Leopold-Franzens-Universität Innsbruck

Institute of Computer Science
Interactive Graphics and Simulation Group

Bachelor Thesis

Procedural Generation of Mountain Ranges Based on Geology

Bernhard Fritz
bernhard.e.fritz@student.uibk.ac.at

advised by
Univ.-Prof. Dipl.-Inf. Matthias Harders

Innsbruck, September 20, 2016

Abstract

In this bachelor thesis several geological phenomena are examined in terms of what impact they have on landscape and how they can be implemented algorithmically. In the development process of all algorithms and simulations presented in this thesis, special emphasis was put on real-time capability.

When computers were not as powerful as today, terrain generation was purely based on fractals. Due to its performance, this technique was really popular at the time. Unfortunately, close inspection reveals that fractally generated terrain looks rather unnatural. This derives from the fact that fractals are self-similar, meaning that landscape features will unavoidably repeat themselves. Despite fractal-based terrain generation being incredibly fast, it will not be subject of further research. The problems, old-fashioned, synthetic terrain generation algorithms come with will be made clear later on in this thesis by comparing several well-known algorithms based on their configurability and realism.

However, highly configurable synthetic terrain generation algorithms should not be considered obsolete yet. An example of how GPU-based Voronoi diagrams can be used to procedurally generate synthetic terrain as well as how this technique enabled the development of a geology-based tectonic plate movement simulation will be demonstrated.

To further enhance the realism of an arbitrary computer generated scenery, it is common practice to make use of geology-based erosion algorithms. Erosion describes all geological phenomena involved in giving mountains their distinctive shape. By taking advantage of cellular automata as well as smoothed particle hydrodynamics fluid simulations it will be shown how it was possible to simulate thermal weathering and hydraulic erosion while still complying with the real-time capability requirements mentioned before.

Finally, several subtle GLSL shader techniques for enhancing the visual appearance of the rendered scene will be presented.

Keywords procedural, terrain, heightmap, generation, tectonic, plate, erosion, simulation, OpenGL, GLSL

Contents

1 Introduction **1**

2 Related Work **3**

3 Methods **5**
3.1 Diamond-Square algorithm 5
3.2 Fault algorithm 6
3.3 RMP algorithm 8
3.3.1 Graph-based implementation 8
3.3.2 Voronoi-based implementation 9
3.4 Thermal erosion algorithm 9
3.5 Hydraulic erosion algorithm 10
3.5.1 Implementation based on previous research 10
3.5.2 Evaluation of various shallow water models 12
3.5.3 Implementation based on Smoothed Particle Hydrodynamics (SPH) . . 13
3.6 Tectonic plate simulation 13
3.6.1 Implementation using Box2D physics engine 14
3.6.2 Generation of mountain ranges due to tectonic plate collisions 15
3.7 Visualization techniques 15
3.7.1 Texture splatting 15
3.7.2 Bloom shader effect 16
3.7.3 Exporting heightmaps 18
3.7.4 Capturing screenshots with stblib 19
3.7.5 Video recording with ffmpeg 19

4 Results **21**
4.1 Comparison of hydraulic erosion algorithms 21
4.2 Mountain range generation algorithm alterations 22
4.3 Thermal erosion algorithm variations 22
4.4 Graph-based and Voronoi-based RMP performance analysis 23

5 Conclusion & Future Work **27**

List of Figures

1.1 Heightmap grid 1

3.1 MPD initialization 5
3.2 MPD iteration 1 5
3.3 MPD iteration 2 5
3.4 Diamond-Square initialization 7
3.5 Diamond-Step 1 7
3.6 Square-Step 1 7
3.7 Diamond-Step 2 7
3.8 Square-Step 2 7
3.9 Fault initialization 7
3.10 Fault iteration 1 7
3.11 Fault iteration 2 7
3.12 Fault iteration 3 7
3.13 10 cones 10
3.14 100 cones 10
3.15 1000 cones 10
3.16 Rotated Von Neumann neighborhood 10
3.17 Thermal erosion 11
3.18 Hydraulic erosion 12
3.19 Normal scene render pass 17
3.20 Bright-pass filter pass 17
3.21 Horizontal Gauss blur pass 17
3.22 Vertical Gauss blur pass 17
3.23 Bloom shader pass 17
3.24 Simplified heightmap export process [14] 18

4.1 Crooked surface eroded by cellular automata-based hydraulic erosion 21
4.2 Pyramid surface eroded by cellular automata-based hydraulic erosion 21
4.3 Hemispheric surface eroded by cellular automata-based hydraulic erosion 21
4.4 Crooked surface eroded by SPH-based hydraulic erosion 22
4.5 Pyramid surface eroded by SPH-based hydraulic erosion 22
4.6 Hemispheric surface eroded by SPH-based hydraulic erosion 22
4.7 Visualization of the experiment 23
4.8 Each affected area raised by the same amount 23
4.9 Each affected area raised by a random amount 23
4.10 Heightmap generated with Fault algorithm 23
4.11 Height-aware thermal erosion 23
4.12 Constant thermal erosion 23

4.13 Voronoi performance stress test 25

List of Tables

4.1 Graph-based RMP performance analysis 24
4.2 Voronoi-based RMP performance analysis 25

Declaration

By my own signature I declare that I produced this work as the sole author, working independently, and that I did not use any sources and aids other than those referenced in the text. All passages borrowed from external sources, verbatim or by content, are explicitly identified as such.

Signature: _____ Date: _____

Chapter 1

Introduction

As I was born in Tyrol, a mountainous area in Austria, it made me curious about how mountains like these surrounding my homeland came into existence in the first place. According to my research, there is still no scientific consensus on what geological processes were originally involved in the creation of these rock formations. My interest in computer graphics allowed me to grasp geological phenomena, such as tectonic plate movement as well as various types of erosion, hands-on in terms of implementing algorithms and simulations that imitate said behavior.

The foundation of all algorithms and simulations discussed in this bachelor thesis are heightmaps and heightmap transformations. A heightmap is a concept in computer graphics that allows to model landscapes in an efficient way, meaning that terrain elevation is well-defined for any coordinate tuple (x, z). Due to this, it is possible to export heightmaps as grayscale image files. This process is described in detail in section 3.7.3. Heightmaps consist of a predefined number of tiles. These tiles again consist of two triangles each as depicted in fig. 1.1.

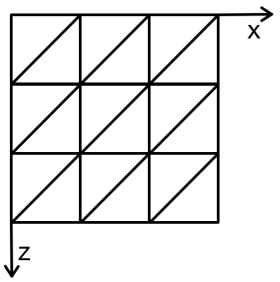

Figure 1.1: Heightmap grid

When speaking of heightmap resolution one can refer to the number of tiles it consists of. Usually heightmaps have square dimensions, meaning the number of columns and rows of tiles is the same.

To model the geological phenomena responsible for the creation of mountain ranges on Earth, it is necessary to have a basic understanding of how they work. As research suggests, tectonic plate movement is one of the main contributors when it comes to the generation of rock formations. The key principles of plate tectonics are based on the two essential layers Earth consists of: the lithosphere and asthenosphere. The cool and rigid lithosphere is divided in distinct, solid tectonic plates, which float on top of the hot, fluid-like asthenosphere. There

are several ways lithospheric plates can interact with each other:

- They can grind past each other, which results in strong earthquakes leaving the plates involved in this process essentially unaffected.
- They can slide apart from each other (diverging plates), which results in volcanic activity along the boundaries of the plates involved.
- They can slide toward each other (converging plates) and cause one plate to move underneath the other (subduction), or they can collide and pile up.

Tectonic plate movement is only one of many geological phenomena that directly influences shape and appearance of mountain ranges. Mountain features like distinct summits, ravines, valleys and cliffs are product of another type of geological phenomenon called erosion.

Erosion describes the geological processes involved in the transportation of soil, rock or dissolved material from one location to another. The driving forces responsible for this phenomenon are:

- Water (i.e. hydraulic erosion)
- Wind (i.e. wind erosion)
- Ice (i.e. glacial erosion)
- Gravity (i.e. thermal erosion)

Water in terms of rainfall, rivers or streams is able to dissolve sediment and carry it along potentially large distances before evaporating and leaving behind the dissolved material. This process is called hydraulic erosion and is commonly found in humid regions (→ section 3.5).

Wind erosion is primarily present in arid regions. Small airborne particles carried by wind are very capable to wear down rocks and produce new particles in the process of doing so.

Glacial erosion is a very effective way of erosion. The glacier slowly "crawls" along the mountain surface and picks up potentially large rocks that grind up the surface underneath it.

Thermal erosion is the result of temperature variations during day/night cycles, which can cause rocks to break apart and be transported to different locations due to gravity (→ section 3.4).

Before diving into more detail about the geological background involved in many of the algorithms and simulations presented in this bachelor thesis, it is important to note that not all terrain generation algorithms are based on geological observations. There are quite a few well-known alternatives (i.e. Diamond-Square or Fault algorithm) that do not necessarily obey the rules of nature. Nevertheless, they are capable of producing impressive results when combined with geology-based simulations or erosion algorithms. For this reason, both types of terrain generation algorithms have been researched on over the course of this thesis.

Chapter 2

Related Work

Kamal and Uddin evaluated various terrain generation algorithms, such as Diamond-Square- and Fault-Algorithm, and came up with their own algorithm called Repeated Magnification and Probing. In their paper *Parametrically Controlled Terrain Generation* [6] they compare many different terrain generation algorithms based on their configurability. Being able to control parameters such as location, spread and complexity of mountains is really useful when it comes to generating very specific scenes in movies, games and the like. Manual terrain modelling becomes more or less obsolete, since one can simply generate several instances of the same scene and pick the most appropriate.

Thermal and hydraulic erosion algorithms were first described by Musgrave. In his paper *The Synthesis and Rendering of Eroded Fractal Terrains* [12] he presented, among other things, how geological erosion processes can be implemented algorithmically and shared some ideas about how material hardness can be included in these algorithms.

Inspired by Musgrave's work, Olsen implemented his own versions of thermal and hydraulic erosion algorithm and also improved both algorithms performance-wise to make them available for real-time applications. His paper *Realtime Procedural Terrain Generation* [13] includes some valuable insights on how to generate realistic terrain with speed optimized versions of thermal and hydraulic erosion algorithms. His techniques were adapted and improved by considering material hardness during the erosion process in the course of this bachelor thesis.

In his paper *Physically Based Terrain Generation* [17], Viitanen describes the theoretical foundations of plate tectonics, compares several existing applications and ultimately implements his own tectonic plate simulation. His in-depth implementation supports converging, diverging as well as subducting plates. Due to the amount of detail, Viitanen was not able to make his simulation real-time capable. Nevertheless, his implementation is one of the most advanced tectonic plate simulations out there.

Allen implemented a continental drift simulation called *cdrift* [2], which models tectonic plate movement based on the assumption that there is a supercontinent in the beginning which breaks up several times and causes each individual subcontinent to float around for a while. After some time, all fragments are drawn back together and some of them merge in this process resulting in the creation of new continents.

Chapter 3

Methods

3.1 Diamond-Square algorithm

This algorithm was first introduced by Fournier et al in 1982 [5]. It can be considered as an extension of Midpoint-Displacement (MPD) algorithm. By recursively subdividing a line into segments of equal length and displacing the resulting midpoints by a random amount, MPD algorithm is able to produce a fairly arbitrary looking 2D-landscape. Figures 3.1 to 3.3 depict the first two iterations of MPD algorithm.

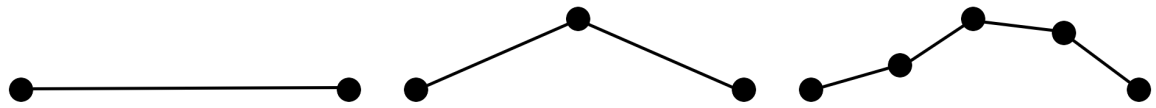

Figure 3.1: MPD initialization Figure 3.2: MPD iteration 1 Figure 3.3: MPD iteration 2

To take MPD algorithm to the next level, Diamond-Square algorithm allows to procedurally generate 3D-landscape. During initialization, Diamond-Square algorithm raises the heightmap's corner vertices to an arbitrary height. Once the corner vertices have been initialized, a *Diamond-Step* followed by a *Square-Step* will be performed in an alternating fashion until all vertices have been initialized. The following pseudocode will demonstrate how Diamond-Square algorithm can be implemented:

```
function randomFloat(range):
  return a random floating point number ∈ [0, range)

procedure diamondStep(hm, roughness, left, right, top, bottom):
  midpointColumn = left+(right-left)/2
  midpointRow = top+(bottom-top)/2
  avgHeight = ... // average height of square corners
  r = roughness * (randomFloat(2)-1)
  hm.setHeightAt(midpointColumn, midpointRow, avgHeight+r)

procedure squareStep(hm, roughness, left, right, top, bottom):
  midpointColumn = left+(right-left)/2
```

```
  midpointRow = top+(bottom-top)/2
  avgHeightTop = ... // average height of top diamond corners
  r = roughness * (randomFloat(2)-1)
  hm.setHeightAt(midpointColumn, top, avgHeightTop+r)
  ... // similar steps for left, right and bottom diamond

procedure ds(hm, roughness, left, right, top, bottom):
  diamondStep(hm, roughness, left, right, top, bottom)
  squareStep(hm, roughtness, left, right, top, bottom)

procedure perform(hm, roughness):
  columns = hm.getColumns()
  rows = hm.getRows()

  // initialize heightmap corners
  hm.setHeightAt(0, 0, roughness)
  hm.setHeightAt(columns, 0, roughness)
  hm.setHeightAt(0, rows, roughness)
  hm.setHeightAt(columns, rows, roughness)

  while columns > 1:
    for row = 0; row < hm.getRows(); row += rows:
      for column = 0; column < hm.getColumns(); column += columns:
        ds(hm, roughness, column, column+columns, row, row+rows)
    roughness /= 2.0
    columns /= 2
    rows /= 2
```

Listing 3.1: DS pseudocode

Figures 3.4 to 3.8 depict the Diamond-Square algorithm applied to a 4x4 heightmap. Orange vertices are being altered in height, while blue vertices contribute to the average height calculation.

3.2 Fault algorithm

This algorithm was first described by Krten [7]. He proposed to randomly draw a line across the heightmap and split the landscape into two regions. By raising one region and lowering the other and repeating the whole process several times, interesting mountain-like features can emerge. Due to the sheer randomness of this algorithm, it is impossible to control the position, shape or spread of the resulting mountains. However, it is possible to control the roughness of the generated terrain by adjusting the amount by which vertices are raised or lowered per iteration.

To determine if a heightmap vertex needs to be raised or lowered, a Boolean function called *raise* will be used. Function *raise* is defined as follows:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

$$\mathbf{v} = (v_1\ v_2\ v_3)^\intercal$$

$$raise(\mathbf{v}) = \begin{cases} 0 & \text{if } sgn(v_2) = -1, \\ 1 & \text{if } sgn(v_2) \geq 0. \end{cases}$$

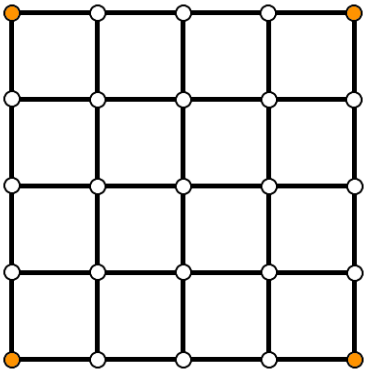

Figure 3.4: Diamond-Square initialization

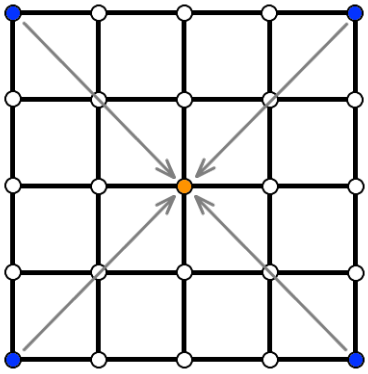

Figure 3.5: Diamond-Step 1

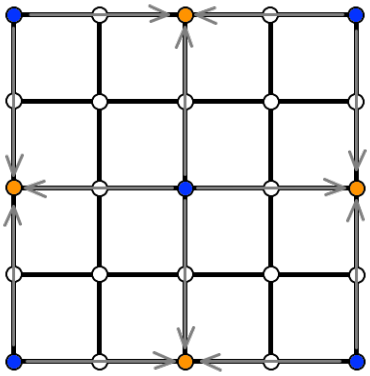

Figure 3.6: Square-Step 1

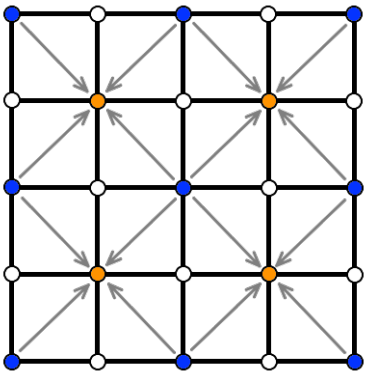

Figure 3.7: Diamond-Step 2

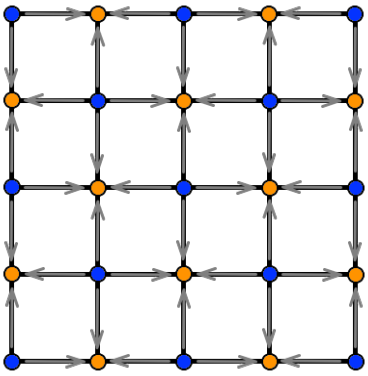

Figure 3.8: Square-Step 2

Let vectors **a** and **b** describe a random cutting line. Then a vertex pointed to by vector **p** will be raised in height if $raise((\mathbf{p}-\mathbf{a}) \times (\mathbf{b}-\mathbf{a}))$ equals 1 and lowered otherwise. Figures 3.9 to 3.12 will depict the first three iterations of Fault algorithm using different shades of green. Light green represents high and dark green low elevation respectively.

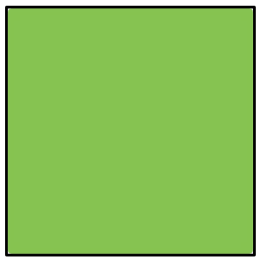

Figure 3.9: Fault initialization

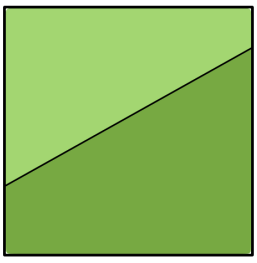

Figure 3.10: Fault iteration 1

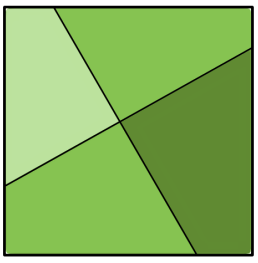

Figure 3.11: Fault iteration 2

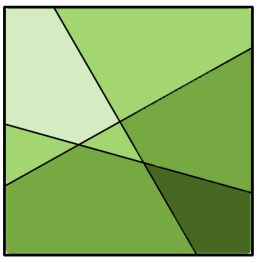

Figure 3.12: Fault iteration 3

3.3 RMP algorithm

RMP stands for *Repeated Magnification and Probing*, was created by Kamal and Uddin [6] and can be categorized as parametrically controllable mountain generation algorithm. Height, spread and location of the mountain can be controlled by parameters. This algorithm is similar to Fault algorithm, meaning that only selected regions are repeatedly altered in height. The process of how these regions are obtained is not described in Kamal and Uddin's paper [6]. Nevertheless, in sections 3.3.1 and 3.3.2 two techniques will be presented that are capable of generating randomly distributed polygon-like shapes which can be used for RMP algorithm.

3.3.1 Graph-based implementation

This implementation starts by placing four lines along the landscape borders as well as l randomly distributed cutting lines all over the landscape. In the next step all line intersections will be determined. To determine if two lines intersect, i.e. they share a point that lies on both lines, the following equation system has to be solved:

$$\mathbf{p} = \mathbf{a} + u * \mathbf{d} \tag{3.1}$$
$$\mathbf{p} = \mathbf{b} + v * \mathbf{e} \tag{3.2}$$

The following notation is used to extract a vector's x and y coordinates:

$$\mathbf{v} = (v_1 \ v_2)^\intercal \tag{3.3}$$

Expanding eqs. (3.1) and (3.2) using notation from eq. (3.3) results in:

$$p_1 = a_1 + u * d_1 \tag{3.4}$$
$$p_2 = a_2 + u * d_2 \tag{3.5}$$
$$p_1 = b_1 + v * e_1 \tag{3.6}$$
$$p_2 = b_2 + v * e_2 \tag{3.7}$$

Eliminating p_1 and p_2 from eqs. (3.4) to (3.7) results in:

$$a_1 + u * d_1 = b_1 + v * e_1 \tag{3.8}$$
$$a_2 + u * d_2 = b_2 + v * e_2 \tag{3.9}$$

Solving eq. (3.8) for u results in:

$$u = \frac{b_1 + v * e_1 - a_1}{d_1} \tag{3.10}$$

Substituting u in eq. (3.9) by eq. (3.10) and solving for v results in:

$$v = \frac{(b_2 - a_2) * d_1 - (b_1 - a_1) * d_2}{e_1 * d_2 - e_2 * d_1} \tag{3.11}$$

Substituting v in eq. (3.2) by eq. (3.11) results in $\mathbf{p}$ if the two lines intersect, i.e. they are not parallel.

(3.12)

These intersections are represented as vertices in an undirected graph. A line can have several intersections. Therefore, they will be iterated through from the start of the line to the end. Let **i** and **j** be two intersections on a line defined by origin **a** and direction **d** as seen in eq. (3.1). To determine which intersection is closer to the start of the line, one has to substitute **p** in eq. (3.1) with **i** and **j** and solve both resulting equations for u. The smaller u gets, the closer it is to the origin of the line. Intersections (i.e. vertices) along a line will be connected in order (closest to farthest) using graph edges. Once the graph is set up, a Depth-First-Search (DFS) based algorithm will be used to determine all minimal cycles. These cycles represent the polygon shapes produced by the previously placed cutting lines. Once the polygons are obtained, another undirected graph will be created. This time the vertices of the graph will represent the polygons. In this graph, two vertices are connected by an edge only if both polygons are next to each other. Once this newly generated graph is set up, another DFS will be performed. As starting point of this search, the polygon which should contain the desired mountain peak is chosen. The combined region of the first r elements of the DFS will be raised in height. A point-in-polygon test, originated by Shirmat [16] and later implemented in C by W. Randolph Franklin [19], was used to determine if a heightmap point should be affected by height alterations or not. The whole procedure will be repeated n times. After several tests, it was clear that the graph-based implementation was far too slow to be used for real-time applications. Therefore, an alternative implementation had to be considered.

3.3.2 Voronoi-based implementation

Instead of using expensive graph algorithms, the idea is to use a Voronoi diagram that does the same job by providing randomly distributed polygons. Still, the problem remains. How should one cheaply generate these diagrams? Actually, there is an intuitive solution to this problem. Imagine an arbitrary amount of randomly colored, flat shaded, overlapping cone geometries that are randomly distributed on a plane. If one looks at this scene from the top using orthographic projection, a Voronoi diagram can be identified. After some research, it appears that this technique was first mentioned in chapter 14 of the *OpenGL Red Book* [18]. This implementation relies on OpenGLs depth testing mechanism and is much faster than the graph-based approach. To combine this technique with RMP, one has to project the Voronoi diagram onto the heightmap, check the color of the polygon at the desired peak location and raise all heightmap tiles that have the same color projected onto them. By repeating the whole process n times, mountains with predefined peak locations can be generated. The spread of the mountain can be altered by increasing or decreasing the amount of cones to be placed during Voronoi generation. The more cones there are on the plane, the more polygons will be created and the resulting mountain will be very steep. Using less cones will result in less polygons, and the mountain will therefore be more spread. The height of the mountain is proportional to the amount of iterations performed. Figures 3.13 to 3.15 will depict three Voronoi diagrams using different amounts of cones.

3.4 Thermal erosion algorithm

Thermal erosion describes the process of material breaking loose due to variations in temperature during day/night cycles. This algorithm was implemented based on Jacob Olsen's description [13]. Basically, he used a cellular automaton to model the process of thermal erosion. Cellular automata operate on grids of cells. By inspecting a cell itself and its neighborhood, one can decide how they should interact with each other. Olsen recommended to use

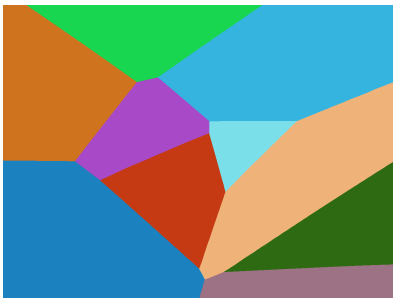

Figure 3.13: 10 cones

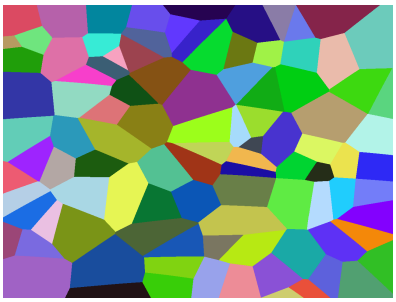

Figure 3.14: 100 cones

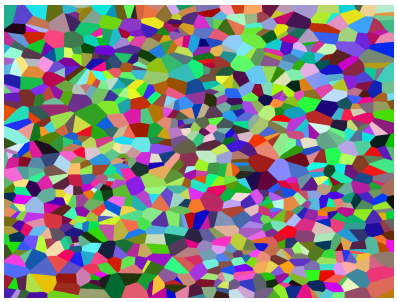

Figure 3.15: 1000 cones

a rotated Von Neumann neighborhood as depicted in fig. 3.16. Each of these cells correspond to heightmap tiles. If the height difference of the inspected cell and one of its neighbors is larger than a so-called talus angle T, material will be transported from the inspected cell to the corresponding neighbor cell as depicted in fig. 3.17. This process will be repeated for all cells/tiles of the heightmap, resulting in a general smoothing of the landscape. Especially steep parts of the heightmap are most affected by this erosion process. By default, Jacob Olsen's algorithm disregards material properties such as hardness or softness (e.g. rock is hard, sand is soft). Therefore, as an extension of this algorithm, a way to also consider material properties will be introduced. By multiplying the amount of material to be transported by a factor that depends on the height of the inspected cell, it is possible to mimic the desired material properties.

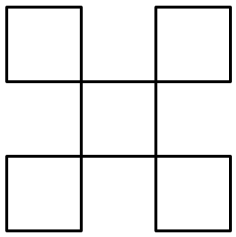

Figure 3.16: Rotated Von Neumann neighborhood

3.5 Hydraulic erosion algorithm

3.5.1 Implementation based on previous research

In my research I came across two different papers that described how to model hydraulic erosion algorithmically. The first one [13] was written by Jacob Olsen and featured a decent algorithm based on a cellular automaton. The idea behind this algorithm is that water (e.g. rain) causes erosion of terrain by dissolving material which is transported by running water, and once the water evaporates, the material will be deposited again. To model this process, it is necessary to keep track of the height, water as well as sediment levels. Therefore, not only a heightmap but also a water and sedimentmap is necessary. Each iteration can be broken down into four steps:

In the first step, a constant amount of water is added to each cell of the watermap.

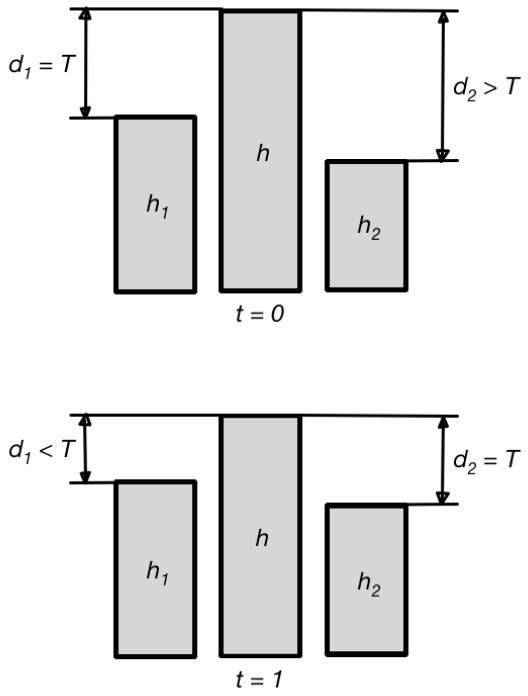

Figure 3.17: Thermal erosion

In the second step, water dissolves a specific amount of material by removing material from the heightmap and adding material to the sedimentmap. A similar technique, as seen in section 3.4, is used to also consider material hardness. This time around, not the height but the steepness of the mountain influences the amount of material to be dissolved.

In the third step, water will transport sediment to lower altitudes. Since water is a fluid, it will always try to level out, meaning it will always flow from higher to lower tiles if they are next to each other. Olsen described a cellular automaton-based approach as depicted in fig. 3.18 which is capable of simulating water flow. There are two cases to consider: If the main cell's water level is larger than the difference of the maximum altitude a (i.e. sum of cell height and water level) and the average altitude $\overline{a}$, then the excess water will be distributed among the neighboring cells while not causing them to overflow. In the other case, i.e., if the difference of a and $\overline{a}$ is larger than the main cell's water level, the water of the main cell will be fully depleted and distributed evenly among the neighboring cells.

In the fourth step, water will be evaporated, and sediment will be deposited at its current position by lowering the cells of the water as well as sedimentmap and raising the cells of the heightmap. The whole process will be repeated n times.

The second paper [9] by Mei et al showcases a more advanced model of hydraulic erosion. The basic principle is still the same as described by Olsen, but Mei uses a more sophisticated water flow model, which also considers fluid velocity. Due to the force of flowing water, more sediment will be carried away, which generally results in a more realistic model of hydraulic erosion. After studying Mei's algorithm and implementing his water flow model, a flaw in his algorithm became evident. Given that Mei's algorithm is timestep-based, it gets unstable if

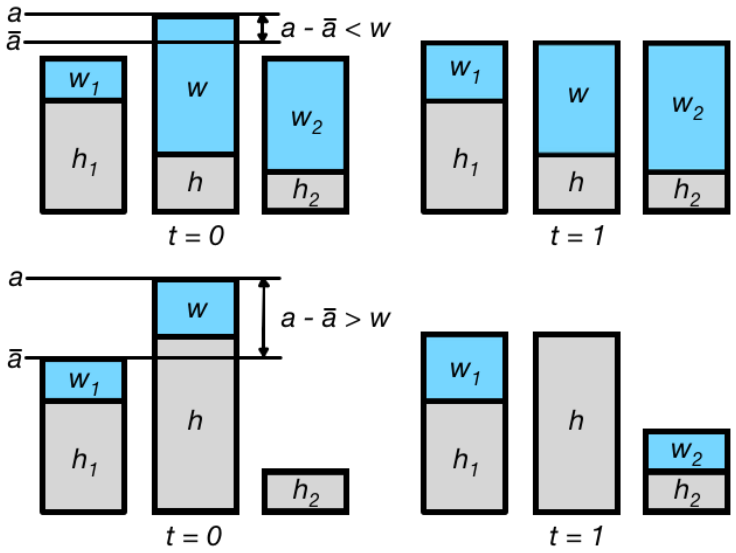

Figure 3.18: Hydraulic erosion

too large timesteps are used. Water waves begin to oscillate and ultimately the water height calculation will result in an arithmetic overflow and render the simulation unusable. To keep the simulation stable, very small timesteps have to be used. In fact, the timesteps required to keep the simulation stable are so small (e.g. $\Delta t = 10$ms) that one cannot even notice any difference in water flow for a couple of minutes. For a real-time application this is totally unacceptable, and therefore, this model was disregarded.

3.5.2 Evaluation of various shallow water models

After implementing Olsen's hydraulic erosion model and testing various parameters, it turned out that the results were not too realistic. Due to the lack of performance, the result of using Mei's variant was not too satisfying either. To achieve higher realism, a more sophisticated, yet performant model had to be considered. While researching about hydraulic erosion, I also stumbled upon shallow water models. These models are commonly used to predict tsunami movement but can also be utilized for other purposes such as hydraulic erosion simulation. Due to the complexity at hand, I was not able to implement my own shallow water model based on previous work and kept on looking for existing solutions. I found two promising projects: The first one [3], written in C++ by Alex Darcy, used a Riemann solver to numerically solve the shallow water equations. The only problem was that he assumed that the terrain underneath the water was perfectly flat. This might be a valid assumption for tsunami simulations but did not work for my area of application.

The second project [4] by Trevor Dixon seemed very promising. He implemented the shallow water equations using JavaScript and Three.js framework. His code was much more readable in comparison to Alex Darcy's. The only thing missing was water ground interaction. After a considerable amount of time, I was ultimately not able to figure out how to include this feature and went back to searching for alternatives. This is when I found out about smoothed particle hydrodynamics fluid simulations.

3.5.3 Implementation based on Smoothed Particle Hydrodynamics (SPH)

There are dozens of different existing SPH solutions out there. Most of them were proprietary and those that were not, brought other issues with them such as hardware incompatibility or real-time incapability. Last but not least, I came across a GitHub project [8] by Saeed Mahani. He wrote a basic SPH simulation featuring a box which is filled up by fluid particles while obeying the laws of physics. The project was real-time capable and had no special hardware requirements. However, several alterations to the SPH simulation had to be made. First of all, the whole visualization code had to be replaced to be compatible with the rest of my GLFW-based project. After that, the simulation had to be extended to also support heightmaps, which it previously did not. The original code only supported planes which could not be used to model arbitrarily shaped surfaces. Due to this, an alternative solution had to be implemented. My first idea was to simply add support for static (non-moving) particles and distribute them all across the heightmap. After implementing it, it became evident that this approach had several flaws. Not only were particles passing through the heightmap because of small holes inbetween the static particles, but the simulation also became horribly slow due to the sheer amount of particles present in the scene. As a result, a different solution had to be found. In order to make the particles appear to be influenced by the heightmap surface, some kind of force has to counteract their tending downwards movement due to gravity. To implement this kind of behavior, it is necessary to continuously check if a particle is touching the surface. A fast triangle-ray intersection test by Möller and Trumbore [10] was used to check if a particle is below the heightmap surface or not. If the intersection test is indeed positive, a short impulse in the direction perpendicular to the impact point on the heightmap surface has to be applied to the particle. Since the normals are only calculated per heightmap vertex, it was necessary to approximate the impulse direction at the impact point using interpolation. Since the heightmap is rasterized into tiles, it can be guaranteed that there are always four normals in close proximity to the impact point. These normals are normalized vectors. To get the interpolated normal from these four vectors, one has to sum them up and normalize the result. The performance of this solution was much better than the previous one, and no more particles were dropping below the surface.

Once the SPH simulation was working as intended, it was possible to simulate a more realistic hydraulic erosion model. The idea is that each particle gets some capacity to store sediment. If a particle touches the terrain, it dissolves a specific amount of it, based on the predefined particle acidity. Additionally, each particle has a predefined lifespan, and once its time runs out, it evaporates and leaves behind the sediment it carried. Heightmap changes take place if a particle dissolves sediment or if a particle evaporates. A particle potentially travels a large distance, which implies that carried sediment will almost always end up somewhere distant from where it was dissolved in the first place. To simulate rain, the particles will be spawned at a certain altitude at random positions on the xz-plane. The user can observe the simulation in real time and experience the effects of hydraulic erosion first-hand.

3.6 Tectonic plate simulation

Due to the complexity of real tectonic plate movement, the design of the simulation had to be simplified to a degree which allowed it to be implemented programmatically. As a result, the problem at hand was reduced to two dimensions. The general idea is to split the heightmap into several pieces, representing tectonic plates. In addition, they should be kept in motion to simulate magma flowing underneath the surface. To model this non-trivial behavior, I

decided to use a physics engine. Box2D, a 2D physics engine, supports convex polygons with up to eight vertices as well as collision detection, is very well documented and offers a test environment called *testbed*. Splitting of arbitrarily shaped polygons is not supported by Box2D though. Nevertheless, this engine was far superior to its alternatives in terms of documentation, community support and features. After going through the documentation and some code samples, I was able to implement the missing features myself.

3.6.1 Implementation using Box2D physics engine

When working with Box2D, it is very common to start off by writing a test application for *testbed*. To make this test application available in *testbed*, it is necessary to extend the class *Test* provided by Box2D and list the newly created test class in *TestEntries.cpp* as specified in the Box2D documentation. The advantage of *testbed* is that it provides its own visualization interface as well as several debugging options. The downside is that *testbed* tests can not be used for production systems right away and have to be altered during the integration process. Once the base test class is set up, several methods such as *Step*, *Create*, *Keyboard*, etc. provided by the super class *Test* can be overwritten. *Step* will be called automatically by the test environment at each timestep. *Create* will be called only once and should be used to initialize variables or data structures. Method *Keyboard* will be triggered each time a key is pressed. Its parameters give information about which key was pressed. To add an object to the Box2D world, several steps are necessary. Firstly, a body object has to be defined. Secondly, a fixture object has to be defined and linked with the body object it corresponds to. In Box2D, bodies can either be static, dynamic or kinematic. Static bodies do not move. Even if a dynamic body collides with a static body, only the dynamic body will be affected by the collision. Dynamic bodies can move and are affected by all other bodies. Kinematic bodies can also move but are not affected by collisions caused by dynamic bodies. For the tectonic plate simulation, dynamic as well as kinematic bodies will be used. The dynamic bodies represent the tectonic plates and the kinematic bodies are responsible for keeping the simulation in motion as well as keeping the dynamic bodies contained within the simulation area. Body objects can be used to access information such as current position, angle, velocity, mass, etc. The shape of a Box2D body depends on its fixture. To define a fixture, it is necessary to set the fixture's density attribute and shape. Based on the fixture's density and area it consumes, the body's mass will be determined. Box2D offers basic shapes such as rectangles and circles as well as convex polygons with a maximum of eight vertices. The tectonic plate simulation will be initialized with five rectangle fixtures. Four of these will be used to define the kinematic bodies' shapes, i.e., thin beams. The last one will be used to define the dynamic body's shape, a big square. This square will be randomly split a specific amount of times during set up. Unfortunately, Box2D did not offer this splitting functionality per se, and it had to be implemented by hand. To implement splitting of polygons, it is important to know that Box2D only supports convex polygons with up to eight vertices, which need to be specified in counter clockwise winding. In order to split polygons, Box2D's raycasting feature can be very helpful. It allows the programmer to cast rays, and if a ray intersects with an object, a callback function will be called. When a ray is cast through a convex polygon, there should always be exactly one entry point and one exit point. Box2D's raycasting only registers the first intersection point per object though. Due to this, two rays had to be cast in opposite directions. Having both, the first ray's entry point and the second ray's entry point as well as all vertices the polygon consists of, the only thing left to do is to determine which vertex belongs to which slice of the polygon. The following pseudocode should adequately elaborate

this procedure:

```
entry = entry point of first ray
exit = entry point of second ray
rayCenter = ((entry.x+exit.x)/2,(entry.y+exit.y)/2)
rayAngle = atan((entry.y-exit.y)/(entry.x-exit.x))
for each vertex v of polygon:
  cutAngle = atan((v.y-rayCenter.y)/(v.x-rayCenter.x))-rayAngle
  if cutAngle < -PI:
    cutAngle += 2*PI
  if cutAngle > 0 and cutAngle <= PI:
    // vertex v belongs to polygon1
  else:
    // vertex v belongs to polygon2
```

Listing 3.2: Point in slice check pseudocode

Once the polygons (i.e. tectonic plates) are all set up, collision callbacks provided by Box2D will be used to act whenever a collision occurs. For the tectonic plate simulation only the collisions between the dynamic objects, i.e. the plates, are important. Therefore, the program is set up to only act on collisions in which two dynamic objects are involved. Additionally, collisions between two polygons having less than two collision points are disregarded. By enforcing these constraints, the program will only act on collisions that occur between two dynamic objects and consist of exactly two collision points. Finally, in order to use these collision points for heightmap alterations, they have to be transformed from the Box2D coordinate system to the heightmap coordinate system.

3.6.2 Generation of mountain ranges due to tectonic plate collisions

Each tectonic plate collision results in two collision points due to the custom collision callback method mentioned before. By connecting these points with an imaginary line, it would be possible to procedurally generate a mountain range along it. This is where an advanced version of RMP algorithm comes into play. Similarly, this version also uses a Voronoi diagram to generate arbitrary polygons. Different from the previous RMP implementation, there will be two coordinate pairs instead of a single one. Between these coordinates, Bresenham line algorithm will be performed, and all colors passed by the algorithm will be remembered. In the next step, the Voronoi diagram will be projected onto the heightmap, and all heightmap vertices inside polygons that are colored with one of the remembered colors will be raised in height. This procedure will be repeated each time a tectonic plate collision occurs. As a result, procedurally generated mountain ranges will arise whenever two tectonic plates collide.

3.7 Visualization techniques

3.7.1 Texture splatting

Texture splatting can be used to make a computer generated terrain more visually appealing. By using textures and blending them together based on height and slope, it is possible to texture procedurally generated terrain regardless of shape and complexity of terrain features. For this application four textures have been chosen:

- Sand
- Grass

- Stone
- Snow

Blending textures together is performed by the fragment shader. The OpenGL shader language GLSL offers a function *mix* that allows to linearly interpolate between two texels. *Mix* consists of three parameters:

- First texel color
- Second texel color
- A floating-point number between 0.0 and 1.0 representing the mixture ratio

```
color0 = sand texture
color1 = grass texture
color2 = rock texture
color3 = snow texture

max_height = max height of terrain
min_height = min height of terrain

difference = max_height-min_height
delta = difference / 4
threshold0 = delta*1
threshold1 = delta*2
threshold2 = delta*3

position = pass-through position from vertex shader

if position.y < threshold0:
  texel = color0
if position.y >= threshold0 and position.y < threshold0+delta):
  texel = mix(color0, color1, (position.y-threshold0)/delta)
if position.y >= threshold0+delta and position.y < threshold1):
  texel = color1
// the remaining textures are blended similarly...
```

Listing 3.3: Fragment shader texture chooser pseudocode

At first only the height information was used to determine which texture should be applied to the terrain. Of course that model did not look too realistic since the slope information had not been considered back then. In nature, grass does not grow on steep slopes. Similarly, snow can mostly be found on flat areas. Therefore, in steep slopes grass and snow texels have to be overridden by rock texels. A common approach to determine the steepness of a slope is to use the normal vectors' y coordinates. Using thresholds, it is again possible to utilize OpenGL's *mix* function to mix two texels. Unfortunately, there are some special cases, namely steep slopes inside an area between two height thresholds, which required bilinear interpolation instead of linear interpolation. These cases were solved by calling *mix* in a nested way, meaning the first or second parameter of *mix* is itself the result of a previous *mix* invocation.

3.7.2 Bloom shader effect

A bloom shader effect is a popular choice to enhance the realism of a scene. The goal is to reproduce image imperfections caused by real-world cameras. Brightly lit spots often appear to be glowing when captured with a camera. To model this effect two filters are necessary:

- Bright-pass filter
- Blur filter

The bright-pass filter will be applied to the original frame. After that, the result will be blurred using a blur filter, e.g. a two-pass Gaussian filter. Finally, the original frame as well as the blurred bright-pass frame will be blended together resulting in an image with brightly glowing spots. Altogether the bloom filter consists of five rendering passes:

1. Normal scene render pass (fig. 3.19)
2. Bright-pass filter pass (fig. 3.20)
3. Horizontal Gauss blur pass (fig. 3.21)
4. Vertical Gauss blur pass (fig. 3.22)
5. Bloom shader pass (fig. 3.23)

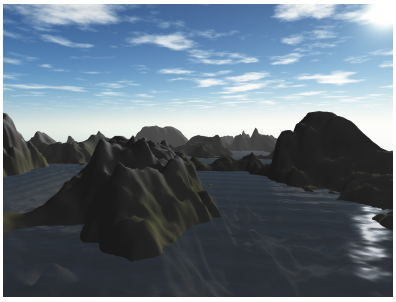

Figure 3.19: Normal scene render pass

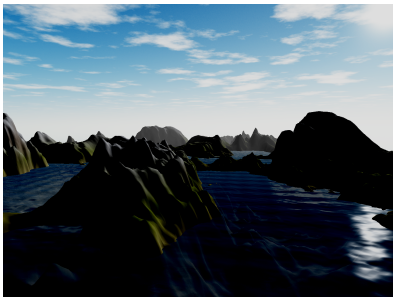

Figure 3.20: Bright-pass filter pass

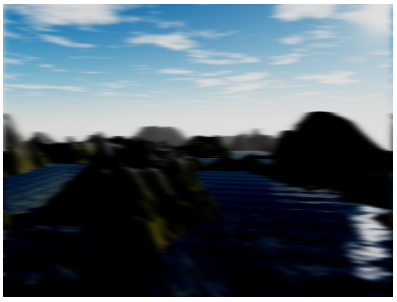

Figure 3.21: Horizontal Gauss blur pass

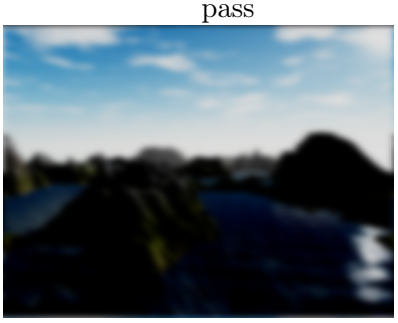

Figure 3.22: Vertical Gauss blur pass

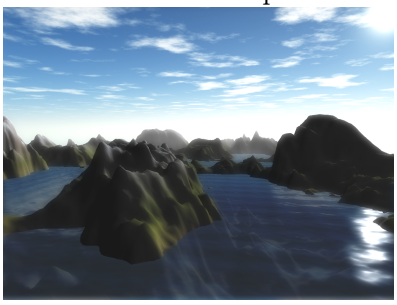

Figure 3.23: Bloom shader pass

Bright-pass filter

This filter makes bright regions even brighter while not modifying dark regions. It is possible to specify the range of colors to be brightened as well as the amount they should be brightened by. The bright-pass shader code is based on Erik Reinhard's formula:

$$L_d(x,y) = \frac{L(x,y) * \frac{1+L(x,y)}{L^2_{white}}}{1+L(x,y)}$$

which can be found in his paper [15] called *Photographic Tone Reproduction for Digital Images*.

Two-pass Gaussian filter

A two-pass Gaussian filter consists of two passes, namely a horizontal and a vertical Gaussian blur pass. During these passes, each pixel will be averaged using a one-dimensional Gaussian kernel. The two-pass Gaussian filter is computationally less expensive than the one-pass Gaussian filter using a two-dimensional Gaussian kernel. For this filter, the vertex shader is responsible for setting up the pixel coordinates to be averaged by the fragment shader based on Gauss distributed weights.
Vertex shader pseudocode:

```
UV = input pixel location
direction = (1,0) or (0,1) depending on pass direction
offsets[] = one-dimensional Gaussian kernel

for i between 0 and offsets.length:
  blurUV[i] = UV+(direction.x*offsets[i],direction.y*offsets[i])
```

Listing 3.4: Two-pass Gaussian filter vertex shader

Fragment shader pseudocode:

```
texture = input frame
blurUV[] = array of input pixel locations to be averaged
weights[] = array of Gauss distributed values based on kernel

color = (0,0,0) // output color

for i between 0 and weights.length:
  color += texture.pixelAt(blurUV[i])*weights[i]
```

Listing 3.5: Two-pass Gaussian filter fragment shader

3.7.3 Exporting heightmaps

Obviously, once an interesting heightmap has been generated, it is desirable to save it for future usage. To generate heightmaps that are compatible with the huge variety of tools available, it is necessary to choose a common file format. Research shows that the majority of terrain generation applications supports 8-bit grayscale images using .png file format. It is important to note that 8-bit images allow only a maximum of 256 different height levels. For small heightmaps with equal or less than 512x512 tiles this is more than enough. Each pixel of the grayscale image will be used to store vertex y coordinates. Figure 3.24 depicts

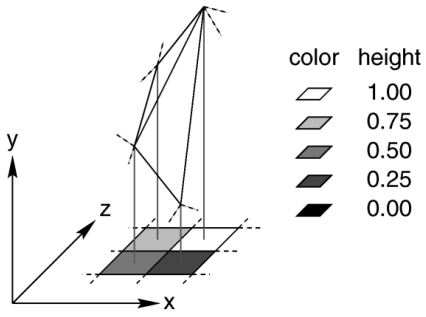

Figure 3.24: Simplified heightmap export process [14]

this process for a 1x1 heighmap. There are some special cases to consider which would result in invalid heightmap exports if not handled properly:

- Heightmaps containing vertices with negative y coordinates
- Heightmaps containing vertices with y coordinates exceeding the upper bound

These problems can be resolved by translating and scaling the heightmap along the y axis in a way that satisfies both requirements. The following pseudo-code shows how this issue was dealt with programmatically:

```
imageWidth = hm.getColumns() + 1
imageHeight = hm.getRows() + 1
image[] = array of length imageWidth*imageHeight

for y between 0 and imageHeight:
  for x between 0 and imageWidth:
    image[y*imageWidth + x] = ((hm.getHeightAt(x,y) - hm.getMinHeight()) / (hm.
    getMaxHeight() - hm.getMinHeight())) * 256
```

Listing 3.6: Heightmap export pseudo-code

3.7.4 Capturing screenshots with stblib

OpenGL provides a function called *glReadPixels*, which copies a specified region of pixels from the video card memory to the RAM. RGB images require width * height * 3 bytes per pixel. Since a *char* requires exactly one byte of space, it is commonly used to represent color channel information. In fact, a library called *stblib* written by Sean T. Barret et al is perfectly capable of saving these kinds of character arrays in various file formats like .jpg or .png and was used for this purpose.

3.7.5 Video recording with ffmpeg

To efficiently record a video, it is necessary to allocate enough space beforehand. Reallocating space during the capturing process leads to noticeable stuttering and is not an option. The following formula was used to calculate the amount of required space:

$$space(width, height, fps, t) = width * height * 3 * fps * t \text{ bytes}$$

By using OpenGL's *glReadPixels* function, it was possible to save each frame into an array of type char. Once all the frames are captured, they need to be encoded using ffmpeg's MPEG2 encoding algorithm. Therefore, each frame has to be converted from RGB to YCbCr color space using ffmpeg library. For some reason, ffmpeg's YCbCr color conversion algorithm flips the frame vertically during conversion. After flipping the frame back, it is ready to be encoded by ffmpeg. Once all frames are encoded, ffmpeg will save the video as .mpg file on the harddisk. The file format .mpg can be played with almost any video player and does not require too much space, due to MPEG2 encoding algorithm provided by ffmpeg.

Chapter 4

Results

4.1 Comparison of hydraulic erosion algorithms

The algorithms being compared are Jacob Olsen's cellular automata-based approach as well as the SPH-based alternative. Both algorithms were applied to predefined heightmap surfaces of different shapes for ten minutes each. The shapes in concern are

- a crooked surface
- a pyramid-like surface
- a hemispheric surface.

Although Olsen's algorithm showed consistent results independently of the shape it was applied to, the results were quite unspectacular. Figures 4.1 to 4.3 depict Olsen's algorithm

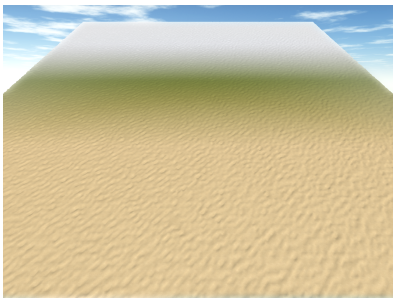

Figure 4.1: Crooked surface eroded by cellular automata-based hydraulic erosion

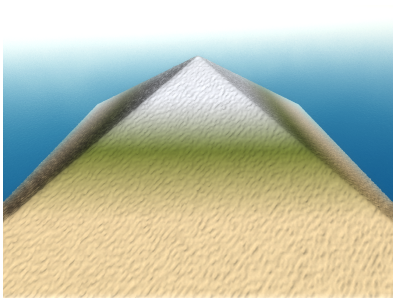

Figure 4.2: Pyramid surface eroded by cellular automata-based hydraulic erosion

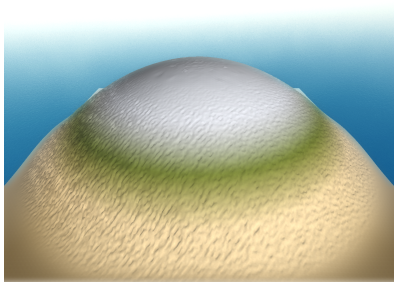

Figure 4.3: Hemispheric surface eroded by cellular automata-based hydraulic erosion

applied to the shapes mentioned before. All ditches measured about the same width and depth and looked very unoriginal. Performing the experiment several times did not bear any different results either. The SPH-based approach on the other hand showed very promising results: Firstly, the shape the algorithm was applied to directly influenced the erosion process. Secondly, the generated ditches had each different widths and depths. Performing the experiment several times resulted in totally different surface transformations. This is due to the fact that the SPH-based approach simulates real particle movement, while Olsen's approach is an abstract model based on cellular automata and does not consider flow velocity. Figures 4.4

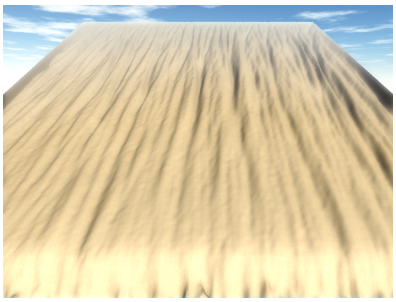

Figure 4.4: Crooked surface eroded by SPH-based hydraulic erosion

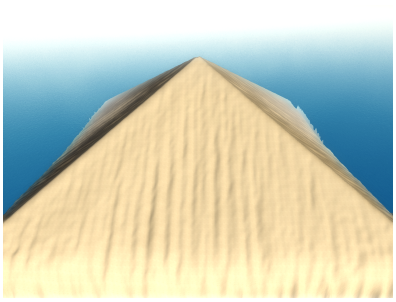

Figure 4.5: Pyramid surface eroded by SPH-based hydraulic erosion

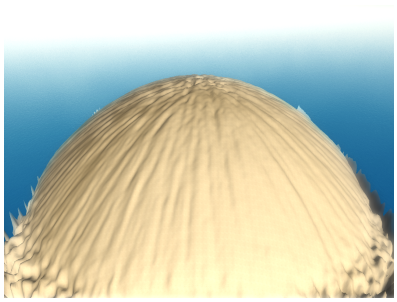

Figure 4.6: Hemispheric surface eroded by SPH-based hydraulic erosion

to 4.6 depict the SPH-based algorithm applied to the shapes mentioned before. Concluding this experiment, it is evident that utilizing SPH simulations for hydraulic erosion algorithms bears very promising results. SPH simulations are capable of modelling fluid flow velocity autonomously. This behavior substantially influences the realism of the hydraulic erosion process, and therefore, SPH simulations should be preferred over cellular automata when it comes to modelling hydraulic erosion.

4.2 Mountain range generation algorithm alterations

The mountain range generation algorithm is a major component of the tectonic plate simulation, as previously described in section 3.6.2. It uses a modified version of Voronoi-based RMP algorithm to procedurally generate mountain ranges. By extracting the mountain range generation portion from the tectonic plate simulation and removing the randomness of tectonic plate movement, it was possible to reliably conduct an experiment. The goal for this experiment was to determine if slight alterations in the mountain range generation code, namely dynamic instead of static areal height assignments, would result in higher realism. Figure 4.7 might help to understand what the experiment is all about. Instead of raising the whole bordered area, the goal is to raise each colored area inside the border individually. The experiment consisted of two iterations. To determine if the alterations had any effect, a control experiment was performed using the unaltered code as depicted in fig. 4.8. Figure 4.9 shows the final result after the alterations. Unfortunately, the effect of the alterations is very subtle. While the control experiment showed no variation in mountain top elevation, the altered version of the algorithm produced mountain tops that slightly vary in height. Even though the effects were subtle, the alterations were still merged into the original mountain range generation code since they had no impact performance-wise while still contributing to the realism of the scene.

4.3 Thermal erosion algorithm variations

Fast terrain generation algorithms such as fault algorithm produce very rough and spiky terrain features. Therefore, terrain generated by fault algorithm has to be smoothed before usage. For this purpose, thermal erosion algorithm can be used. It is capable of removing spikes and smoothing the terrain in general. Based on the roughness of the terrain, the following two variations of thermal erosion should be considered:

- Height-aware thermal erosion

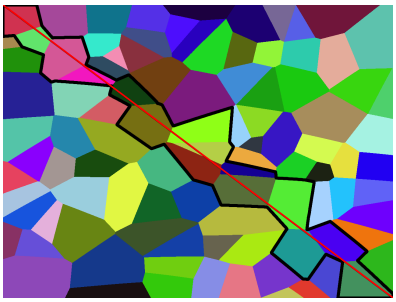

Figure 4.7: Visualization of the experiment

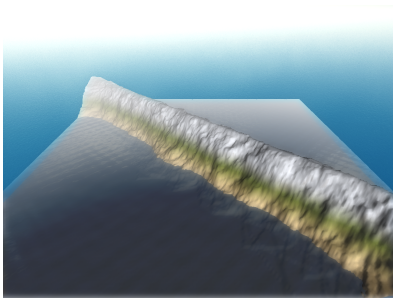

Figure 4.8: Each affected area raised by the same amount

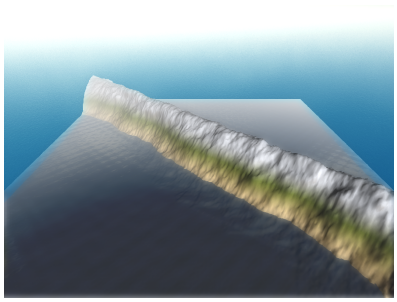

Figure 4.9: Each affected area raised by a random amount

- Constant thermal erosion

In general, height-aware thermal erosion is a very good choice for almost any terrain roughness. It considers material hardness based on height, which results in very realistic terrain transformations. However, experiments show that for very rough and spiky terrain, constant thermal erosion is preferred to height-aware thermal erosion since the latter causes heavy loss of detail in lower elevations and keeps spikes in higher elevations unaffected. These effects can be witnessed in figs. 4.10 to 4.12.

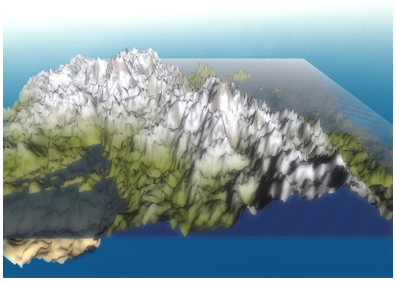

Figure 4.10: Heightmap generated with Fault algorithm

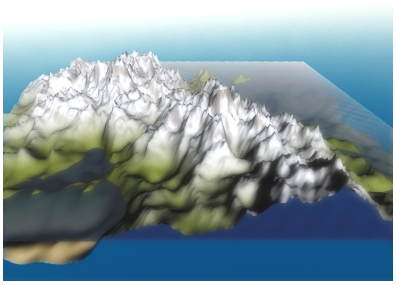

Figure 4.11: Height-aware thermal erosion

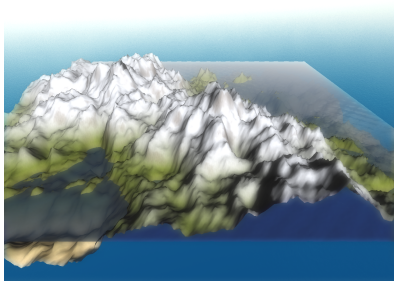

Figure 4.12: Constant thermal erosion

4.4 Graph-based and Voronoi-based RMP performance analysis

The goal of this performance analysis is to compare two totally different implementations of RMP algorithm. The graph-based algorithm consists of several expensive steps including a minimal-cycle-search as well as a depth-first-search. These calculations solely take place on the CPU. The Voronoi-based algorithm takes full advantage of the GPU by generating Voronoi diagrams using the method described in section 3.3.2. Both algorithms were given three tasks which required them to generate a single mountain of increasing complexity in the middle of a blank heightmap. The level of mountain complexity was determined by the amount of polygons generated in each iteration. The amount of polygons generated by the graph-based RMP algorithm is influenced by the number of cuts placed per iteration. Using Euler's formula also known as the *Lazy Carterer's Sequence* [11], it is possible to calculate the

maximum number of polygons p generated by placing n cuts per iteration:

$$p = \frac{n^2 + n + 2}{2} \tag{4.1}$$

The amount of polygons generated by the Voronoi-based RMP algorithm is directly influenced by the number of cones placed per iteration. Each task consisted of 100 iterations to ensure measurability. By repeating each task ten times, it was possible to reduce the noise to a neglectable amount. The experiment was performed on a MacBook Pro (2.8GHz Intel Core i5 processor, Intel Iris onboard graphics). Tables 4.1 and 4.2 list the recorded results. The

Table 4.1: Graph-based RMP performance analysis

	Task 1	Task 2	Task 3
# of cuts	3	4	5
max. # of polygons	7	11	16
# of iterations	100	100	100
sample 1	2.93s	45.47s	5891.85s
sample 2	3.01s	35.41s	5921.50s
sample 3	2.80s	43.66s	5929.62s
sample 4	3.01s	39.26s	5962.49s
sample 5	3.02s	59.81s	5941.39s
sample 6	2.98s	53.29s	5957.13s
sample 7	3.05s	48.87s	5942.96s
sample 8	2.93s	49.97s	5965.95s
sample 9	3.02s	49.33s	5889.50s
sample 10	3.00s	42.22s	5971.09s
μ	2.975s	46.729s	5937.348s
σ^2	$5.27 \cdot 10^{-3}$s	50.139s	855.362s
σ	$7.26 \cdot 10^{-2}$s	7.081s	29.247s

results clearly show the benefits of Voronoi-based RMP. Voronoi-based RMP algorithm scales much better than graph-based RMP algorithm. While graph-based RMP algorithm already slows down significantly, generating eleven randomly distributed polygons, the Voronoi-based alternative seems to perform about the same regardless of the number of polygons to be generated. This behavior is very valuable for real-time applications, and therefore, the Voronoi-based RMP algorithm was chosen over the graph-based alternative. With the graph-based RMP algorithm out of the picture, it would also be interesting to find the limits of the Voronoi-based alternative. For this reason, another experiment was conducted. This time up to 100000 polygons were generated per iteration, and the duration t was noted down after each task consisting of 100 iterations was completed. Figure 4.13 shows a plot of the data recorded during this experiment. The plot clearly shows that the Voronoi-based RMP algorithm is capable to generate up to 1000 polygons per iteration in real time. For real-time terrain generation, fast generation of huge amounts of randomly distributed polygons is a huge deal. Not only is it possible to control the spread of a mountain with high precision, it is also possible to generate more detailed mountains in general.

Table 4.2: Voronoi-based RMP performance analysis

	Task 1	Task 2	Task 3
# of cones	7	11	16
# of polygons	7	11	16
# of iterations	100	100	100
sample 1	2.04s	2.06s	1.99s
sample 2	1.99s	1.96s	2.06s
sample 3	1.96s	1.94s	2.04s
sample 4	2.01s	1.97s	2.01s
sample 5	1.99s	2.00s	1.99s
sample 6	1.98s	1.97s	2.10s
sample 7	2.03s	2.19s	2.03s
sample 8	1.98s	1.96s	1.99s
sample 9	1.95s	1.97s	2.02s
sample 10	1.96s	2.02s	2.02s
μ	1.989s	2.004s	2.025s
σ^2	$8.99 \cdot 10^{-4}$s	$5.49 \cdot 10^{-3}$s	$1.23 \cdot 10^{-3}$s
σ	$3.00 \cdot 10^{-2}$s	$7.41 \cdot 10^{-2}$s	$3.5 \cdot 10^{-2}$s
speedup	1.496	23.318	2932.024

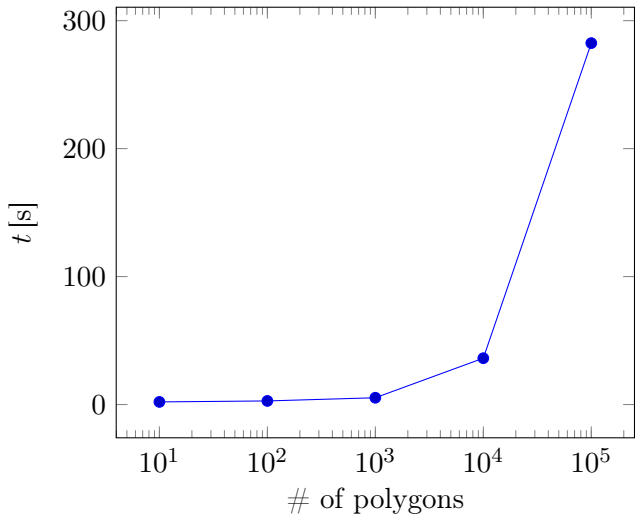

Figure 4.13: Voronoi performance stress test

Chapter 5

Conclusion & Future Work

This bachelor thesis featured some well-known synthetic terrain generation techniques and showcased how geology-based implementations are far superior in terms of configurability and realism. A way to combine synthetic and geology-based algorithms was proposed, which ultimately led to the development of a simplified real-time capable tectonic plate simulation. Besides terrain generation, two very specific terrain erosion algorithms were implemented, experimented with and optimized for real-time usage. Finally, a few simple but effective visualization techniques were presented that could be adapted to be used in any arbitrary environment.

Of course, there are some areas that could potentially be improved by more sophisticated implementations which take full advantage of the hardware available. Heightmap transformations, for example, are currently CPU-based, using an array-like two-dimensional data structure. GPU-based texture data structures, however, would be considerably faster for this type of application. Unfortunately, the changes required are non-trivial and a complete overhaul of the erosion algorithms as well as the tectonic plate simulation would be necessary.

The SPH-based hydraulic erosion simulation is currently bound by CPU as well. Due to this computational bottleneck, only a very limited amount of about 3000 simultaneously moving particles can be simulated. GPUs would be much more capable to handle the concurrent computations required for simulating fluid particle motion. Existing GPU-based SPH simulations like GPUSPH [1] are perfectly capable to simulate up to 50 million particles. Increasing the quantity of particles would imply a reduction in particle size and enable a more fine-grained hydraulic erosion process.

The most noticeable types of terrain erosion are thermal and hydraulic erosion. That is why both of which were implemented in the course of this bachelor thesis. Nevertheless, also other types of erosion such as wind and glacial erosion have a significant impact on landscape and could be implemented in future.

Although the presented tectonic plate simulation is perfectly capable to model plate movements and collisions, it only accounts for a quite limited amount of phenomena involved in plate tectonics. To give an example, subducting and diverging plates are not properly supported yet. Considering the limitations of two dimensions, it is very difficult to realistically simulate subducting plates, namely plates overlapping each other, using the Box2D physics engine. A more sophisticated physics engine could make use of all three dimensions and would be able to handle scenarios like these. Furthermore, plates are currently only split during the initialization process of the simulation. A different approach would be to continuously split plates based on force thresholds. Naturally occurring, devastating catastrophes like volcanic eruptions and earthquakes should not be disregarded either. Further research, in terms of how

these phenomena could be simulated in a virtual environment is necessary. Obviously, there is always a trade-off between performance and level of detail to be considered when deciding to use more sophisticated models instead of simple, yet effective alternatives.

To improve the visual appearance of the scenery, it would be possible to procedurally generate textures instead of using the same seamless textures over and over. Although barely noticeable, under close inspection, recurring texture patterns can be spotted. Shadows are currently not supported. Frequently changing meshes like heightmaps are not well suited for shadow mapping, since it would be too expensive to generate a new shadow map each time the heightmap is altered. A possible solution to this problem would be generating shadow maps only occasionally, e.g., when taking screenshots. Water waves are currently implemented by combining sine waves with OpenSimplexNoise to continuously alter an additional semi-transparent heightmap consisting of 256x256 tiles and applying an animated diffuse water texture. To improve the performance of water waves, it would make sense to remove the additional water heightmap and use an animated specular, normal and diffuse texture to mimic water waves on a single tile instead. At the moment, only the sky is reflected in the water. To reflect the landscape as well, a post-processing step would be necessary. In this step, the part of the landscape that is visible in the reflection would have to be vertically mirrored and cropped afterwards using the stencil buffer.

Bibliography

[1] Gioseppe Bilotta Alexis Hault and Robert A. Dalrymple. Gpusph, 2008. [Online; accessed September 4, 2016].

[2] David Allen. cdrift. [Online; accessed August 19, 2016].

[3] Alex Darcy. Shallow water simulation with a finite volume scheme. [Online; accessed September 14, 2016].

[4] Trevor Dixon. Shallow water implementation (requires three.js). [Online; accessed September 14, 2016].

[5] Alain Fournier, Don Fussell, and Loren Carpenter. Computer Rendering of Stochastic Models. *Commun. ACM*, 25(6):371–384, June 1982.

[6] K. Raiyan Kamal and Yusuf Sarwar Uddin. Parametrically Controlled Terrain Generation. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, GRAPHITE '07, pages 17–23, New York, NY, USA, 2007. ACM.

[7] Robert Krten. Generating Realistic Terrain. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 1994.

[8] Saeed Mahani. Physics-based simulation final project. [Online; accessed September 14, 2016].

[9] Xing Mei, Philippe Decaudin, and Bao-Gang Hu. Fast Hydraulic Erosion Simulation and Visualization on GPU. In Marc Alexa, Steven J. Gortler, and Tao Ju, editors, *PG '07 - 15th Pacific Conference on Computer Graphics and Applications*, Pacific Graphics 2007, pages 47–56, Maui, United States, October 2007. IEEE.

[10] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

[11] Thomas L. Moore. Using euler's formula to solve plane separation problems. *The College Mathematics Journal*, 22(2):125–130, 1991.

[12] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The Synthesis and Rendering of Eroded Fractal Terrains. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, pages 41–50, New York, NY, USA, 1989. ACM.

[13] Jacob Olsen. Realtime Procedural Terrain Generation: Realtime Synthesis of Eroded Fractal Terrain for Use in Computer Games. 2004.

[14] POV-Ray. Height field. [Online; accessed July 29, 2016].

[15] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 267–276, New York, NY, USA, 2002. ACM.

[16] M. Shimrat. Algorithm 112: Position of Point Relative to Polygon. *Commun. ACM*, 5(8):434–, August 1962.

[17] Lauri Viitanen. Physically Based Terrain Generation: Procedural Heightmap Generation Using Plate Tectonics. B.S. Thesis, 2012.

[18] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[19] W. Randolph Franklin (WRF). PNPOLY - Point Inclusion in Polygon Test, 1994. [Online; accessed August 28, 2016].