 4:6238–6249.

Zhang, J., Wang, C.-b., Qin, H., Chen, Y., and Gao, Y. (2019). Procedural modeling of rivers from single image toward natural scene production. *The Visual Computer*, 35(2):223–237.