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, June 6, 2016

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam odio lorem, fermentum nec lectus sit amet, vehicula tristique arcu. Nunc id leo volutpat, facilisis sem vel, dapibus felis. Ut vehicula, leo sed feugiat egestas, enim velit rhoncus risus, a congue libero massa a enim. Aenean purus neque, tempor a dictum vitae, luctus eu sapien. Nam eget libero accumsan, placerat quam eu, congue turpis. Pellentesque tincidunt vel mi et mattis. Maecenas erat ligula, pretium ac nunc sit amet, molestie sollicitudin massa. Nunc mauris elit, iaculis et euismod vel, congue nec felis. Ut nec molestie mauris, eu gravida odio. Sed in ligula nulla.

Keywords keyword1, keyword2.

Contents

List of Figures

List of Tables

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

1.1 Heightmaps

A heightmap is a surface and its height (e.g. its y coordinate) is well-defined for any coordinate tuple (x,z). Due to this, it is easily possible to export heightmaps as grayscale image files (e.g. dark portions = low elevation, light portions = high elevation). Heightmaps consist of a predefined number of tiles. These tiles again consist of 2 triangles each. The resolution of a heightmap is the number of tiles it consists of. Usually heightmaps use square dimensions, meaning the number of columns and rows of tiles is the same.

Chapter 2

Related Work

Kamal and Uddin evaluated various terrain generation algorithms like Diamond-Square-Algorithm and Fault-Algorithm and came up with their own algorithm called Repeated Magnification and Probing [2]. Thermal and hydraulic erosion algorithms were first described by Musgrave [6]. Based on Musgraves work, Olsen implemented his own version of thermal and hydraulic erosion algorithm and also improved them performance-wise to make them available for realtime-applications [7]. Vitanen describes the theoretical foundation of plate tectonics, compares several existing applications and ultimately implements his own tectonic plate simulation [10].

Chapter 3

Methods

3.1 Diamond-Square algorithm

This algorithm was first introduced by Fournier et al in 1982 [1]. 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 1D-landscape. Figures 3.1 to 3.3 depict the first 2 iterations of MPD algorithm.

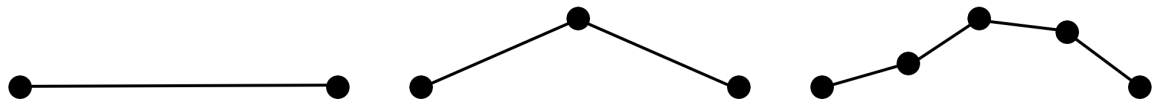

Figure 3.1: MPD init Figure 3.2: MPD iteration 1 Figure 3.3: MPD iteration 2

To take MPD algorithm to the next level Diamond-Square (DS) algorithm allows to procedurally generate 2D-landscape. During initialization, DS 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 DS 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
```

```
14   avgHeightTop = ... // average height of top diamond corners
15   r = roughness * (randomFloat(2)-1)
16   hm.setHeightAt(midpointColumn, top, avgHeightTop+r)
17   ... // similar steps for left, right and bottom diamond
18
19 procedure ds(hm, roughness, left, right, top, bottom):
20   diamondStep(hm, roughness, left, right, top, bottom)
21   squareStep(hm, roughtness, left, right, top, bottom)
22
23 procedure perform(hm, roughness):
24   columns = hm.getColumns()
25   rows = hm.getRows()
26
27   // initialize heightmap corners
28   hm.setHeightAt(0, 0, roughness)
29   hm.setHeightAt(columns, 0, roughness)
30   hm.setHeightAt(0, rows, roughness)
31   hm.setHeightAt(columns, rows, roughness)
32
33   while columns > 1:
34     for row = 0; row < hm.getRows(); row += rows:
35       for column = 0; column < hm.getColumns(); column += columns:
36         ds(hm, roughness, column, column+columns, row, row+rows)
37     roughness /= 2.0
38     columns /= 2
39     rows /= 2
```

Listing 3.1: DS pseudocode

Figures 3.4 to 3.8 depict the DS 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 [3]. He proposed to randomly draw a line across the heightmap and split the landscape into two regions. By raising one and lowering the other region 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 get 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:

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

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

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

Let vectors $\mathbf{a}$ and $\mathbf{b}$ describe a random cutting line. Then a vertex pointed to by vector $\mathbf{p}$ will be raised in height if $raise((\mathbf{p}-\mathbf{a}) \times (\mathbf{b}-\mathbf{a}))$ equals 1 and lowered otherwise. Figures 3.9

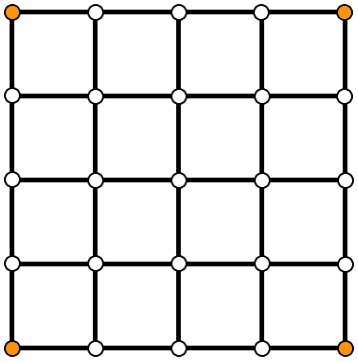

Figure 3.4: DS init

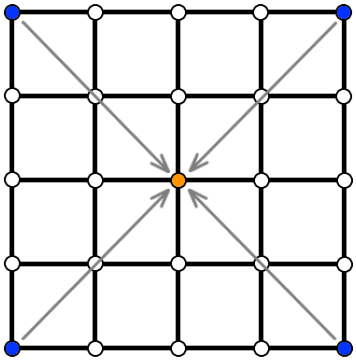

Figure 3.5: DS 1

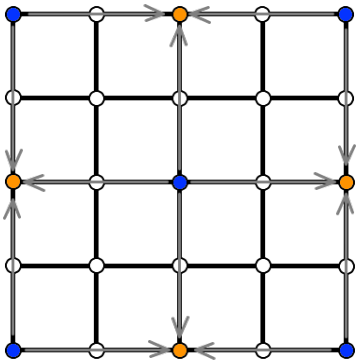

Figure 3.6: DS 2

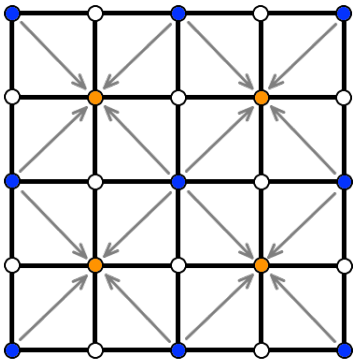

Figure 3.7: DS 3

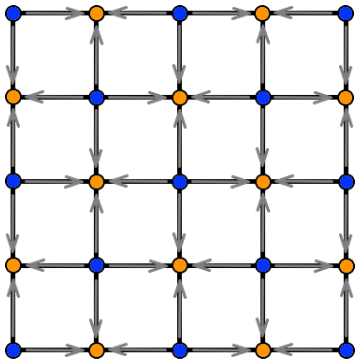

Figure 3.8: DS 4

to 3.12 will depict the first 3 iterations of Fault algorithm using different shades of green. Light green represents high and dark green low elevation respectively.

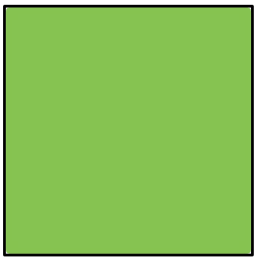

Figure 3.9: Fault init

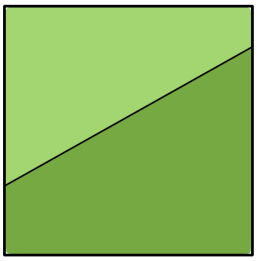

Figure 3.10: Fault 1

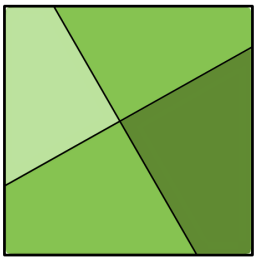

Figure 3.11: Fault 2

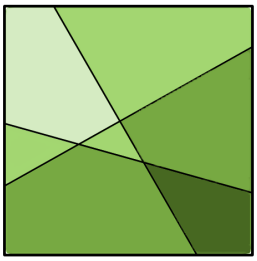

Figure 3.12: Fault 3

3.3 RMP algorithm

RMP stands for "Repeated Magnification and Probing", was created by Kamal and Uddin [2] 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 how these regions are obtained is not described in Kamal and Uddin's paper [2].

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 4 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, 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 the extract a vector's x- and y-coordinates:

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

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 $\mathbf{i}$ and $\mathbf{j}$ be two intersections on a line defined by origin $\mathbf{a}$ and direction $\mathbf{d}$ as seen in eq. (3.1). To determine which intersection is closer to the start of the line one has to substitute $\mathbf{p}$ in eq. (3.1) with $\mathbf{i}$ and $\mathbf{j}$ 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 2 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 start point of the 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 [9] and later implemented in C by W. Randolph Franklin, 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 realtime 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" [11]. This implementation relies on OpenGLs depth testing 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 3 Voronoi diagrams using different amounts of cones.

Figure 3.13: 10 cones

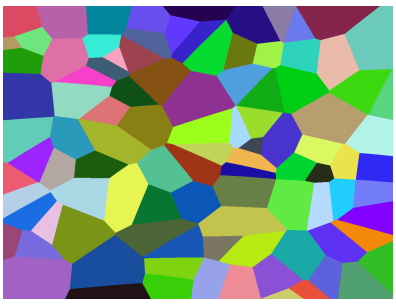

Figure 3.14: 100 cones

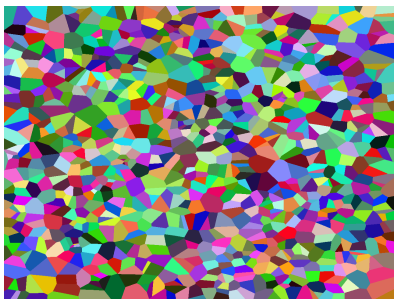

Figure 3.15: 1000 cones

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 [7]. 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 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 like 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.

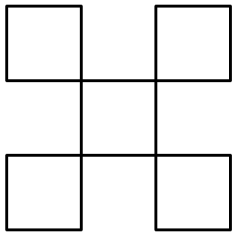

Figure 3.16: Rotated Von Neumann neighborhood

3.5 Hydraulic erosion algorithm

3.5.1 Implementation based on previous research

In my reasearch I came across two different papers that described how to model hydraulic erosion algorithmically. The first one [7] 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 gets 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 4 steps:

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

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.

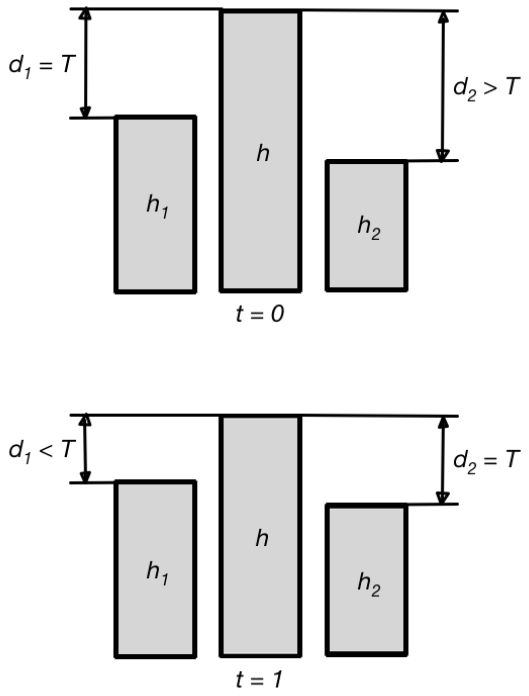

Figure 3.17: Thermal erosion

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. Olsen described a cellular automaton based approach as depicted in fig. 3.18 which is capable of simulating water flow. There are 2 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 [4] 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 results in a generally 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 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. $\Delta t = 10\text{ms}$) that one cannot even notice any

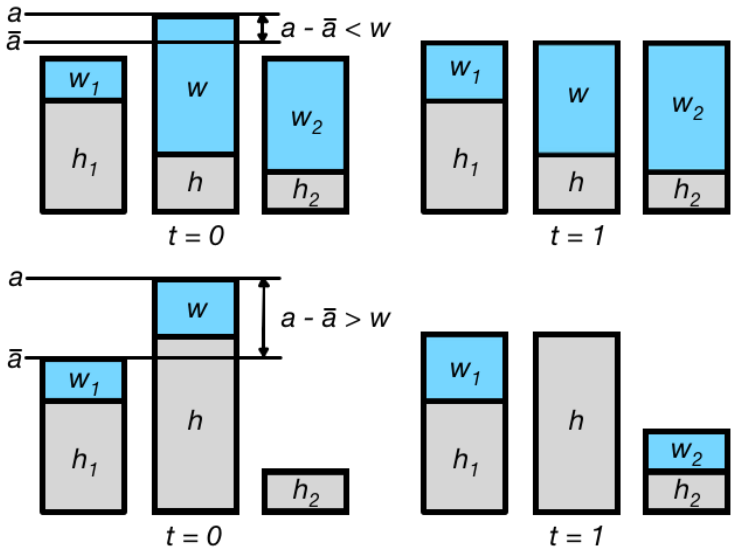

Figure 3.18: Hydraulic erosion

difference in water flow for a couple of minutes. For a realtime 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 trying out 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 like 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, 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 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 spending a considerable amount of time I ultimately was 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 like hardware incompatibility or realtime incapability. Last but not least I came across a GitHub project by Saeed Mahani.

He wrote a basic SPH simulation featuring a box which gets filled up by fluid particles while obeying the laws of physics. The project was realtime capable and had no special hardware requirements. However, several alterations to the SPH simulation had to take place in order to make it usable for this project. 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, 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 behaviour it is necessary to continuously check if a particle is touching the surface. A fast triangle-ray intersection test by Möller and Trumbore [5] was used to check if a particle is below the heightmap surface or not. If the intersection test indeed is 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 4 normals in close proximity to the impact point. These normals are normalized vectors. To get the interpolated normal from these 4 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 utilize it 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 in a certain height at random positions on the xz-plane. The user can observe the simulation in realtime 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. Additionally they should be kept in motion to simulate magma flowing underneath the surface. To model this non-trivial behaviour I decided to use a physics engine. Box2D, a 2D physics engine, supports convex polygons with up to 8 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 like "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. The tectonic plate simulation will consist of dynamic as well as kinematic bodies. The dynamic bodies will represent the tectonic plates and the kinematic bodies will be 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 like current position, angle, velocity, mass, etc. To describe the shape of a Box2D body, fixtures will be used. 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 like rectangles and circles as well as convex polygons with a maximum of 8 vertices. The tectonic plate simulation will be initialized with 5 rectangle fixtures. 4 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 therefore 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 8 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:
```

```
6   cutAngle = atan((v.y-rayCenter.y)/(v.x-rayCenter.x))-rayAngle
7   if cutAngle < -PI:
8     cutAngle += 2*PI
9   if cutAngle > 0 and cutAngle <= PI:
10    // vertex v belongs to polygon1
11  else:
12    // 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 are 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 which do not have 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 callbacks mentioned before. Using these points one can draw an imaginary line onto the heightmap. The goal would be to procedurally generate a mountain range along this imaginary line. This is where an advanced version of RMP algorithm comes into place. This version of RMP will also use a Voronoi diagram to generate arbitrary polygons. Different from the previous RMP implementation this time there will be two coordinate pairs instead of a single one. Between these coordinates Bresenham line algorithm will be performed and all colors passed 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 between two colliding tectonic plates.

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 offers a function "mix" that allows to linearely 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 similarily...
```

Listing 3.3: Fragment shader texture chooser pseudocode

Up to now only the height information is used to determine which texture should be applied on the terrain. Of course this model does not look too realistic since the slope information has not been considered so far. Grass does not grow on steep slopes. Same goes for snow. Snow will only be at rest 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 regions.

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 [8].

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 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.

Table 3.1: Empty table

3.7.4 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:

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

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

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam odio lorem, fermentum nec lectus sit amet, vehicula tristique arcu. Nunc id leo volutpat, facilisis sem vel, dapibus felis. Ut vehicula, leo sed feugiat egestas, enim velit rhoncus risus, a congue libero massa a enim.

Figure 4.1: Empty figure

Chapter 5

Conclusion & Future Work

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Bibliography

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

[2] 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.

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

[4] 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.

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

[6] 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.

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

[8] 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.

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

[10] Laurii Vitanen. Physically Based Terrain Generation: Procedural Heightmap Generation Using Plate Tectonics. B.S. Thesis, 2012.

[11] 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.