Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Thomas Lund Mathisen

Multigrid feature-based terrain generation with erosion

Master's thesis in Master of Science in Informatics
Supervisor: Theoharis Theoharis
July 2021

NTNU
Norwegian University of
Science and Technology

Thomas Lund Mathisen

Multigrid feature-based terrain generation with erosion

Master's thesis in Master of Science in Informatics
Supervisor: Theoharis Theoharis
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Acknowledgements

I want to thank the Norwegian University of Science and Technology for enabling me to explore this research field and especially professor Theoharis Theoharis for supervising. A big thank you to Maria for giving me structure and feedback over the last year and listening to me pointing out *pretty* and *fake*-looking terrains wherever we go. Thanks to Hanne for an unlimited amount of coffee. I would also like to thank Øystein and Kristine for taking the time to understand this thesis and proofreading it. Finally, I want to thank Lilis, the corona-cat, for the mental support.

Abstract

Landscape generation has been a field of research for almost thirty years and is a fundamental part of many applications. Despite this, there are still a lot of problems left to be solved and designers are forced to manually design terrains as they cannot realize their ideas with the current tools. In this thesis, we explore two problems related to feature-based terrain generation: the ability to create fine-scale and large-scale terrain features and mitigating the repetitiveness of noise used in "Feature based terrain generation using diffusion equation" by Hnaidi *et al.* Our contribution is our method with four novelties as well as an implementation of this method integrated with Unity 3D. We combine feature and noise-based terrain generation with hydraulic erosion by controlling the noise generation and erosion simulation with diffused parameters. The first novelty is using multiple diffusion equations enabling the user to generate smoother terrain and generate both fine- and large-scale features. Our second novelty is simulating erosion on multiple grids to efficiently create erosive features of different scales. Our third novelty is diffusing erosion parameters to constrain the erosion by controlling the rain and hardness of the terrain. Our last novelty is creating diffused noise warping without creating any artifacts in the terrain. These novelties make it easier to generate terrain with fine- and large-scale features and mitigate a lot of the repetitiveness of the noise.

Sammendrag

Landskap generering har vært et forsknings-felt i nesten 30 år og er fundamentalt i mange applikasjoner. Til tross for dette er det fortsatt mange uløste problemer, og designere er nødt til å designe terreng manuelt for å realisere ideene sine, på grunn av begrensninger i dagens verktøy. I denne oppgaven utforsker vi to problemer relatert til trekk-basert terreng generering: muligheten til å lage terreng med både fin-skala og stor-skala trekk og å minske den gjentakende effekten til støy brukt i "Feature based terrain generation using diffusion equation" av Hnaidi *et al.* Vårt bidrag er metoden vår med fire nyskapninger og implementasjonen vår av denne metoden i Unity 3D. Vi kombinerer trekk- og støy-basert terreng generering med hydraulisk erosjon ved å kontrollere støy-generering og erosjons-simuleringen med diffuse parametere. Vår første nyskapning er å bruke flere diffusjons-likninger slik at brukeren kan generere jevnere terreng med både fin-skala og stor-skala trekk. Vår andre nyskapning er å simulere erosjon med flere rutenett, for å effektivt simulere erosjon i ulik skala. Vår tredje nyskapning er å bruke *diffuse* parametere for å begrense erosjon ved å kontrollere regn og hardheten på terrenget. Vår siste nyskapning er å generere *diffust* fordreid støy uten å skape unaturligheter i terrenget. Disse nyskapningene gjør det enklere å generere terreng med fin-skala og stor-skala trekk og minsker mye av den gjentakende effekten til støy.

Contents

List of Figures

Glossary

artifact	An unnatural and unwanted effect.
asset	A finished component in Unity 3D which one can easily add to a scene.
CPU	Central Processing Unit.
GPU	Graphics Processing Unit.
grid level	A grid level is a stage in the multigrid solver. In Figure 3.3 each row is a multigrid level. The size of the grid is equal to 2^x where x is the grid level.
kernel	A function in a compute shader that is called from the CPU. For instance, a kernel is often run for every cell in a grid, every pixel of an image or every triangle in a mesh.
LoD	Level of Detail.
scene	We use *scene* in a similar manner to how it is used in Unity 3D. A scene is a three-dimensional space in which objects can be positioned and rendered.
spline	A spline is a piecewise polynomial curve.
terrain feature	A larger part of a terrain for example a hill, valley, mountain, river or crevasse.

Chapter 1

Introduction

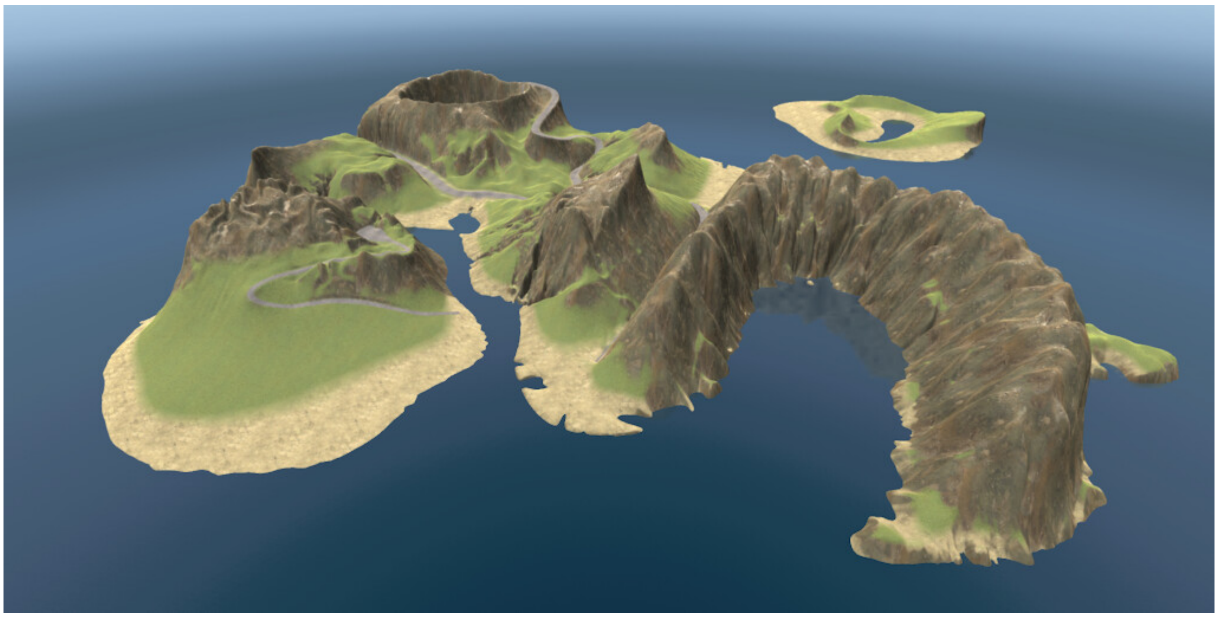


Figure 1.1: A terrain generated with our method

1.1 Motivation and background

Generated landscapes and worlds are more and more used as our lives become more digital. Digital landscapes are used in many computer games and have recently had a prominent role in the movie The Lion King (2019) [2] and the TV show The Mandalorian [3]. The Lion King (2019) was created and filmed in a virtual reality video game. The Mandalorian was filmed entirely in studios and the landscape added digitally afterwards. Already in 2004 the game World of Warcraft [4] featured a huge world with different biomes and terrain features. The premise of the game is to defeat enemies and complete quests to gain levels and abilities until you can defeat the strongest foes. You team up with and fight other players while exploring the world. The game was designed with different zones with different atmospheres, vegetation, landscape features, and creatures. One of the motivations to gain levels was to be able to freely explore the world and all its wonders. This exploration aspect was also used in the game No Man's Sky [5] from 2016. In this survival game, a player can repair their spacecraft to travel between planets to find resources, currency and new species. All of these planets are entirely procedurally generated which means there are about 18 quintillion unique worlds [6] the player could explore. Such exploration games could not exist without the research done in this field.

Research on terrain generation has been explored since the 90s, and there are still problems left to be solved. Already in 1982 two algorithms for large-scale terrain generation were proposed [7]. Since then the field has evolved to involve complex simulations of natural phenomena and machine learning. Though machine learning can generate very realistic terrains these methods still suffer from one of the same problems as the first methods did - user control [7]. User control is how easy it is to change the terrain and form it in a specific way. Hnaidi *et al.* created a method for generating terrain with good user control where the user controls the terrain by drawing splines [1]. According to Galin *et al.* [7], their methods do not look that realistic and it can be tedious to add details. Generating terrain with simulations of natural phenomena looks a lot more realistic. These methods do however also suffer from a lack of user control and they are very performance expensive. Because of the lack of user control as well as other problems designers are forced to revert back to manual editing of the terrain to realize their intentions [7].

1.2 Objective

In this thesis, we explore the possibilities of designing terrain with good user control and interactivity by utilizing feature-based design using splines. We focus on improving the paper "Feature based terrain generation using diffusion equation" [1] by Hnaidi *et al.* to make it easier to create features of small and large scale and mitigate the repetitiveness of the noise. To mitigate the repetitiveness of noise we explore warping the noise and simulating erosion.

Our contribution is a new method for generating terrain with an implementation of terrain generation using splines, diffusion, and erosion simulation in Unity 3D [8] that can be used for further research and terrain generation.

As our objective is to explore we define performance and memory optimizations as out of scope for this thesis.

1.3 Research questions

RQ1) How can one design a consistent terrain with a combination of fine-scale and large-scale features?
RQ2) How can one mitigate the repetitiveness of the noise added in feature-based terrain created with diffusion?

Chapter 2

Background and related Work

In order to answer our research questions we first need to look at the different methods used to generate terrain and why answering these questions contributes to the field. In this chapter, we look at four generalized methods which cover most of the terrain generation methods used and the different data structures for representing the terrain. Then we look at previous papers describing two terrain generation methods essential to this paper: feature-based terrain using diffusion and erosion. Finally, we briefly explain the Graphics Processing Unit (GPU); how the GPU compares to the Central Processing Unit (CPU), how one programs the GPU and some new features introduced in recent years.

2.1 Terrain generation methods

Today there are many methods for generating terrain, several of which are discussed in the recent paper by Galin *et al.* [7]. We give a quick overview of some of the methods, but for a more general overview we refer the reader to the paper "A Review of Digital Terrain Modeling" [7].

2.1.1 Noise-based generation

Noise-based generation is a method suitable for creating large-scale terrain [7]. Any noise-based generation relies on a smooth noise function such as improved Perlin noise [9] or Worley noise [10]. These types of functions are extensively researched [10]–[15] for use in many fields including terrain modeling [16]. Different layers of noise are calculated on the grid with different amplitude and frequencies and added to create the final terrain. A common function for creating terrain using noise where S is a smooth noise function and p is a point:

$$N(p) = \sum_{k=0}^{n} A_k S(\phi_k p) \tag{2.1}$$

where A_k is the amplitude for each layer, n denotes the number of layers and ϕ_k is the frequency of each layer. This results in basic terrain types, however, this can be adapted to create more complex forms. Carpentier *et al.* [14] proposed a more complex function which uses warping. Warping is mapping the point p in a non-linear manner before sampling the point with the noise function S. They proposed a more general function using three transformation functions T_{in}, T_{pre} and T_{post}.

$$N(p) = T_{post}(\sum_{k=0}^{n} A_k T_{in}(S(T_{pre}(\phi_k p)))) \tag{2.2}$$

These three transformations can be used independently or combined to recreate well-known noise functions. One can by defining T_{pre} as a non-linear mapping function create warped noise. T_{pre} can be defined as $T_{pre}(h) = 1 - |h|$ to create the well-known ridge noise. T_{post} can be used to make hills and valleys steeper and the area in between flatter. In their paper Carpentier *et al.* [14] used these transformations to create "erosive noise" which they designed to emulate the effects of hydraulic erosion. Carpentier *et al.* state that "...defining such a function is probably as much an art as it is a science". Though noise-based generation methods work well for generating infinite terrain or randomized terrain, they are difficult to control and often overly reliant on noise functions [7].

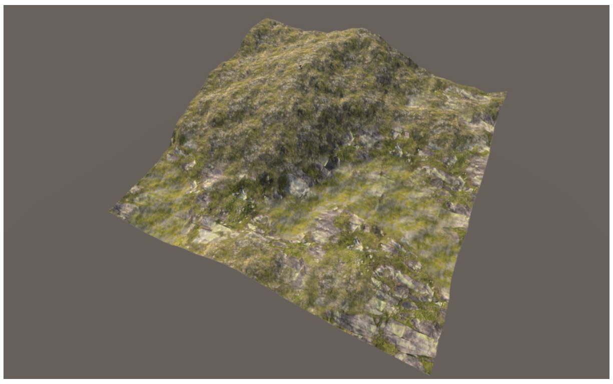


Figure 2.1: Noise-based terrain using Equation 2.1

2.1.2 Feature-based terrain generation

Feature-based terrain generation methods generate terrain from a set of parameters called features. These features could be as simple as a point in space representing a mountain top [17] or as complicated as the statistical measurements of mountains [18]. feature-based methods generate terrain around these features and depending on the method often add noise [1] or erosion [18]. When features define the height of the terrain at certain points in the terrain both the Djikstra shortest path algorithm [17] and solving a diffusion equation [1] has been used to fill in the rest of the terrain smoothly. Génevaux *et al.* published a paper [19] generating terrains with complete river networks from a contour and partial river sketches.

Feature-based methods give great user control as the input is closely related to the resulting terrain. By moving a mountain top feature the entire mountain is moved and the surrounding terrain adjusts to the change. This gives designers great control over the terrain and let them create the terrain they envision. When designing feature-based terrains one can iteratively improve the terrain by adding more and more features to control the generation. The drawback with this method is that adding fine details to the terrain could take a lot of time. The method can not generate terrains of infinite size like noise-based generation can. It can not be as realistic as simulated and example-based terrains either. Feature-based terrains are preferable when the designer needs a lot of control over the terrain and is designing a static world.

2.1.3 Terrain process simulation

Terrain process simulation is used when creating natural terrain by simulating natural phenomena that affect the terrain like tectonic-plates movement, thermal erosion, and hydraulic erosion. These effects can create natural-looking terrain features. The drawback with this technique is mainly the time these simulations take as well as a lack of user control. A simulation is often based on initial settings which are all the user can change and running a simulation can take from a second to several hours [7]. We discuss simulating erosion further in section 2.4.

2.1.4 Example-based terrain generation

Example-based terrains utilize machine learning to generate terrains. The results of the technique are heavily dependent on the data the machines are trained on and will create terrains that are similar to that data. This method can provide moderate user control where users can provide sketches of terrain or heightmaps using different labels. For this to be possible there has to be a significant amount of examples using sketches or labels which are often labeled by hand. Guérin *et al.* [20] automated the labeling process by analyzing real terrain data and extracting curves and other data which they used as input to the algorithm. They generate realistic terrains with erosive features in less than a second with a synthesizer. We rendered one of their results in Figure 2.2. The results are limited as they need to train one terrain synthesizer for each type of terrain and the designer needs to learn to draw input sketches in a way the synthesizer understands. Example-based methods are limited by their training and input data and do not always generate realistic terrains. This is an under-explored part of terrain generation [7] with the potential to generate large realistic terrains in seconds.

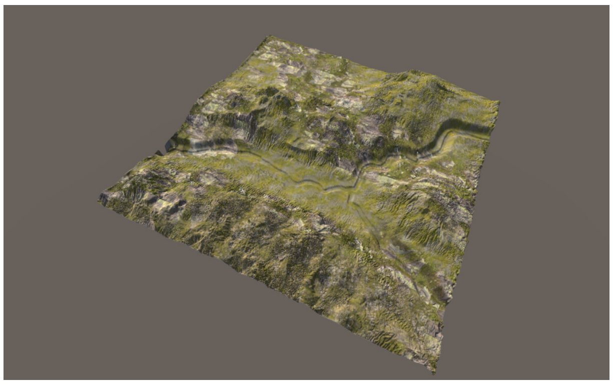


Figure 2.2: Example-based terrain from Guérin *et al.* [20]

2.2 Terrain data representations

To represent terrain data for three-dimensional scenes one can use elevation models, volumetric models, or something in between. These categories are based on how well they can represent three-dimensional terrain features.

2.2.1 Elevation models

The most common representation of terrain is with a heightmap [7]. A heightmap is a two-dimensional grid of values where each cell represents the terrain height at that point. This is very easy to create and work with as it can be represented as a grey-scale image or texture. As the terrain is defined with only one height-value for each coordinate it is not possible to represent terrain features with overhangs or caves as this requires more information. Figure 2.3 shows a heightmap on the left and a rendering of how it looks as a terrain on the right.

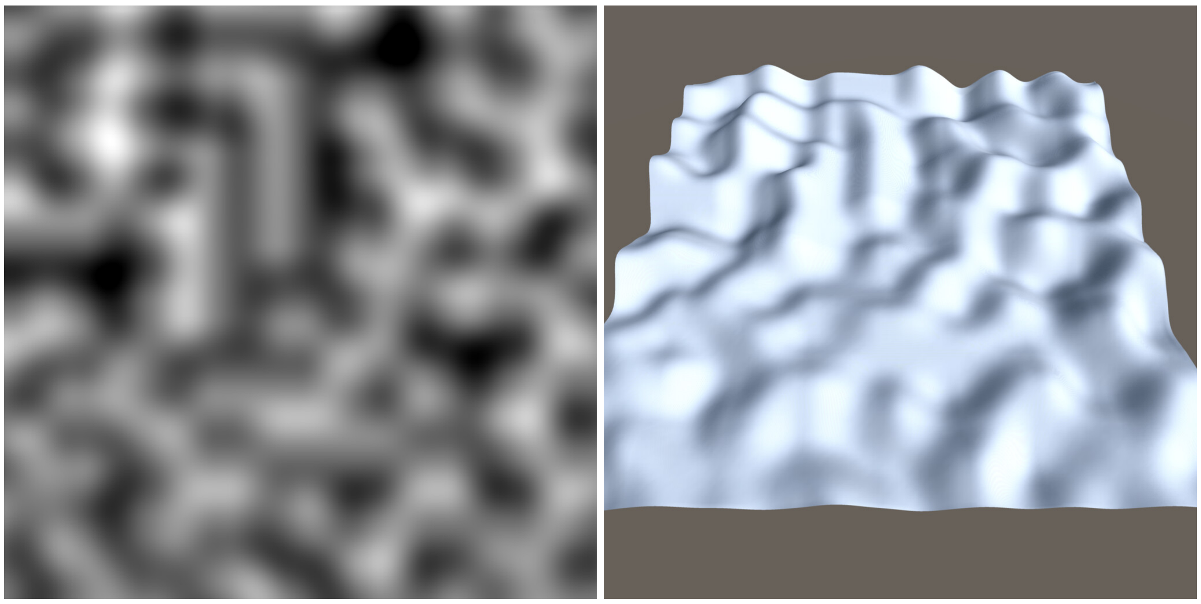


Figure 2.3: A heightmap and a rendering it as a terrain

2.2.2 Volumetric models

Volumetric models are models for representing three-dimensional terrain. Perhaps the most common representation is voxels which are evenly distributed data points in a three-dimensional grid. They take a lot more space than heightmaps as they have one more dimension. A cell in a three-dimensional grid often represents if there is terrain at that point or not, but can store any information such as material hardness and temperature. Using such models gives a lot more flexibility for the terrain as you can have overhangs and caves, but it is as of writing this paper slow and not feasible for higher resolution terrain [21].

2.2.3 Hybrid models

Hybrid models are data representations somewhere between the two previously discussed models created to mitigate one or more weaknesses while still being reasonably fast. One such model is a multilayered heightmap which is used in some simulations to represent different types of material with various properties [7]. A multilayered heightmap uses many times more memory than a heightmap and is more difficult to render.

2.3 Feature-based terrain generation using diffusion equation

In 2010 Hnaidi *et al.* [1] published a paper describing how one can generate terrain using diffusion equations at an interactive speed. Their method lets a user define splines with different properties and different heights along the curves. These splines are used in the diffusion equations which they solve to generate a smooth terrain that follows the curves of the splines. To make the terrain more natural they generate noise-based on one of the diffusion equations and add it to the smooth terrain.

Hnaidi *et al.* [1] takes the concept of using diffusion to make a smooth transition between splines from Orzan *et al.* [22] which used the concept to generate images. To solve the diffusion both papers based their implementation on the paper by McCann *et al.* [23] which uses a multigrid implementation. In this section, we will describe the method used by Hnaidi *et al.* [1], Orzan *et al.* [22] and explain diffusion and multigrid solver. Figure 2.4 shows a feature-based terrain created with diffusion.

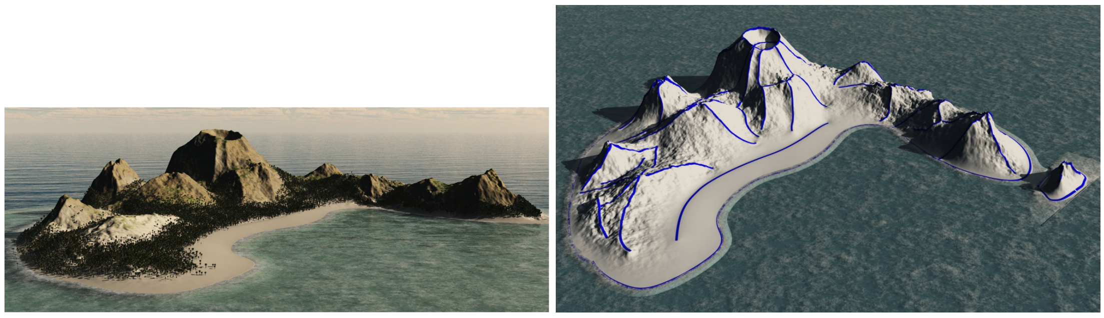


Figure 2.4: feature-based terrain from Hnaidi *et al.* [1]

2.3.1 Diffusion curves

In 2008 Orzan *et al.* [22] published a paper about using diffusion curves in image design. They described and implemented an algorithm for describing vector graphics using Bézier splines and diffusing colors from each side of the splines to color the image.

Orzan *et al.* found that using splines was quite intuitive [22] and their method was better at representing images as vector graphics than previous methods. An image was represented with 2D splines where each spline had a color on each side as well as a blur parameter. Both the color and blur could vary along the spline independently of each other. To generate the image from this data they rasterized the splines onto grids, solved two diffusion equations using the grids, and applied a blur to the image.

Hnaidi *et al.* [1] used a very similar method as Orzan *et al.* [22] for generating terrain. They introduced the concept of guided diffusion. This lets the user guide the terrain up or down away from each spline. To do this they changed color sources to represent the normal vectors of the curves. With this data, they could guide the terrain to create different terrain features. They added a gradient parameter that denoted the angle of the guided diffusion as well as two parameters for generating noise. Finally, they combined guided and unguided diffusion by introducing two weighing parameters α and β. We explain the math for this in subsection 2.3.4. Throughout chapter 3 we explain the differences between our method and the method Hnaidi *et al.* [1] used.

2.3.2 Diffusion

Diffusion is the movement of a substance from an area of high concentration to an area of low concentration. It can, for instance, be used for describing heat transfer and fluid mechanics. Diffusion can often be described with a boundary value problem as a Poisson equation. A Poisson equation has this form:

$$Au = f \tag{2.3}$$

where A is some operator acting on an unknown scalar field u with a non-homogeneous source term f. The boundary-value in this context is a definition of what value to use when at the edge of our scalar field. One can for instance use the closest value.

A popular method for solving Poisson equations is using Jacobi iteration. This method uses the residual to correct the equation in an iterative manner. Jacobi iteration works well with modern computing as a lot of the computation can happen in parallel which the GPU excels at. This method does not always converge towards a solution, if the operator A is not strictly diagonally dominant it might not converge towards a steady-state solution [24]. A commonly used operator is the five-point Laplacian discretized which is approximately:

$$Au = u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} \tag{2.4}$$

2.3.3 Multigrid solver

A multigrid solver is a method for solving Poisson equations that utilize the power of the GPU. Briggs *et al.* [25] explains the inner workings of a multigrid solver in detail as well as providing pseudo-code for an implementation. We use the same variable convention as they do. We start with the problem

$$Au = f \tag{2.5}$$

wherein the context of a multigrid solver A is a second derivative function, u is the unknown and f is our source term. We want to find the variable u or a close approximation of it. To do this we start with an initial guess v_0 which we set to 0. We calculate the residual of the equation:

$$r = f - Av \tag{2.6}$$

If the residual r is less than a threshold, for instance $r < 0.01$, we accept v as the solution. However if the residual is not less than the threshold we need to improve our guess v_0.

To improve our guess v we use Jacobi iteration.

$$v_{i+1} = Av_i + f_i \cdot h^2 \tag{2.7}$$

where $h = \frac{1}{2^g}$ and g is the grid level. Equation 2.7 is a Jacobi relaxation term from Briggs *et al.* [25]. We use this as the general Jacobi relaxation term to explain our relaxation term in section 3.4. One does not need to use this to define a Jacobi relaxation term, both Briggs *et al.* [25] and McCann *et al.* [23] has proposed other more complicated relaxation terms.

The convergence of the grid is approximately $1 - O(h^2)$ which is not practical to work with on larger grids. To improve the convergence time we can solve the problem on a smaller grid which is significantly faster and use the solution for the smaller grid as our initial guess v_0. When u and f are grids, we reduce the problem to a smaller grid by using the restriction matrix, $\mathscr{R}$, and interpolate the problem back using the interpolation, $\mathscr{P}$, matrix in Equation 2.8.

$$\mathscr{P} = \mathscr{R} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{2.8}$$

We can do this recursively and solve the equation when there is only one unknown at $h = 1$. Then we interpolate our solution to a larger grid and approximate a solution using Jacobi iteration. We repeat interpolating and solving the grid until the grid is of the correct size. Then we have an approximate solution to our initial problem.

We can visualize this method with Figure 2.5. In this example multigrid, we approximate a solution to the Poisson equation 2.5, where f is a two-dimensional grid of size 2048x2048. f is defined based on some input data. Then we restrict f with Equation 2.8 to half its size repeatedly until it is small enough. In this case, we chose 8x8, but it can be as small as 1x1. We define v_0 as a grid of size 8x8 where all the cells have value 0. Then we refine v_0 with f_0 in our Jacobi relaxation. The Jacobi relaxation term is defined with a function like Equation 2.7. We relax v_0 multiple times. When r, computed with the residual equation Equation 2.6, is lower than a set threshold, we interpolate v. We repeat refining v by relaxing it with f of a similar grid size. When our residual r is low enough we interpolate v again. This process is repeated until v is as big as the initial grid size and the residual is lower than the threshold. Then we have reached an acceptable solution.

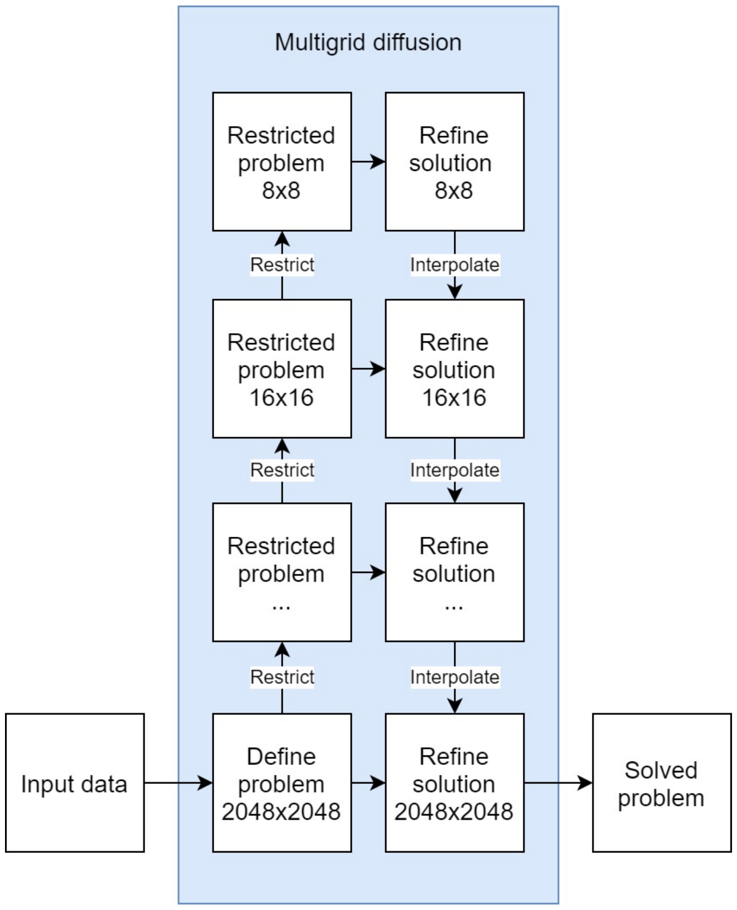


Figure 2.5: Visualization of a multigrid

Multigrid solvers are usually more complicated than what we described. One can change the method to run at interactive speeds [23] and iteratively improve the results. We do not explain how these solvers work as we do not use them for our implementation. Similarly to Hnaidi *et al.* [1] and Orzan *et al.* [22] we only use the theory above for our multigrid solver. To learn about the more complicated multigrid solver we refer the reader to Briggs *et al.* [25] and McCann *et al.* [23].

2.3.4 Generating terrain with multigrid solver

The paper "Feature based terrain generation using diffusion equation" by Hnaidi *et al.* [1] uses the multigrid solver to diffuse curves on a canvas inspired by Orzan *et al.* [22]. Orzan *et al.* introduced a local restraint to the Poisson Equation 2.5. Let C denote a sparse grid with seed values.

$$Au = f u_{x,y} = C_{x,y} \text{ if } C_{x,y} \text{ stores a value} \tag{2.9}$$

Hnaidi *et al.* [1] used this equation to smooth seed values placed by a user in the form of splines similar to Orzan *et al.* [22] to create a heightmap representing a terrain. Hnaidi *et al.* [1] used this equation with $f = 0$ to get a smooth terrain and a smoothing kernel in their Jacobi relaxation function. They made some additions to the Jacobi relaxation function we used earlier Equation 2.7 by adding a weighting with α and β and added a gradient.

$$v_{i+1} = \alpha A v_i + \beta(F_N + G_{x,y}), \text{ where } A = \frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.10}$$

where F_N is the weighted average based in the direction of the normal vector of the curve, $G_{x,y}$ is the change in height and the two components are weighted by α and β where $0 \leq \alpha, \beta \leq 1$ and $\alpha + \beta \leq 1$.

These changes are made to have more control over the diffusion process so the diffusion looks more like the terrain features they try to generate. They similarly to Orzan *et al.* [22] only run one cycle with the number of Jacobi iterations based on the grid level h where the number of Jacobi iterations are $5(l - h)$ where l is the number of grid levels and h is the current grid level.

2.4 Erosion simulation

In the context of terrain generation, we define erosion simulation as a process to simulate the effects of some natural phenomena on the terrain where mass is transported from one location to another. This can be simplified into three steps, detach mass from one location, transport it to another location and deposit the mass at the new location [7]. Here we will discuss the two most common phenomena that are used in erosion simulations: hydraulic and thermal erosion.

2.4.1 Hydraulic erosion

Hydraulic erosion simulates the terrain deformation from sediment being picked up by water and moved to another location at a lower altitude. There are two methods for simulating such erosion: Eulerian and Lagrangian [7]. The Eulerian approach is discretizing the terrain

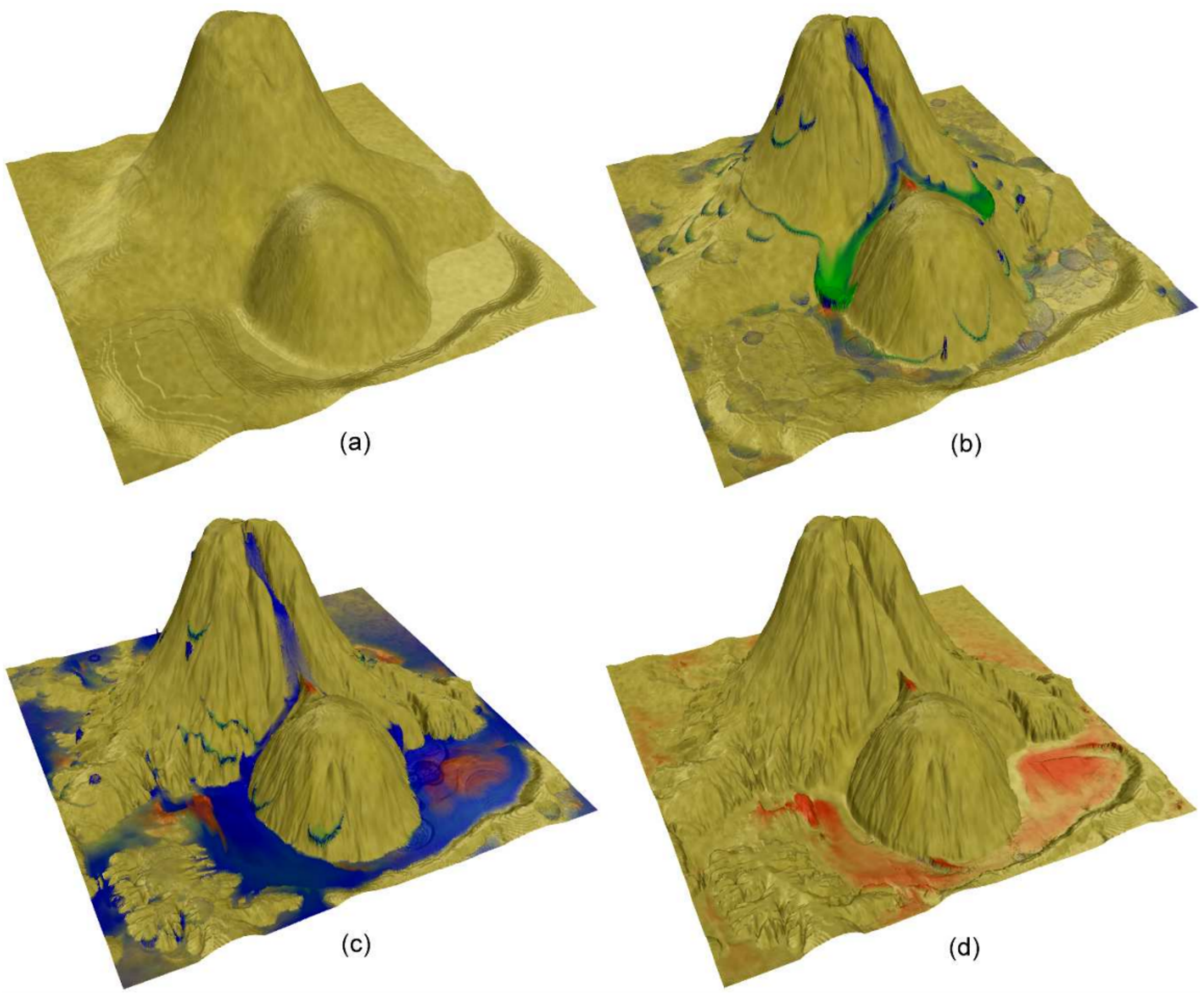


Figure 2.6: Terrain eroded with hydraulic erosion from Mei *et al.* [26]

Copyright © 2007, IEEE

onto a grid and remember how much water and sediment is in each cell. This is the method used in this thesis and by Mei *et al.* [26]. The Lagrangian approach is simulating particles of water and remember where each particle is located and how much water and sediment each particle has.

Hydraulic erosion creates quite distinct features in a terrain which is shown in Figure 2.6. This figure shows four renderings of the erosion process. Subfigure a) shows the initial terrain, b) is early in the erosion process, c) is later in the erosion process, and d) is the terrain after the erosion. The colors represent different aspects of the erosion process. Blue represents water, green represents suspended sediment, and red represents deposited sediment.

2.4.2 Thermal erosion

Thermal erosion is simulating rock slides caused by weathering of the terrain making it unstable. The weathering is mainly caused by water sliding in between rocks freezing and expanding when the temperature changes causing the rocks to break. This erosion mainly affects cliffs and steep slopes. Weathering of the rocks was one of many erosion processes that Musgrave *et al.* [27] described as a part of thermal erosion of heightfields.

2.5 Programming the GPU

The GPU has a different architecture than the CPU which can be utilized to speed up calculations on computationally heavy algorithms that can be run in parallel. Goodnight *et al.* [28] compared their multigrid implementation on the CPU and the GPU and found the GPU implementation between 13 and 15 times faster. One of the key differences is that the GPU can load multiple mathematical calculations using the same operator and calculate them all simultaneously. This can be utilized to run the same operation for every pixel of an image or every element of a list. When it comes to terrain generation this is used to speed up noise functions, simulations, diffusion, and rendering, however, rendering is out of scope for this thesis.

When programming on the GPU, one has to use a language the GPU can understand. Such languages are called shader languages. There are different types of shaders one can use. In DirectX 9 only fragment and vertex shaders were supported [29]. These are the two shaders needed to render triangle-based objects. This is exploited in multigrid-solvers [28] and simulations [26] as there were no alternatives. With DirectX 11 the compute shader was introduced [30]–[32]. The compute shader does not have the same restrictions as the fragment and vertex shader and can be significantly faster for some calculations [33]. In this paper, we will only use the compute shader when running code on the GPU.

Chapter 3

Methods

In this chapter, we give an overview of our method for generating terrain before defining new concepts and explaining how we use diffusion, multigrid solver, noise, and erosion.

3.1 Overview

First, we introduce our method for generating terrain from our features with diffusion, noise, and simulating erosion to get the final terrain. Then we introduce our multigrid solver and the differences from the previous works of Hnaidi *et al.* [1] and Orzan *et al.* [22]. Finally, we introduce our four novelties which we discuss in chapter 6.

Figure 3.1 shows how we divide our method into four parts. This figure uses actual data from generating a terrain with our method using one spline. Each box with an image is a texture used in our method except "3D feature splines" which is a representation of our input spline. "Final terrain" is a high-resolution rendering of the generated terrain. Our input data are curves with attached parameters which we call feature splines. We define these in section 3.2. This data is modeled by a user in a 3D environment. Each feature spline contains information about its position, the height of the terrain, gradients as well as noise, warp, and erosion parameters. We rasterize the feature splines to six grids as shown in Figure 3.1. Then we diffuse the parameters and normals and use guided diffusion on the heightmap. In the third step, we generate and add noise the heightmap which we use to simulate erosion. After adding the erosion to the heightmap we get our final terrain.

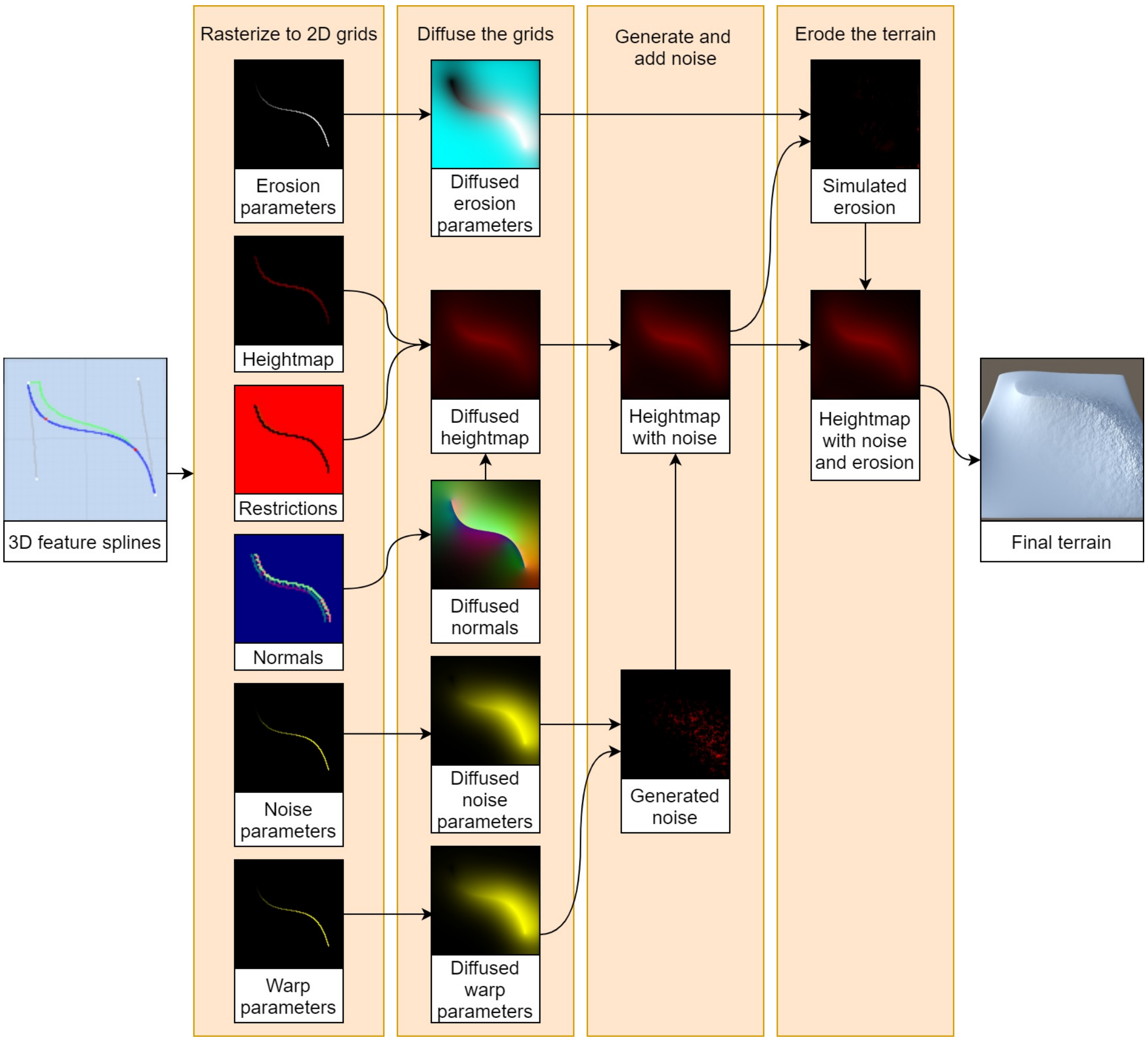


Figure 3.1: Illustration of our method

This is a short overview of our method which is similar to the method of Hnaidi *et al.* [1] with added parameters and erosion. They used a multigrid solver to generate their terrain. Our multigrid solver differs from theirs by doing all the four steps above and not just the diffusion. A multigrid solver solves a diffusion equation by reducing the problem to a smaller grid, solving the easier problem, interpolating the easier problem to a larger grid, and refining the solution. Instead of reducing the problem, we define the problem for each level in the multigrid as shown in Figure 3.2. We define the problem by rasterizing the feature splines and we refine the solution with our Jacobi relaxation term described in section 3.4 which results in a smooth terrain.

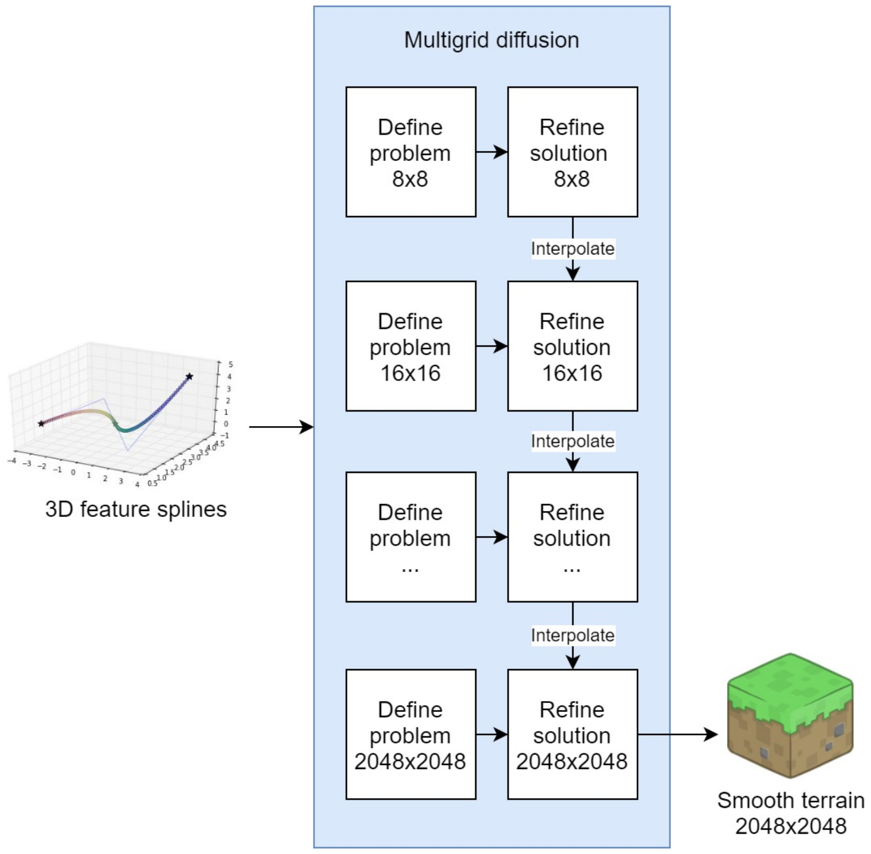


Figure 3.2: Simplified multigrid solver

We extend our multigrid solver to simulate erosion in addition to solving the diffusion equation. Similarly to refining and interpolating the result to the diffusion equation we erode and interpolate the result of the erosion. As shown in Figure 3.3 we take the smooth intermediate terrain, add noise, and erode the terrain on each level of the multigrid. This is what we call multigrid erosion - the process of eroding the terrain at smaller grids and interpolating the erosion results. We describe this in more detail in section 3.7 and subsection 4.5.1.

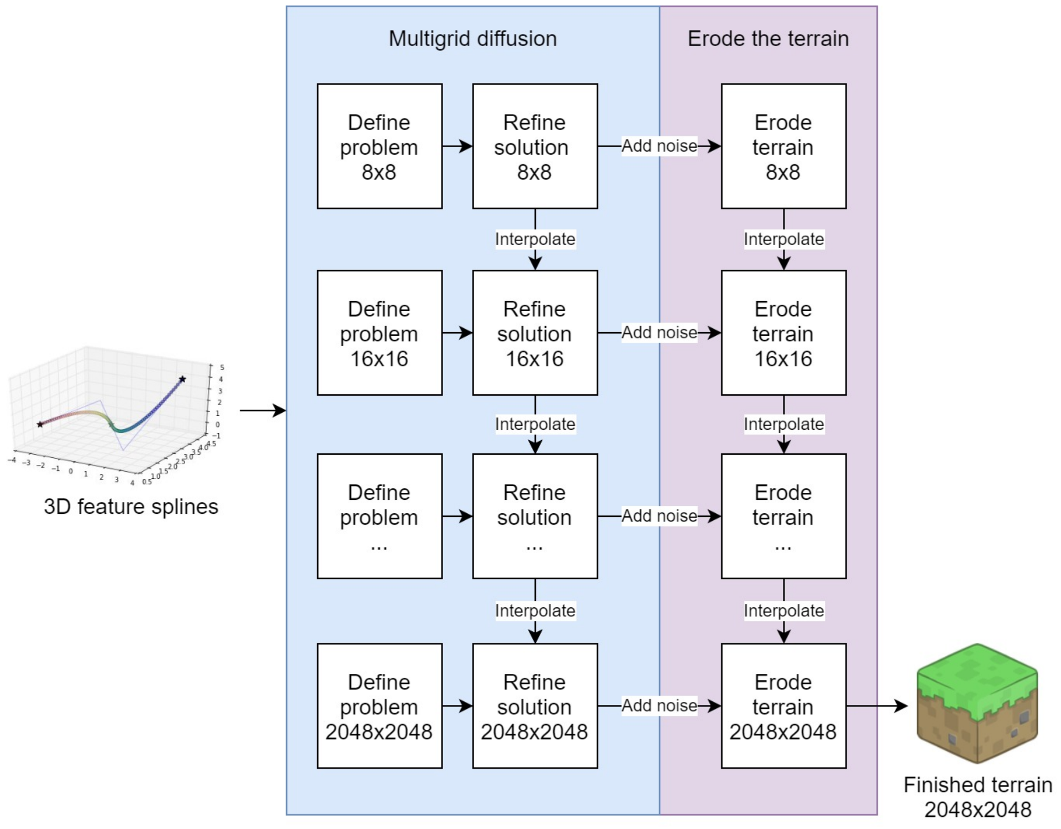


Figure 3.3: Illustration of our multigrid solver where each row represents a grid level

Our method has four novelties which we discuss individually in chapter 6. The first novelty is our multigrid diffusion which we illustrated with Figure 3.2 where we define the problem on each grid level. section 3.4 describes our diffusion and multigrid diffusion in detail as well as shows how this novelty gives more control over the diffusion process. Our second novelty is the multigrid erosion illustrated with Figure 3.3. The third novelty is our constrained erosion which gives the user more control over the erosion process. We explain both multigrid and constrained erosion with the erosion process in section 3.7. Our last novelty is extending the noise equation from Hnaidi *et al.* [1] with the warped noise equation from Carpentier *et al.* [14]. We refer to this novelty as diffused warped noise and explain it in section 3.6.

3.2 Concepts

We introduce two new concepts which we will use throughout the thesis: feature spline and meta point. Both of these are based on concepts introduced by Hnaidi *et al.* in [1] and extended and adjusted by us.

3.2.1 Feature spline

Feature splines are three-dimensional vector-based piecewise Bézier cubic splines with a set of attached meta points. The feature splines work like feature curves in Hnaidi *et al.* [1] but are in three dimensions instead of two and have slightly different data attached to them which we call meta points. Each feature spline contains one or more Bézier curves, the number of curves are denoted as C, and zero or more meta points are denoted as M_i.

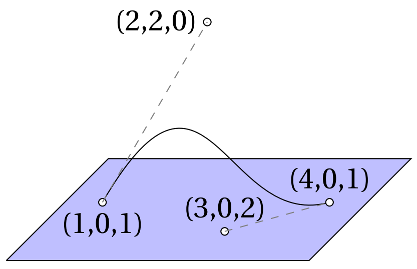


Figure 3.4: Three-dimensional cubic Bézier curve

3.2.2 Meta point

Meta points are data points placed along a feature curve describing the properties of the spline. A meta point is defined by the following attributes where m denotes the number of meta points:

$$M_i = (u_i, r_i, (a_i, b_i, h_i), (A_i, R_i), (D_i, F_i), (w_i, s_i)) \in [0, m]$$

Meta parameter	Type	Description
u_i	float	the position parameter
r_i	float	half the width of the feature spline
a_i and b_i	2D vectors	they control the gradient on both sides of the spline
h_i	float	hardness of the gradient
A_i and R_i	floats	noise parameters
D_i and F_i	floats	noise warping parameters
w_i and s_i	floats	parameters used to control erosion

Table 3.1: Meta point parameters

A meta point is attached to a feature spline and positioned on the spline based on the parameter u_i. With $u_i = 0$ the meta point is positioned on the start of the spline and $u_i = 1$ is positioned at the end of the first curve. The value is restricted to be between $[0, C]$ where C

denotes the number of curves of the feature spline. All parameters except for u_i are interpolated linearly between the closest meta points along the spline. Figure 3.5 shows four meta points positioned on a feature spline with one curve.

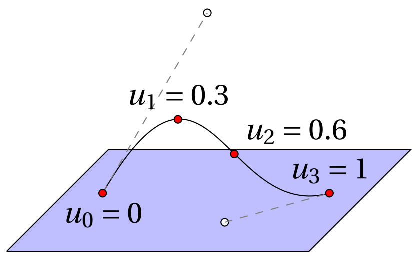


Figure 3.5: Positions of four meta points on a feature spline

A feature spline has a width, r_i, a gradient on both sides, a_i and b_i, and hardness, h_i. The wider the spline the more cells it covers in the grid during our diffusion process. The gradients have a width, height, and hardness used for calculating β and $G(x, y)$ in the diffusion. From the direction of the curve, we visualize this in Figure 3.6.

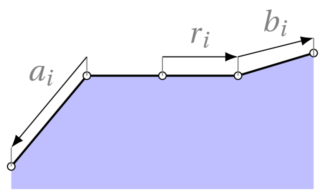


Figure 3.6: Visual representation of a meta point r_i, a_i and b_i from the direction of the curve

The rest of the parameters in Table 3.1 are used for generating noise and simulating erosion. We explain how we prepare the parameters in section 3.5 with parameter diffusion and then use them in section 3.6 and section 3.7.

3.3 Terrain modeling

A user models a terrain by placing multiple feature curves with attached meta points in a 3D space. The terrain follows the feature splines and goes in the directions of the gradients defined by the meta points. With meta points, a lot of terrain features can be created. There are many examples in Hnaidi *et al.* [1].

3.4 Diffusing the terrain

Our diffusion is expressed as a Poisson equation with a restriction on the solution and a boundary value problem. Let $S(x, y)$ denote the height value of our curves at (x, y). Now we can describe our diffusion as:

$$Au = 0 \tag{3.1}$$

$$u(x, y) = S(x, y) \text{ if } S(x, y) \text{ is set} \tag{3.2}$$

We simplify our Poisson equation to a Laplace equation with the local restriction in Equation 3.2. We define our second derivative function as a discretized matrix similar to the one defined by Hnaidi *et al.* [1]. Let $0 \leq \alpha$, $0 \leq \beta$ and $0 \leq \alpha + \beta \leq 1$. $G(x, y)$ is a grid with guiding gradients and $N(x, y)$ is normalized normal vectors. The Jacobi relaxation iterates over i.

We define the Jacobi relaxation term as:

$$v_{i+1} = \alpha A v_i + \beta(B v_i + G) \tag{3.3}$$

This equation is a weighted version of the general Jacobi relaxation Equation 2.7 where $h = 1$. We set $h = 1$ because we define our Poisson equation at each grid level. This leads to each grid level being the first. The first part of the equation weighted with α is the smoothing part. We define A from the nine-point approximation to the Laplace function, similarly to how Hnaidi *et al.* [1] used the five-point approximation, shown in Equation 3.4. This averages the current cell to the surrounding cells.

$$A = \frac{1}{8}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.4}$$

The second part, weighted with β, is a guiding term that guides the diffusion in the direction of the normalized normal vector scalar field N. B calculates the value in the opposite direction of the normal vector in a bilinear manner. This part takes the pixel in the direction of N

and adds the gradient G.

$$B = \begin{bmatrix} 0 & max(0, N_y)^2 & 0 \\ min(0, N_x)^2 & 0 & max(0, N_x)^2 \\ 0 & min(0, N_y)^2 & 0 \end{bmatrix} \tag{3.5}$$

By rearranging the previous equations we get the following Equation 3.6 which is equal to the general Jacobi relaxation term from Equation 2.7 when $h = 1$. We use the operator K for our more complex operator rather than A to avoid confusion.

$$v_{i+1} = Kv_i + f, \text{ where } f = \beta G \tag{3.6}$$

$$K = \begin{bmatrix} \frac{1}{8}\alpha & \frac{1}{8}\alpha + \beta \cdot max(0, N_y)^2 & \frac{1}{8}\alpha \\ \frac{1}{8}\alpha + \beta \cdot min(0, N_x)^2 & 0 & \frac{1}{8}\alpha + \beta \cdot max(0, N_x)^2 \\ \frac{1}{8}\alpha & \frac{1}{8}\alpha + \beta \cdot min(0, N_y)^2 & \frac{1}{8}\alpha \end{bmatrix} \tag{3.7}$$

Our relaxation is slightly more complex than the one used by Hnaidi *et al.* [1] and uses a nine-point approximation of the Laplace function instead of an averaging function. We use the nine-point approximation and not the five-point approximation as this seems to converge with marginally fewer iterations.

We will use the same terminology with this diffusion as Hnaidi *et al.* [1] where this is referred to as guided diffusion. If $\alpha = 1$ and $\beta = 0$ for all cells in the grid, we call it diffusion.

For our boundary value problem, we chose the closest value. This means when a point in any of our calculations is outside of our grid we sample the closest point which is inside of the grid instead.

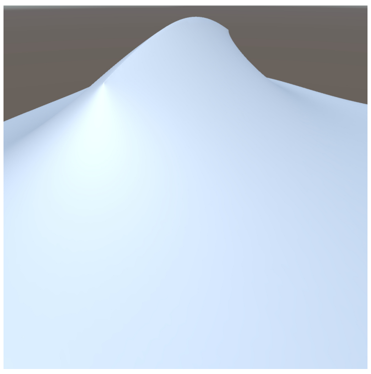


Figure 3.7: A spline diffused on all grid levels, 2-12

3.4.1 Multigrid solver

The change we have made to our multigrid solver compared to Hnaidi *et al.* [1] is that we define our diffusion equation on each grid level, not just the largest grid. We start at a smaller grid. We start our diffusion at a grid with a size of 4x4. According to our estimates, they started with a grid of size 32x32 (or 64x64). We would argue they chose this size because it limits the area a spline affect which makes it easier to work with. This is also faster as you do not need to calculate the smallest grids. The ability to limit the area a spline affect is very useful when using the workflow they described: incrementally adding more details.

We made the process of incrementally adding details easier by letting the user decide how much area each spline should affect. Our multigrid solver defines the diffusion equation on each grid level. This lets the user decide if they want to include each feature spline on which grid levels. We diffused the spline shown in Figure 3.7 where we only defined it on some grid levels to show how the spline affects the grid on the different grid levels. The spline used in Figure 3.7 is defined on all grid levels 2-12, that is from grid size 4x4 to 2048x2048. We rendered this spline three times in Figure 3.8 showing the diffusion on the smallest grids, grid levels 2-6 (grid sizes 4x4 to 64x64) in (a), grid levels 6-8 (grid sizes 64x64 to 256x256) in (b), and grid levels 8-12 (grid sizes 256x256 to 2048x2048) in (c). Together these three renderings splines are equal to the spline rendered in Figure 3.7, thus we call it a deconstruction of the spline. Splines defined on the smallest grid levels as in Figure 3.8a are very smooth and great for defining the general shape of the terrain. On the general shape, we can add smooth curvature, ridges, and rivers with splines defined on the middle grid levels as done in Figure 3.8b. We can add sharp details like ridgelines, roads, erosive features, or cliffs with a spline defined on the largest grid levels as in Figure 3.8c. Grid levels let the user decide how much of the terrain changes when adding a new spline. It can be compared to different brush sizes when painting. One can also create different features by excluding the spline from some grid levels. Figure 3.9 shows the spline where it is excluded from the grid levels 5-8.

Figure 3.7, Figure 3.8 and Figure 3.9 are all renderings from the same angle of the same spline. The only difference in the input data between Figure 3.7 and the others are which grid levels the spline is defined on.

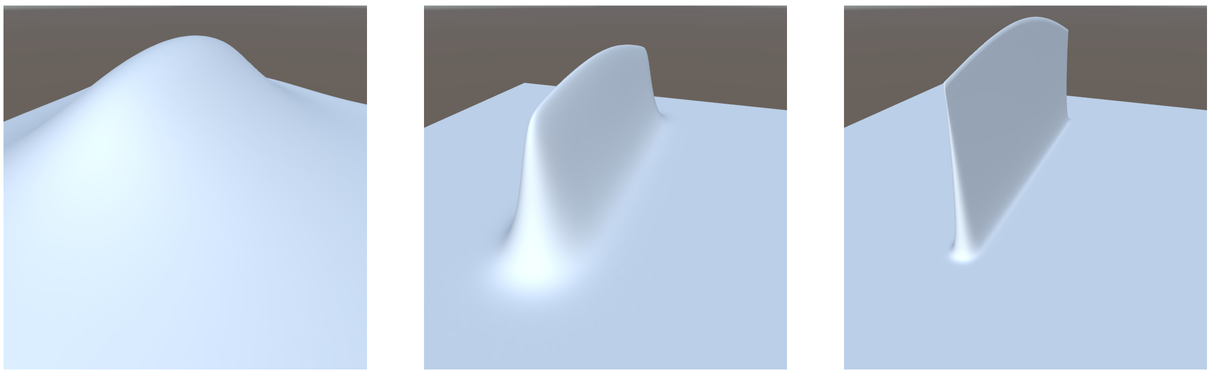

(a) Defined on grid levels 2-6 (b) Defined on grid levels 6-8 (c) Defined on grid levels 8-12

Figure 3.8: Deconstructed diffusion of a spline

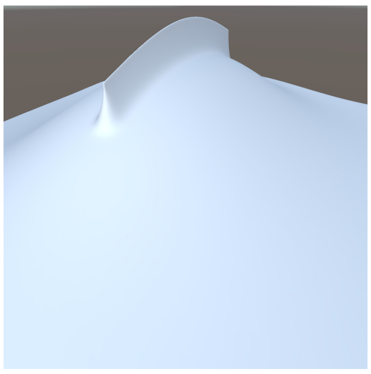


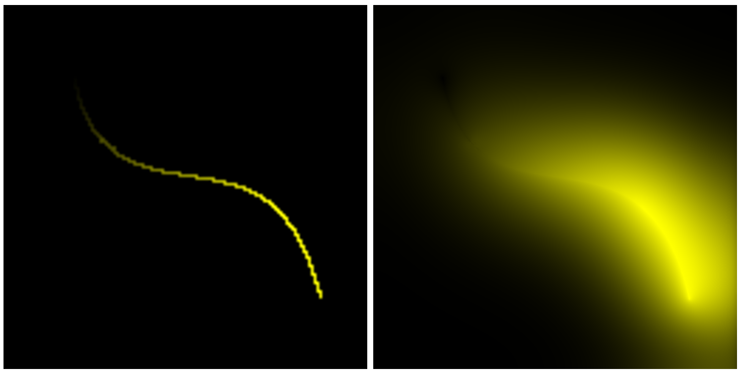
Figure 3.9: A spline diffused on grid levels 2-5 and 8-12

3.5 Parameter diffusion

We introduce the concept of parameter diffusion which we use when generating our diffused warped noise as well as when simulating erosion. This was used by Hnaidi *et al.* [1] for generating noise that was coherent with the terrain features and we use this method to give more control over our erosion as well as generate our noise. Parameter diffusion is the process of mapping the parameters from our feature splines and meta points to a grid. When diffusing a parameter we rasterize the data from the feature splines to a grid and interpolate the values linearly between the meta points along each spline. The resulting grid contains our diffusion seed values which we use in our local restriction in Equation 3.2. We do not use gradients, thus we can introduce a new local restriction: $\beta = 0$ and $\alpha = 1$ for all cells. Our Jacobi relaxation term simplifies to:

$$v_{i+1} = Av_i \tag{3.8}$$

After solving the diffusion equation we get a smooth grid which we can sample to get the parameter value on any cell in the grid. Figure 3.10 shows the rasterized parameters and the solved diffusion grid of two parameters stored in the red and green components of an image. The spline in the figure has a meta point at the beginning with value 0 for both parameters and a meta point at the end with value 1 for both parameters represented as red and green. The parameter diffusion is affected by what grid levels the splines are defined on.



(a) A rasterized parameter — Rendered at a 128x128 grid to clearly show the line and color change

(b) The result of diffusion — Rendered at a 1024x1024 grid to show the result

Figure 3.10: Example of parameter diffusion of two values

3.6 Noise generation

Our noise generation is based on multiple layers of smooth noise. The behaviour of the noise changes based on our noise and warp parameters which we have diffused. We diffuse the parameters (A_i, R_i) and (D_i, F_i) from our meta point definition in subsection 3.2.2. To

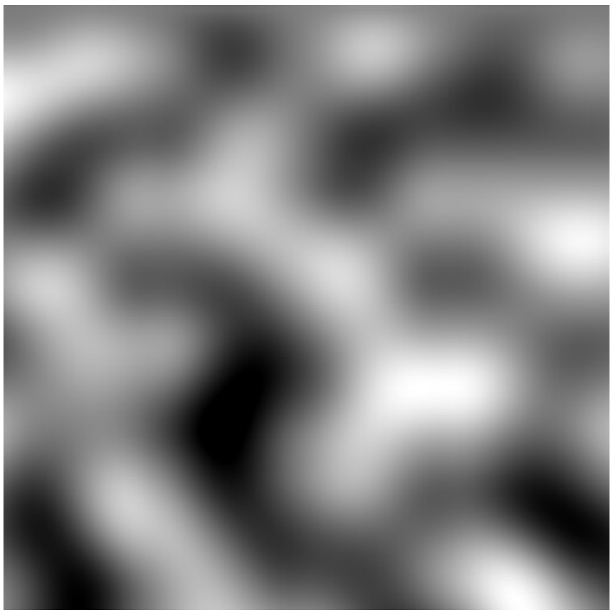


Figure 3.11: Improved Perlin noise [9]

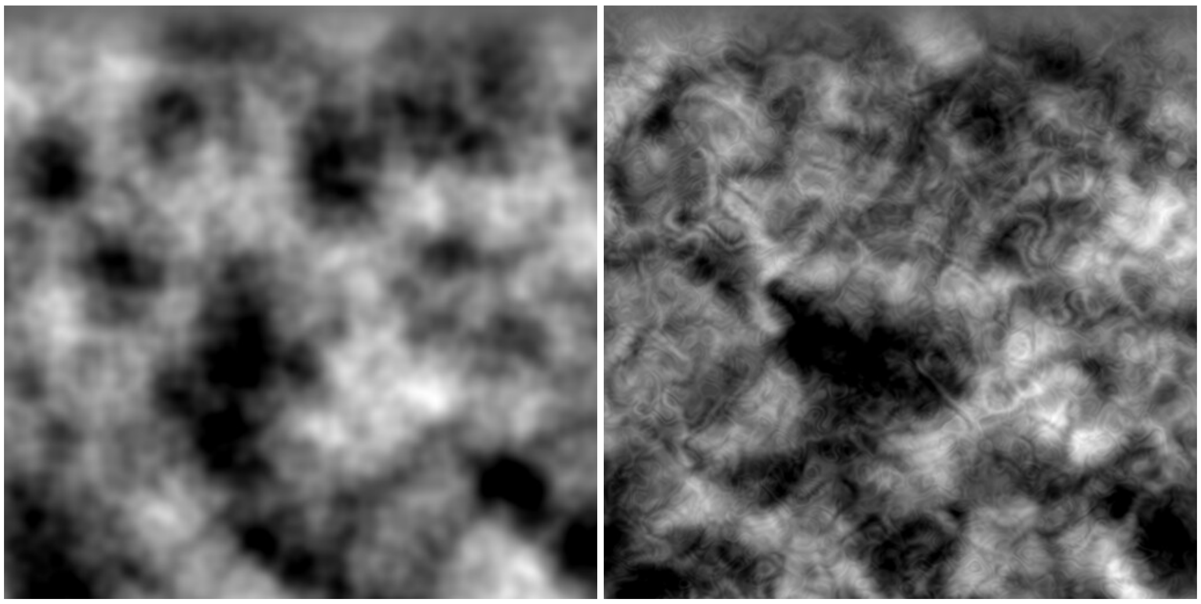
refer to our diffused parameters we use $A(x, y)$, $R(x, y)$, $D(x, y)$ and $F(x, y)$ respectively. Let $S(x, y)$ be a smooth noise function. We use improved Perlin noise by Perlin [9] visualized in Figure 3.11. Hnaidi *et al.* [1] used $A(x, y)$ and $R(x, y)$ similarly to us. They defined their noise $N(x, y)$ as the following where r and n are constant integer values.

$$N(x,y) = A(x,y)\sum_{k=0}^{n}\frac{1}{r^{k(1-R(x,y))}}S(r^k x, r^k y) \tag{3.9}$$

This noise is rendered in Figure 3.12a with $n = 4$ and $r = 2$. We introduce a transformation of the points before sampling the smooth noise function S. This is a part of the noise generation function which Carpentier *et al.* [14] used. They defined three transformation functions used in different parts of their noise function. Carpentier *et al.* defined their general noise function $N(p)$ as this:

$$N(p) = T_{post}(\sum_{k=0}^{n} A_k T_{in}(S(T_{pre}(\phi_k p)))) \tag{3.10}$$

To create a warp effect with this equation we need to define T_{pre} as a non-linear mapping function. We combine these noise functions by including T_{pre} from Carpentier *et al.* [14] in the noise equation from Hnaidi *et al.* [1] to get our noise function Equation 3.11.



(a) Without warping (b) With warping

Figure 3.12: The effect of noise warping

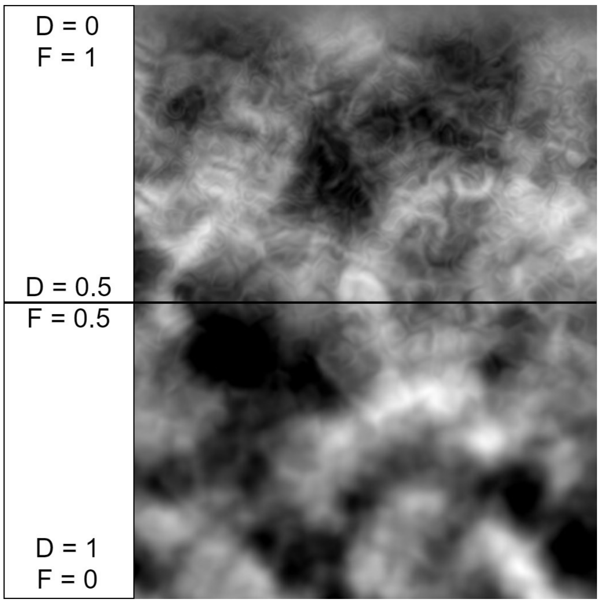
$$N(x,y) = A(x,y)\sum_{k=0}^{n}\frac{1}{r^{k(1-R(x,y))}}S(T_{pre}(r^k x, r^k y, k)) \tag{3.11}$$

We use $n = 4$ and $r = 2$ to generate our noise. These values control the number of layers of noise and how the layers differ in frequency and amplitude. We create a warp effect with our definition of T_{pre}. To make the function non-linear we sample the smooth noise function at different locations to get an offset vector and add this to the point p. With the following equation, D scales the lower frequencies and F scales the higher frequencies.

$$T_{pre}(p,k) = p + (k\cdot F(p) + \frac{D(p)}{1+k})[S(s_1+p), S(s_2+p)] \tag{3.12}$$

where s_1 and s_2 are seed values which should be different and p is a point or two float values. In our implementation we used $s_1 = 50$ and $s_2 = 150$. These equations generate smooth noise with no visible artifacts. For different noise, one can experiment with different values of n and r as well as the smooth noise function S. Figure 3.12 shows the same noise with and without warping.

We can see the effect of D and F separately in Figure 3.13. In Figure 3.13a we vary the parameters D and F to show how these parameters affect the terrain. We used $D = 1$ and $F = 0$ at the bottom of the image, $D = F = 0.5$ in the middle and $D = 0$ and $F = 1$ at the top of the image. One can see that the bottom half has the large features warped and the top half has warping of a smaller scale as well. Figure 3.13b was generated with no warping in the middle and varying warping and roughness above and below. This image shows that our noise function does not generate any artifacts.



(a) The effect of D and F

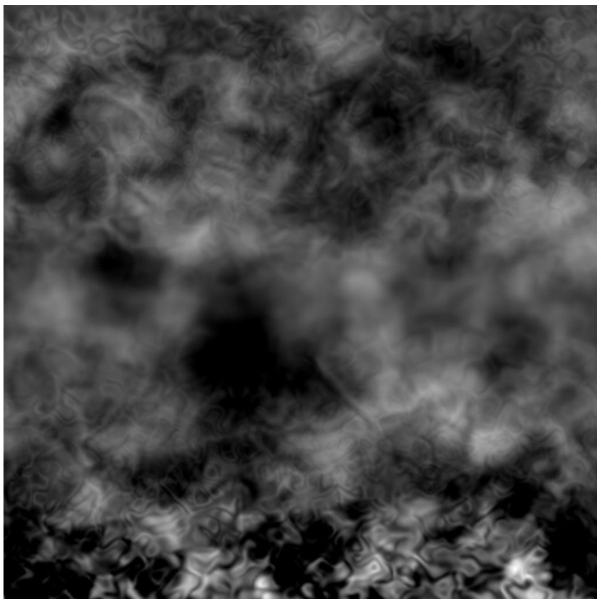

(b) Varying values of R, D and F

Figure 3.13: Blending of noise

3.7 Erosion simulation

After generating a smooth terrain and adding warped noise we use this to simulate erosion. We chose to use an Eulerian approach to erosion simulation meaning simulating using a grid. This was a natural choice as we use grids in our multigrid solver. The erosion can make the terrain look more natural as it simulates a natural process. Noise is by nature repetitive at a certain scale. This is slightly mitigated by parameter diffusion, though with a large amount of noise it is quite noticeable. This is why we added erosion as this is not repetitive and changes the terrain in a natural way.

3.7.1 The erosion process

Our method for hydraulic erosion is very similar to the method proposed by Mei *et al.* [26]. Our method is a seven step model designed to run on the GPU using shallow water differential equations to simulate water flow. The faster the water moves and the steeper the terrain is the more erosion occurs. In this section, we describe each of the seven steps in our method.

This simulation has a lot of variables that the user can set to adjust the water and erosive behavior. For a better reading experience, we introduce these variables as a table with a short description in Table 3.2. We simplified all equations which use Δt. Whenever Δt is used to calculate a value, the new value represents the value after Δt time has passed and should be written with $X_{t+\Delta t}$, assuming X is the updated value. When X is used in the equation it represents the value at the current time X_t.

Erosion parameters	Short name	Description
Pipe Area	A	The diameter of the virtual pipe
Gravity	g	The downwards acceleration affecting the water
Pipe Length	l	The length of the virtual pipe
Sediment Capacity	K_c	How much sediment can the water hold
Suspension Rate	K_s	How fast sediment is picked up into the water
Deposition Rate	K_d	How fast sediment is deposited from the water
Evaporation	K_e	Percentage of water evaporated
Max Rain Intensity	I_{max}	Maximum water in a raindrop
Min Rain Intensity	I_{min}	Minimum water in a raindrop
Max Rain Size	R_{max}	Maximum radius of a raindrop
Min Rain Size	R_{min}	Minimum radius of a raindrop
Time Delta	Δt	Step size of the simulation

Table 3.2: Erosion Parameters

Water increment

First, we add water to the simulation either as a water source or as randomized rain. A water source has a position, radius, and intensity describing how much water is added. Rain has similar properties as a water source, but they are randomized each time this step is run. The position parameter is a randomly chosen continuous point inside of the grid. Both the intensity and radius are randomly chosen continuous values between minimum and maximum values set by the user. We denote the minimum and maximum intensity as I_{min} and I_{max} and similarly R_{min} and R_{max} for the radius.

Algorithm 1 is pseudo-code to explain how we randomly choose our properties and then update the water height. Let *width* and *height* be the width and height of our grid and let *random* be a function returning a random number between 0 and 1. First, we calculate our three properties before checking each cell in our grid. If the position of a cell is inside of the radius of the raindrop we add water to this cell. The amount of water we add is the intensity property multiplied with w_i which is our diffused parameter defined in subsection 3.2.2. We store the water height in the grid S_w.

Algorithm 1 Water increment

```
$intensity \leftarrow random() \cdot (I_{max} - I_{min}) + I_{min}$
$radius \leftarrow random() \cdot (R_{max} - R_{min}) + R_{min}$
$position \leftarrow float2(random() \cdot width, random() \cdot height)$
for all $(x, y) \leftarrow grid$ do
    if $(x - position.x)^2 + (y - position.y)^2 \leq radius^2$ then
        $S_w[(x, y)] \leftarrow S_w[(x, y)] + intensity \cdot w_i[(x, y)]$
    end if
end for
```

Water flux

Then we calculate the flow of water called water flux, based on the height difference between neighboring cells. For each cell, we compare the height to the four closest neighboring cells and calculate how much water should flow out to each of the cells. We denote d as the four closest neighbors $d = l, r, t, b$ which represents the neighboring cell to the left, right, top and bottom respectively.

$$f_d = max(0, f_d + \Delta t \cdot A \cdot \frac{g \cdot \Delta H_d}{l}) \tag{3.13}$$

where d denotes the direction and is ran for up, down, left, and right, f is a grid where our flux is stored and ΔH_d is defined in the following equations.

$$\Delta H_l(x, y) = (S_w(x, y) + S_t(x, y) + T(x, y)) - (S_w(x-1, y) + S_t(x-1, y) + T(x-1, y)) \tag{3.14}$$

$$\Delta H_r(x, y) = (S_w(x, y) + S_t(x, y) + T(x, y)) - (S_w(x+1, y) + S_t(x+1, y) + T(x+1, y)) \tag{3.15}$$

$$\Delta H_t(x, y) = (S_w(x, y) + S_t(x, y) + T(x, y)) - (S_w(x, y+1) + S_t(x, y+1) + T(x, y+1)) \tag{3.16}$$

$$\Delta H_b(x, y) = (S_w(x, y) + S_t(x, y) + T(x, y)) - (S_w(x, y-1) + S_t(x, y-1) + T(x, y-1)) \tag{3.17}$$

where S_t is a grid representing the removed and added sediment, S_w is the water height and T is the terrain height. To ensure we do not move more water out of a cell than there is in the cell, we use a scaling factor K. We update the value of f_d by multiplying it with K in Equation 3.19. Though Equation 3.19 is not a valid mathematical representation of updating f_d, we use it as this is how the process is described by Mei *et al.* [26].

$$K = min(1, \frac{S_w \cdot l_x \cdot l_y}{(f_l + f_r + f_t + f_b)\Delta t}) \tag{3.18}$$

$$f_d = K \cdot f_d \tag{3.19}$$

Water update

We use the water flux grid to update the water height. The new water height for each cell is the current water height plus the sum of incoming flux subtracted by the sum of outgoing flux. We calculate the outgoing flux by summing the the four flux values in the current cell. To calculate the incoming flux we calculate the outgoing flux in the direction of the current cell from the four closest neighbours. For example we calculate the incoming flux from the left by taking the right outflow of the cell to the left: $f_r(x-1, y)$.

$$\begin{aligned} S_w = &S_w + \Delta t(\\ &(f_r(x,y) + f_l(x,y) + f_t(x,y) + f_b(x,y)) \\ &-(f_r(x-1,y) + f_l(x+1,y) + f_t(x,y-1) + f_b(x,y+1))) \end{aligned} \tag{3.20}$$

Water velocity

The velocity is computed based on the flux map. We calculated the velocity as the flux in x and y direction divided by the size of the cell. To make the simulation more stable we restrict the velocity v to $-0.5 < v \cdot \Delta t < 0.5$.

$$V_x = \frac{f_r(x-1,y) + f_l(x+1,y)}{l} \tag{3.21}$$

$$V_y = \frac{f_t(x,y-1) + f_b(x,y+1)}{l} \tag{3.22}$$

Local tilt angle

The local tilt angle, ϕ, is the angle between the direction of the terrain at a point and a flat plane aligned with the horizon. Thus we have to find the direction of the terrain for each point. We take the lowest neighboring point in the x- and y-direction and subtract it from the height of the cell in focus, $H(x,y)$. Let p be a point representing the height difference in the x- and y-direction.

$$\begin{aligned} p_x(x,y) &= H(x,y) - min(H(x-1,y), H(x+1,y)) \\ p_y(x,y) &= H(x,y) - min(H(x,y-1), H(x,y+1)) \end{aligned} \tag{3.23}$$

We restrict p_x and p_y to $0 <= p_x <= 1$ and $0 <= p_y <= 1$ to only suspend sediment when there is a downward slope. By restricting p like this we ignore upwards slopes. This can be formulated as a vector $v = (p_x, p_y, 1)$ and compared to the vector to a vertical vector $u = (0,0,1)$.

$$cos\phi = \frac{v \cdot u}{|v| \cdot |u|} \tag{3.24}$$

As we are using $sin\phi$ in our calculations we can calculate that directly.

$$sin\phi = \sqrt{1 - cos^2\phi} \tag{3.25}$$

The final Equation 3.25 uses the square root which is a quite slow calculation. An efficient approach is to check if either the sediment capacity modifier, s_i, or the velocity is zero to cut off the calculation early.

Suspension and deposition

In this step, we calculate the sediment capacity, C, for each cell and update the terrain height accordingly. We use the velocity and local tilt angle to calculate our sediment capacity as well as our diffused parameter s_i from the meta points. Let E denote the maximum erosion change which limits both sediment suspension and deposition to exceed E. E can have any value between 0 and 1, we found that $E = 0.15$ created erosive features of a desirable maximum depth. We calculate the sediment capacity of a cell with the following equation.

$$C(x, y) = K_c \cdot sin\phi(x, y) \cdot |V(x, y)| \cdot s_i(x, y) \cdot (\frac{1}{E} \cdot (E - |S_h|)) \tag{3.26}$$

Let S_s denote a grid representing the suspended sediment, S_{s1} another grid representing the updated suspended sediment, and S_h represent the change in terrain height. If $C(x, y)$ is greater than $S_s(x, y)$ we suspend some sediment from the terrain, else we deposit some. We decide if and how much sediment to suspend or deposit based on the sediment capacity calculated with Equation 3.26. This algorithm first calculates the sediment capacity and compares it to the amount of suspended sediment. If the capacity is more than or equal to the suspended sediment we move some sediment from the terrain to the suspended sediment S_s, if not we move some sediment the other way. We calculate the amount of sediment to be suspended or deposited based on the suspension and deposition rates K_s and K_d and the difference between the suspended sediment amount and the sediment capacity. We store the suspended sediment temporarily in S_{s1} and write the suspended sediment back to S_s in the next step. Algorithm 2 explains this process as pseudo-code.

Algorithm 2 Suspension and deposition

```
$C(x, y) \leftarrow K_c \cdot sin(\phi(x, y)) \cdot |V(x, y)| \cdot s_i(x, y) \cdot (1/E \cdot (E - abs(S_h)))$
**if** $C(x, y) >= S_s(x, y)$ **then**
    $diff \leftarrow S_s(x, y) + \Delta t \cdot K_s \cdot (C(x, y) - S_s(x, y))$
    $S_h(x, y) \leftarrow min(S_h(x, y) + diff, E)$
    $S_{s1}(x, y) \leftarrow S_s(x, y) - diff$
**else**
    $diff \leftarrow S_s(x, y) + \Delta t \cdot K_d \cdot (C(x, y) - S_s(x, y))$
    $S_h(x, y) \leftarrow max(S_h(x, y) - diff, -E)$
    $S_{s1}(x, y) \leftarrow S_s(x, y) + diff$
**end if**
```

Sediment transportation

Now we can move the sediment that we have suspended with the following equation.

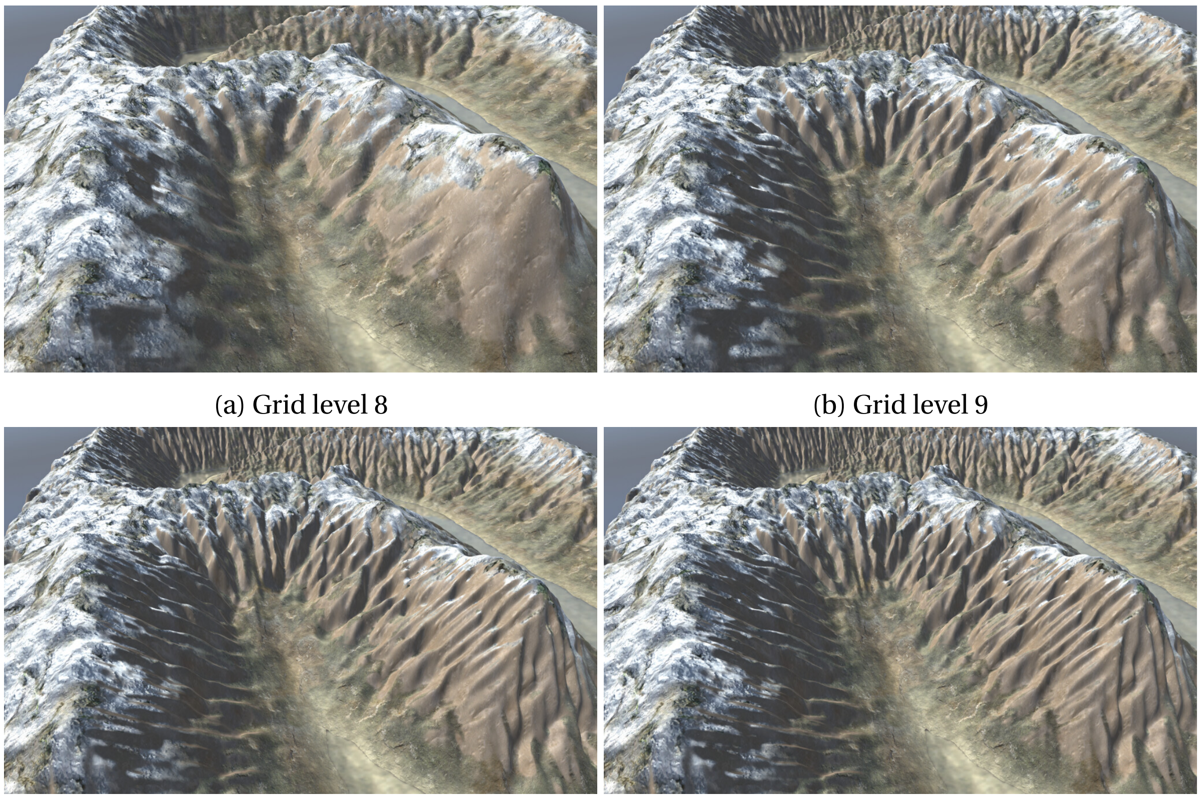
$$S_s(x, y) = S_{s1}(x - V_x(x, y) \cdot \Delta t, y - V_y(x, y) \cdot \Delta t) \tag{3.27}$$

Let p be the vector $(x - V_x(x,y) \cdot \Delta t, y - V_y(x,y) \cdot \Delta t)$. To solve this equation we sample the closest four points by rounding the first and second parameters of p up and down. We sample each of these four points from S_{s1}. Then we weigh each sample based on the distance between p and the four samples. The result is a weighted average between the closest points. This moves some of the sediment in the direction of the velocity.

Water evaporation

Lastly, we update the water height by multiplying it with the evaporation constant. This is the only way to remove water from our simulation as water can not flow out of the grid. We update the water amount, S_w, for the next time-step based on the time-delta, Δt, and the water amount of the current time-step. As mentioned earlier, we do not write this in our equations but it should be there as there is a Δt in the equation.

$$S_w = S_w \cdot (1 - K_e) \cdot \Delta t \tag{3.28}$$


(a) Grid level 8 (b) Grid level 9

(c) Grid level 10 (d) Grid level 11

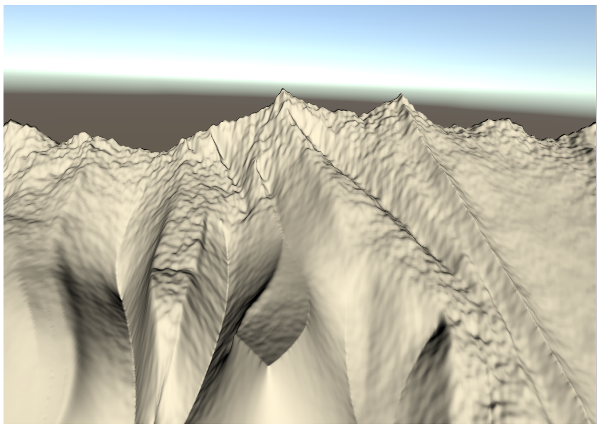
Figure 3.14: Grid levels in a multigrid erosion

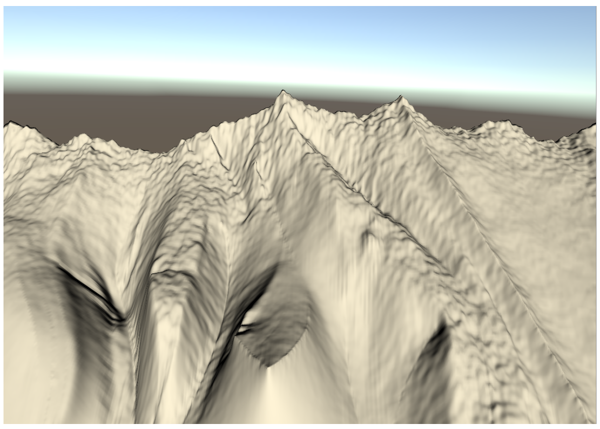
3.7.2 Multigrid erosion

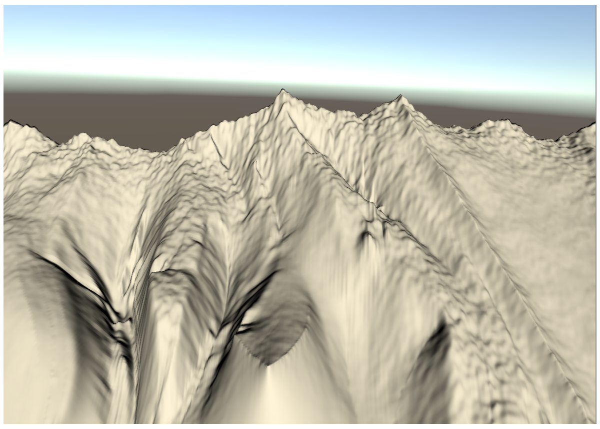
As erosion is computationally expensive we have implemented the concept of multigrid solving as described earlier and applied it to erosion. We erode the terrain at a smaller grid, interpolate the state of the erosion to a bigger grid and repeat the process. This integrates well with our diffusion as we already have created the terrain at different grid sizes. We store the height of the eroded terrain as the original height in one texture and the difference in a separate texture. We evaporate the water by setting evaporation variable $K_e = 0.9$ and run the erosion for 100 iterations on the largest grid. This ensures that no artifacts are created, all the water is evaporated and all the suspended sediment is deposited. Figure 3.14 shows the erosion on all grid levels larger than 7.

3.7.3 Diffused parameters erosion

To make the erosion easier to control we added two parameters to the meta points to help control the erosion process; w_i and s_i. The w_i is multiplied with the added water in the water increment step and s_i is used when calculating C in the suspension and deposition step. This allows us to control how much if any water is added and how much sediment can be suspended. By using these parameters we can erode one part of a mountain and have no effects on the rest of the terrain. In other words, we can erode specific parts of a terrain. In Figure 3.15 we show different renderings of a terrain where we changed the evaporation constant K_e. A lower evaporation constant gives stronger erosive features. Figure 3.15a is rendered without any erosion. This terrain has low values for both w_i and s_i on both sides of the terrain and high values in the middle. This results in erosion in the middle and no erosion on the sides.


(a) No erosion


(b) Little erosion


(c) Medium erosion

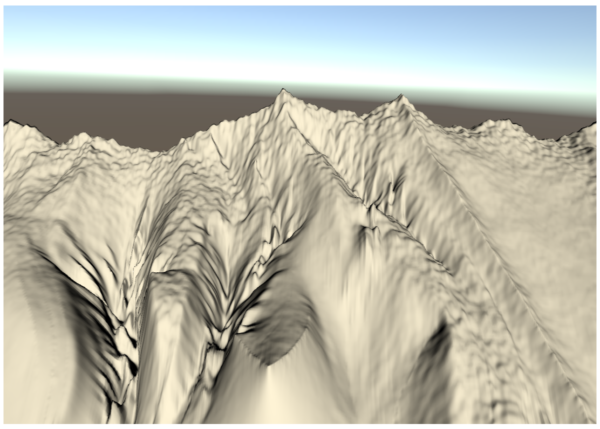

(d) High erosion

Figure 3.15: Different levels of erosion of a mountain side

Chapter 4

Implementation

In this chapter, we first look at our data representation and our rasterization process. Then we explain how we implemented our diffusion equation, how we generate noise and how we erode the terrain using compute shaders. Finally, we explain how we implemented this in the Unity 3D game engine and show the user interface.

4.1 Data representation

As in earlier work in this field [1][22] we store the data as textures after being rasterized. Now there are more powerful and flexible GPUs available. We could store our data as buffers and freely access them, but we defined performance and memory optimizations as out of scope for this thesis. We store our data as textures all using 32-bit float precision per channel. We used in total nine textures to represent our data. We used the following three textures for the diffusion:

- heightmap
- normal vector scalar field with gradient direction
- restrictions

For erosion we used three textures:

- state
- flux
- velocity

Finally we used three textures for our parameter diffusion:

- noise
- warp
- erosion

For our diffusion of the terrain, we stored the diffused heightmap in one channel of the first texture, heightmap. Our second texture, normal vector scalar field with gradient direction, stored the two normal vector components in the first two channels, the gradient direction in the third channel and the mask in the fourth and final channel. We use a mask to denote where the seed values are located for the diffusion which we store in the first two channels of the restrictions texture.

For the erosion, we stored the height difference, water height, and suspended sediment amount in the first three channels of our state texture and use the fourth to temporarily store the sin angle and the sediment amount. The flux stores the water flux, one channel for each direction. Finally, the velocity texture stores the water velocity in the first two channels.

For our parameter diffusion, we stored the appropriate meta point parameters from the meta point in 3.1 in each texture.

In addition to these textures, we temporarily store the calculated noise in a separate texture. In total, we use 9 textures for our data representation after rasterization. All textures are used in the multigrid solver so we need smaller versions of these textures. The result of this is that each texture requires almost twice the memory of the largest texture in use.

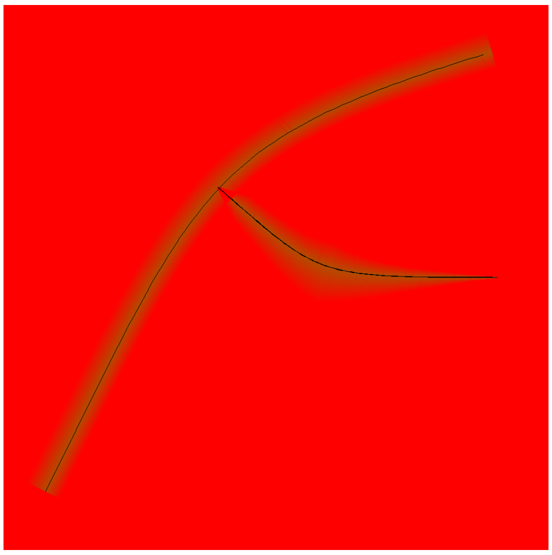
4.2 Rasterizing

In order to use the multigrid solver, we need to rasterize our splines onto discrete grids.

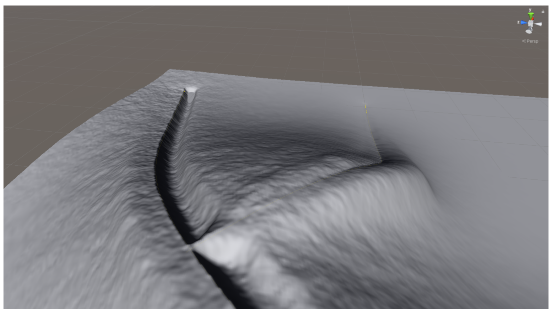
The splines have a line with a width of one pixel and possibly a varying width given by r_i and a varying gradient on both sides given by a_i and b_i. As the splines are rasterized to sizes between 2048x2048 and 4x4 we rasterize both the one-pixel wide line and the varying width and gradients.

The rasterization is done by sampling the spline relative to the resolution of the grid. For rasterizing the one-pixel line we used the Bresenham line algorithm to find all pixels and interpolate the values linearly. The spline radius and gradients were rasterized as quadrangles with different colors on the corners that were interpolated using barycentric coordinates. One-pixel lines and quandrangles are used and combined for rasterizing all the data.

Figure 4.1 shows a restrictions texture and a rendering of the terrain. There are two splines with guiding gradients on both sides. In Figure 4.2 we have the heightmap, normal vector scalar field and noise parameters after diffusion for the same terrain.

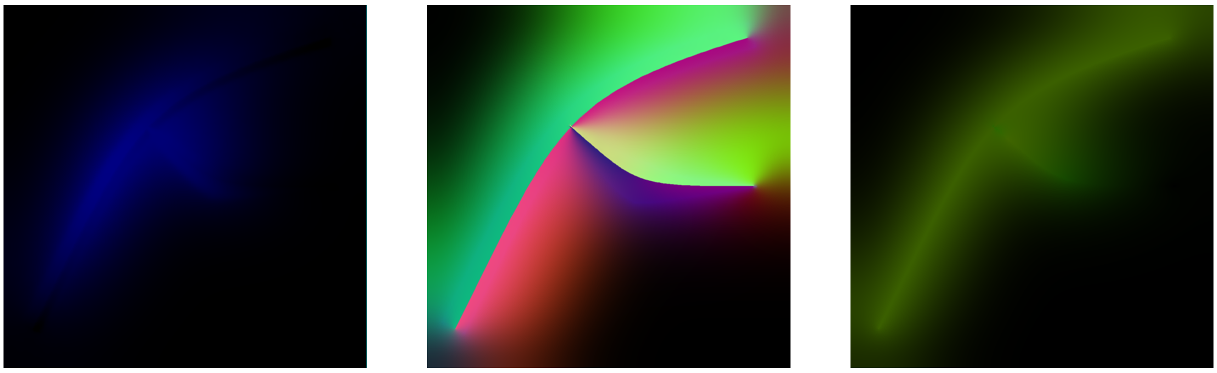


(a) The restrictions texture showing the α and β values as red and green



(b) The rendered terrain

Figure 4.1: Restriction texture with a rendering of the generated terrain



(a) The heightmap after diffusion (b) The normal vectors and gradient values after diffusion (c) The diffused noise parameters

Figure 4.2: The diffused grids for Figure 4.1

4.2.1 Elevation data

The elevation data is the value of the y component of each point along the spline divided by the maximum height of the terrain which is defined by the user. This data is used as seed values in the diffusion of the terrain and is needed on all pixels along the spline and inside of the spline radius triangles. The value is interpolated normally and when multiple values overlap we take the average value.

4.2.2 The normal vector scalar field

The normal vector scalar fields are rasterized using the one-pixel line along the spline. On each pixel, we calculate the perpendicular direction of the spline in the x and z-direction. Let the normalized value of the direction be d and the pixel along the spline be p. We color two pixels for each p along the spline. We set the color of the first point, $p1 = p + d$, to d. That means the first component of d is the red color and the second component is the green color. We set the second point, $p2 = p - d$, to $-d$. Each component of our pixels can have values in the range $[-1, 1]$. We map this range to $[0, 1]$ because our diffusion only converges for positive values.

This method is inspired by Orzan *et al.* [22], however a noteworthy enhancement is that we formulate the diffusion as a Laplace equation by setting f, from our diffusion equation in section 3.4, to 0. Orzan *et al.* set it to the rasterized average line. By not setting a gradient we do not know when we converge on a solution, but we save space and time as we do not need to calculate the error. Hnaidi *et al.* [1] did formulate it as a Laplace equation which is consistent as they normalize the results. The intensity of the values in the solution is not important, the relative intensities are.

4.2.3 Guiding gradients

The gradients defined by a_i and b_i are rasterized as quadrangles on both sides of the spline. They start where the radius quadrangles end. Their length is the x component of the vectors and is always perpendicular to the spline direction. Between the meta points the length is linearly interpolated between $a_i.x$ and $b_i.x$. The values of the vertices are linearly interpolated between $a_i.y$ and $b_i.y$. When the gradient overlap itself or any other gradient the value can be very different. To keep consistency in the terrain, we remove all pixels with overlapping gradients and use diffusion to fill in the holes.

4.2.4 Restrictions

Restrictions are used to weigh the three different formulas in the diffusion step for the terrain and contain two values, α, and β. Wherever there are elevation data we use $\alpha = \beta = 0$. Where there are guiding gradients the value is linearly interpolated between $\alpha = 1 - \beta$, where $\beta = h_i$ is closest to the spline and $\alpha = \beta = 0$ is furthest away from the spline. h_i is interpolated between meta points on the spline. This ensures a smooth transition between the guiding gradient and the rest of the terrain. The rest of the grid is filled with $\alpha = 1, \beta = 0$.

4.2.5 Parameter data

The parameter data consists of two parameters each, for example R_i and A_i for noise, on meta points. This data is rasterized into a texture by rasterizing the splines and using the interpolated values as color values. When splines overlap we use the average value.

4.3 Diffusion and multigrid solver

We extend our diffusion equation with a local restraint and adjust our Jacobi relaxation term accordingly:

$$v_{i+1} = \alpha A v_i + \beta(B v_i + G) + (1 - \alpha - \beta)S \tag{4.1}$$

which is equivalent to the equation described in chapter 3 with the added local restriction:

$$\alpha = \beta = 0 \text{ if } S \text{ is set} \tag{4.2}$$

This fulfills the local restriction that $u = S$ if S is set, which we defined in chapter 3.

We run our Jacobi relaxation with two passes between two textures to avoid side effects as the relaxation uses neighboring values. As the number of cells in a grid grows exponentially with the size of the grid we increase the number of relaxations linearly.

$$iterations = grid\_level \cdot diffusion\_iteration\_multiplier \tag{4.3}$$

where grid level is the square root of the size of the grid. We use this equation to calculate the number of iterations we run the Jacobi relaxation on each grid level. The user can freely choose the *diffusion_iteration_multiplier*.

As the diffusion for our normal vector scalar field and parameters are less important we diffuse them with a four-point average kernel like Hnaidi *et al.* [1] does and use half as many iterations as we use for diffusing the terrain.

We interpolate by using the five-point Gaussian kernel from Equation 2.8 on the solution. We optimized this by using a red-green-black texture. The texture is created using Algorithm 3 and is available during interpolation. We included the pseudo-code for our interpolation kernel in Algorithm 4.

Algorithm 3 Pre-calculate modulo

```
  $t.r \leftarrow x(mod 2)$
  $t.g \leftarrow y(mod 2)$
```

Algorithm 4 Interpolation kernel

```
$center.x \leftarrow x/2$
$center.y \leftarrow y/2$
$color \leftarrow t[(x,y) mod(1024)]$
$result \leftarrow image[center]$
$result \leftarrow result + image[center + (1,0)] * color.r$
$result \leftarrow result + image[center + (0,1)] * color.g$
$result \leftarrow result + image[center + (1,1)] * color.r * color.g$
$result \leftarrow result/(1 + color.r + color.g + color.r * color.g)$
**return** $result$
```

4.4 Generating noise

We use improved Perlin noise as described in "Improving Noise" by Perlin [9] as the smooth noise function S. The function is implemented as a kernel in a compute shader and is used as described in our method section 3.6. We map the coordinates x and y based on the grid level. Let *gridWidth* and *gridHeight* be the size of the current grid size. We scale x and y down to map it to a smaller scale more fit for our smooth noise implementation. To make it independent of the grid level we multiply it with $2048/gridWidth$. We provide a *scale* parameter for the user which is multiplied with the input coordinates to our noise function to give the user more control, and *amplitude* which is multiplied with the noise value for the same reason. Pseudo-code for this is in Algorithm 5 and Algorithm 6.

Algorithm 5 Generate noise

```
$x \leftarrow (2048/gridWidth) \cdot (x/1000)$
$y \leftarrow (2048/gridHeight) \cdot (y/1000)$
$k \leftarrow 0$
$sum \leftarrow 0$
while $k < n$ do
    $s \leftarrow scale \cdot r^k$
    $sum \leftarrow sum + \frac{1}{r^{k(1-R(x,y))}} S(T_{pre}(s \cdot x, s \cdot y, k))$
end while
return $amplitude \cdot A(x,y) \cdot sum$
```

Algorithm 6 T_{pre}

```
$s_1 \leftarrow 50$
$s_2 \leftarrow 150$
$s \leftarrow k \cdot F(x,y) + \frac{D(x,y)}{k+1}$
return $p \leftarrow p + s \cdot [S(s_1 + p), S(s_2 + p)]$
```

4.5 Erosion

The compute shader differs from the fragment shader as it can take any amount of data in and write any amount of data out to buffers or textures. This lets us write to multiple textures in the same step which is not possible on a fragment shader. We should not read and write to the same texture if we compare values with neighboring values of the same texture. That would result in some cells updating using old values and other cells using new values which might break our simulation.

In the first kernel, we implement "Water increment". This needs to be calculated before the next step "Water flux" as water flux compares neighboring values including water height. "Water flux" is a second kernel. For the third kernel, we can include "Water update", "Water velocity" and "Local tilt angle". "Local tilt angle" has to be computed before "Erosion and deposition" as "Local tilt angle" compares the terrain height between cells and "Erosion and deposition" updates the terrain height. "Erosion and deposition" is the fourth kernel and "Sediment transportation and "Water evaporation" is the fifth and final kernel.

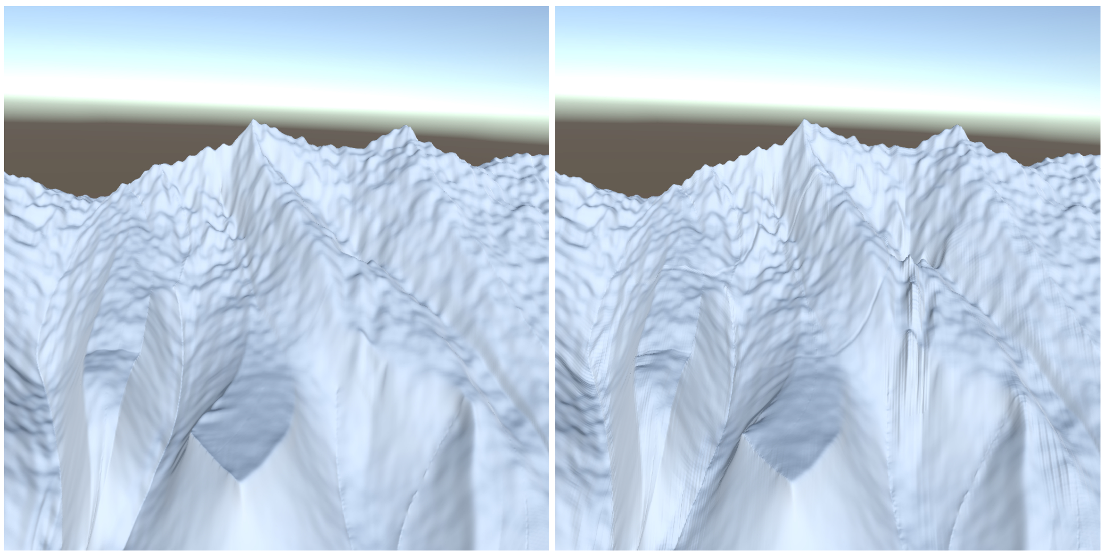
We store the eroded and deposited mass, water height, sediment amount, and local tilt angle in one texture, the output flux in another texture, and the velocity in a third texture. All textures are 32-bit float textures where the first two textures use four channels and the third texture uses two channels.

4.5.1 Multigrid erosion

We run the erosion on heightmaps of different sizes similar to how we solve diffusion. When a terrain is diffused at a grid level, we add noise to the diffused terrain before we use it for erosion. The erosion starts with empty textures of the smallest size. We run all the erosion steps for a user-defined amount of iterations. When we run erosion on any size bigger than the smallest, we interpolate all textures to the next size. This is done the same way as done in diffusion - by using our interpolation Algorithm 4. In addition to interpolating, we run a 5x5 Gaussian smoothing kernel from Figure 4.3 four times on the first three color channels of the state texture, which represent the deposited sediment, water height and suspended sediment, and the water flux texture. This removes most of the edge-like artifacts which are caused by the interpolation, as one can see in Figure 4.4. After all iterations on the largest grid, we set the evaporation rate $K_e = 0.9$ which removes 90% of the water each iteration and

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 4.3: 5x5 Gaussian kernel



(a) Multigrid erosion with smoothing (b) Multigrid erosion without smoothing

Figure 4.4: The effect of smoothing after interpolation in a multigrid erosion

run the simulation without adding water for 100 iterations. This ensures that all the water is removed and all sediment is deposited. Finally, we run the 5x5 Gaussian kernel four times to ensure no artifacts are added in the evaporation step. We provide a diagram outlining this process in Figure 4.5.

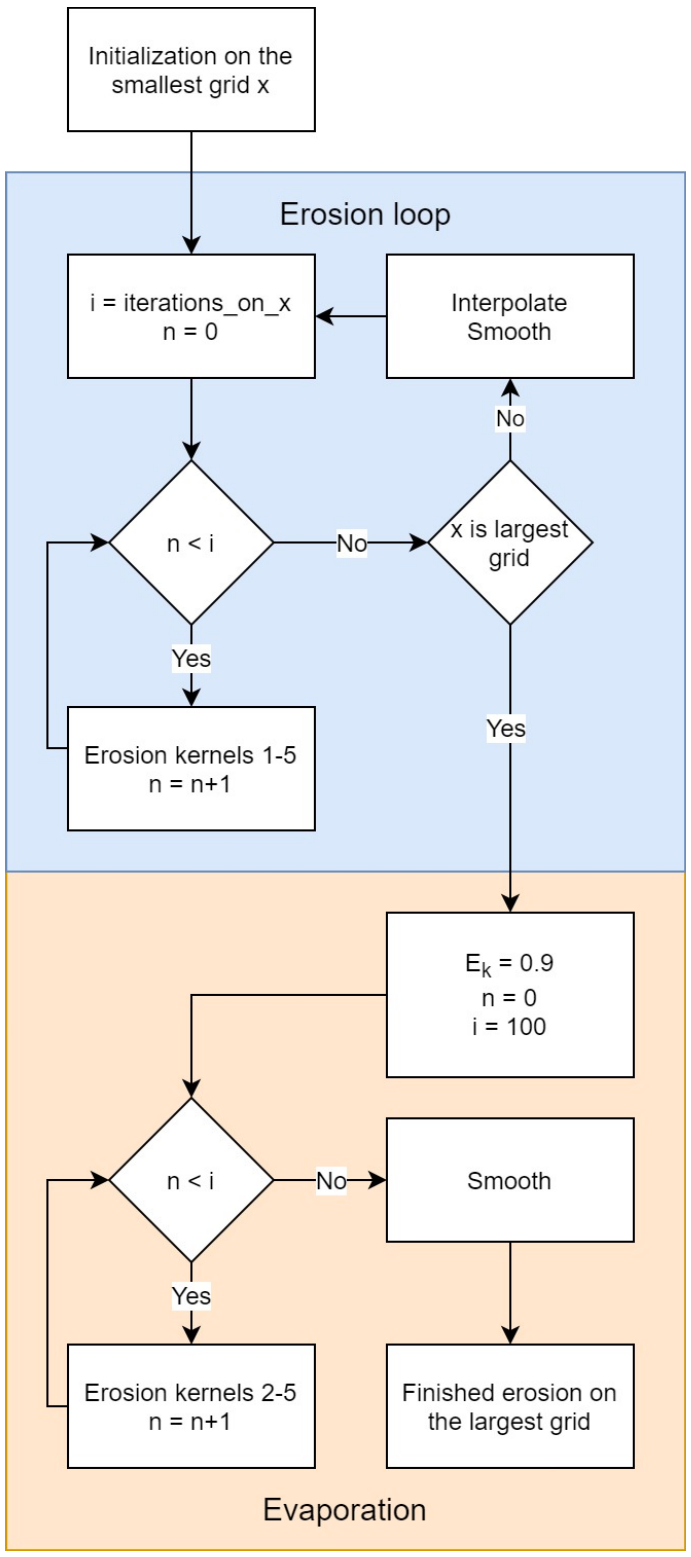


Figure 4.5: Overview of multigrid erosion

Note that erosion kernel 1 "Water increment" is not used in the evaporation.

4.6 Finalizing the terrain

To get the final terrain we sum the terrain height texture from the diffusion process with the sediment and deposition texture from the erosion simulation.

4.7 Integration with a modern game engine

In this section we introduce the Unity 3D game engine and show how we integrated our method with the engine. Lastly we explain how we added texture to our terrains when rendering the results.

4.7.1 Unity 3D game engine

Unity 3D is a game engine to design games and applications for computers, virtual reality, consoles, and mobile devices. It is commercial software with free use for individuals[1] developed by Unity Technologies. The engine has three important components: the game engine, a visual editor, and a code editor.

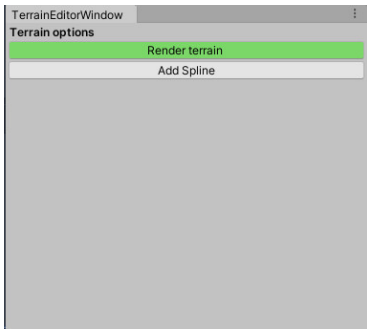
For our implementation, we used the visual editor and the code editor. We used the code editor to write all the code in the project and integrated our code with the visual editor. The feature splines are drawn in 3D and are selectable. When a spline is selected all points defining the Bézier curves are visualized as well as all meta points along the spline. The Bézier curve points can be moved like any other object in Unity and for the meta points, the gradients and radius are visualized and can be changed visually. To change the meta points position on the spline and the noise values a custom editor window is available.

The resulting height map after the diffusion and erosion is copied to a built-in terrain asset in Unity that supports Level of Detail (LoD). This terrain is scaled and moved to align with the splines to visualize the effects of each feature spline.

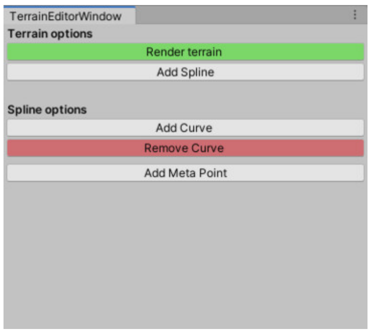
4.7.2 Editing the terrain

The input data to our generation method is defined by feature splines with meta points. These can be changed with the visual editor and from a custom window shown in Figure 4.6.

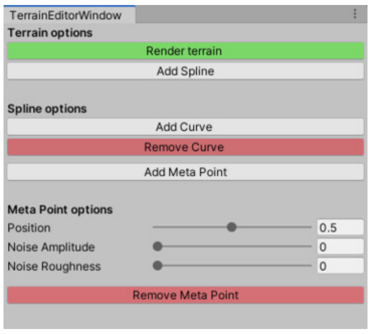
[1]With some restrictions



(a) When the terrain is selected



(b) When a feature spline is selected



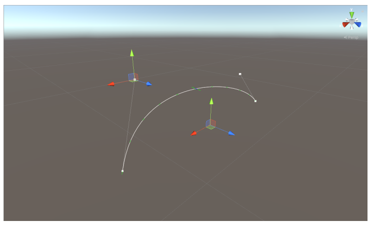
Position 0.5
Noise Amplitude 0
Noise Roughness 0

(c) When a meta point is selected

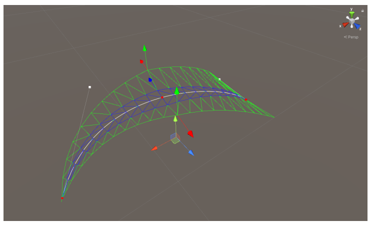
Figure 4.6: The custom editor window in three different states

To start editing, the user can press the "Add spline" button to create the first spline. It starts as a cubic Bézier curve and more curves can be added by pressing the "Add curve" button. After selecting the spline in the hierarchy or by pressing it in the scene window, the user can click on any of the control points and move them freely in any of the three-axis. When having a spline selected the user can press "Add meta point" to add meta points to the spline and then configure the meta points. Once the user is happy with the splines they can press "Render terrain" and the terrain appears based on the feature curves the user configured.

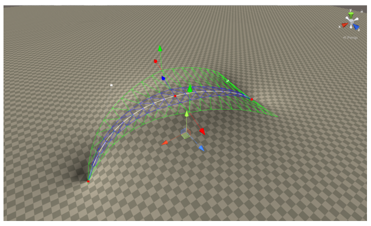
Figure 4.7 shows a feature spline with and without meta points and terrain. White is the Bézier curve, blue is the radius, green is the gradients and the red squares are the location of the meta points.



(a) A feature spline in the visual editor



(b) A feature spline with three meta points



(c) The same spline as in b) with the diffused terrain

Figure 4.7: A feature spline with and without meta points and terrain

4.7.3 Textures used for rendering

To texture our fantasy terrain we used procedural texturing. We used five textures with normal maps to color the terrain. Additionally, we provided the final terrain heightmap texture and final restrictions texture to the shader. We combine our textures by applying different

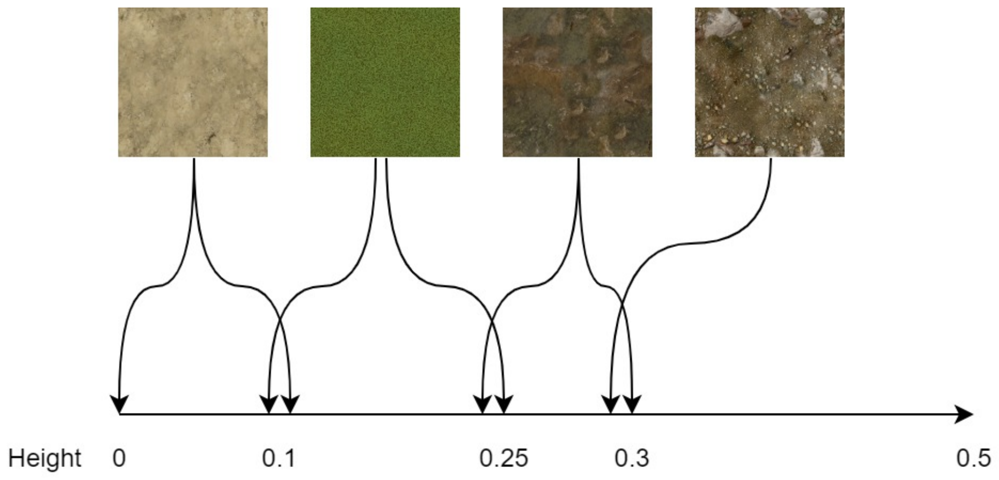


Figure 4.8: Texture weights based on height

weights to each texture where the sum of all the weights equals 1. We use three steps to calculate the weights based on the terrain height, the terrain steepness, and the restrictions texture.

When calculating the weights depending on the height we decide on what height each texture starts and how smooth the transition is between that and the previous texture. Figure 4.8 shows what textures we used and approximate overlap and area where the texture is applied.

We calculate T_r as this has the highest priority. This is calculated from the restrictions texture to texture roads. Let r be a function sampling the restrictions texture.

$$T_r(x, y) = r(x, y) \tag{4.4}$$

Then we calculate steepness, $s(x, y)$, by sampling with this equation:

$$\begin{aligned} s(u, v) = &clamp(0, 1, \\ &|h(u, v) - h(u + 0.001, v)| + \\ &|h(u, v) - h(u - 0.001, v)| + \\ &|h(u, v) - h(u, v + 0.001)| + \\ &|h(u, v) - h(u, v - 0.001)|) \end{aligned} \tag{4.5}$$

$$clamp(a, b, x) = max(a, min(b, x)) \tag{4.6}$$

where u and v denote the position on the texture scaled down to between 0 and 1. Let h be a function sampling the height of the terrain from the texture. Then we calculate the weight of the steepness texture T_s.

$$T_s = T_r \cdot ((s - s_{limit}) + (s < s_{limit} \cdot s > (s_{limit} - s_{smooth})) \cdot \frac{s - s_{limit} + s_{smooth}}{s_{smooth}}) \tag{4.7}$$

where s_{limit} and s_{smooth} could be any number between 0 and 1. The conditionals of the equation evaluate to either 0 if untrue or 1 if they are true. We calculate the restriction weight, T_r, by sampling the restrictions texture to get α used in the diffusion and setting $T_r = 1 - \alpha$.

Then we need to scale our weights so that the sum is equal to 1. Let w_a be the weight for a texture a on a point with height h. We calculate the texture weight T_h by scaling down the weight based on T_r and T_s so that $T_r + T_s + T_h = 1$ for that point.

$$T_a = w_a \cdot (1 - T_s) \cdot (1 - T_r) \tag{4.8}$$

Now that we have calculated and scaled all the weights, we can sample each texture and multiply the texture value with the weight. The sum of this is our color value for the pixel.

Chapter 5

Evaluation

5.1 Overview

In this chapter, we present five terrains generated with our method. Each terrain is designed to showcase one or more of the differences between our and other methods. The terrains were designed by the lead author of this thesis who has never designed any terrains before. All of our terrains are generated as a texture with 2048x2048 pixels and rendered with procedural textures. Finally, we show terrains created with other methods and compare them to our terrains.

For the purposes of reproducing our results, we provide tables 5.1, 5.2 and 5.3 with the parameters we used to generate each terrain. We also provide the time spent generating the terrain with and without erosion in tables 5.4 and 5.5.

We evaluate the terrains by comparing them to results using other methods, by realism compared to images of real terrain, and theoretical and empirical analysis of repetitiveness.

5.2 Fantasy terrain

The fantasy terrain is our main result and is inspired by the southern part of Cape of Stranglethorn in World of Warcraft [4]. With this terrain, we tried to make a terrain designed for a game like World of Warcraft. These terrains usually have roads leading to places where the player can interact with a character or an object. Other places are usually populated with monsters, bandits, or friendly creatures. This inspired the design with two beaches, two grasslands, a volcano, a bay, and a road into the side of a mountain.

We have rendered images from six different locations with and without erosion. For this terrain, we will focus on three aspects. Firstly on the ability to limit splines to certain grid levels to create a smooth terrain with distinct features and roads. Secondly on how the repetitiveness is mitigated by erosion and warp. Thirdly on limiting erosion to avoid eroding roads, islands, and the volcano.

This terrain is our most detailed experiment. It took about four hours to design by a novice designer using 38 feature splines. With efficient storage, it uses about 1.5 kB of space and it takes 1.5 seconds to generate it at 2048x2048 resolution.

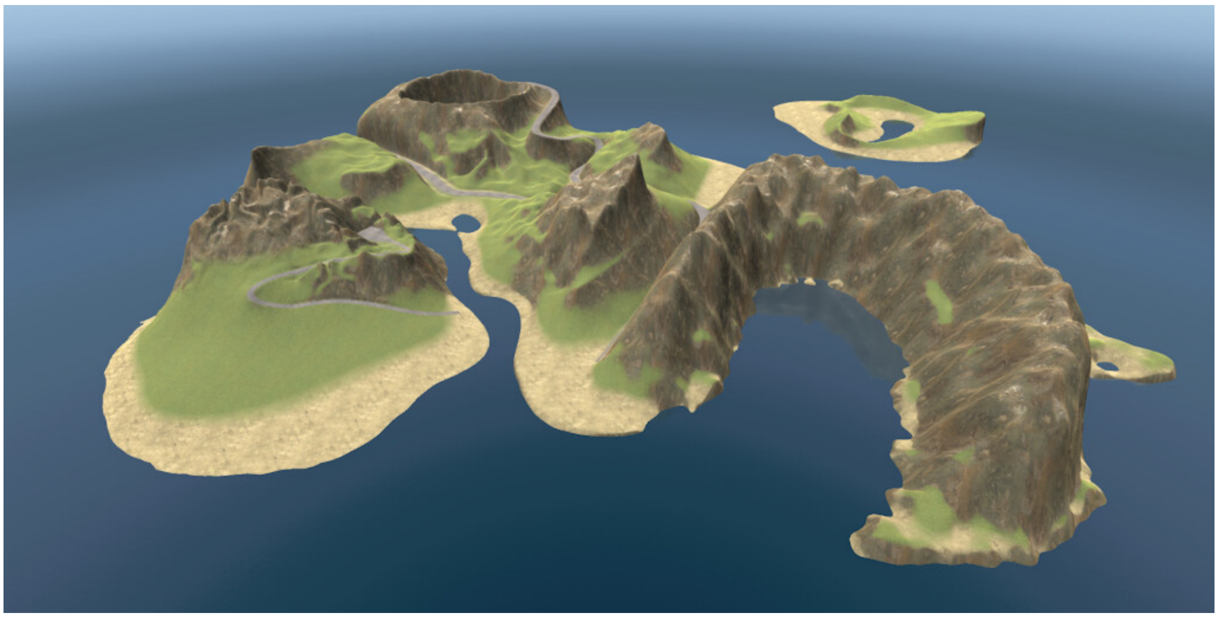


Figure 5.1: Fantasy terrain

Firstly the bay prominent in Figure 5.1 is surrounded by tall and rough mountains. These mountains are created by one feature spline which is only defined at grid levels 2-8 with maximum noise amplitude and roughness. Inside the bay, there is a spline beneath the water level making the mountains very steep. The mountain is also very steep on the left side of the bay caused by a spline separating the bay mountains and the mountain in the center of the image. Creating such a mountain without using spline deconstruction would involve using a large flat gradient on both sides of the mountain spline. This is how Hnaidi *et al.* described generating a hill in their paper [1]. In this case, it would be difficult to use gradients as the splines are too close. When the splines are too close, the normal vectors change direction which can lead to artifacts and unintended features.

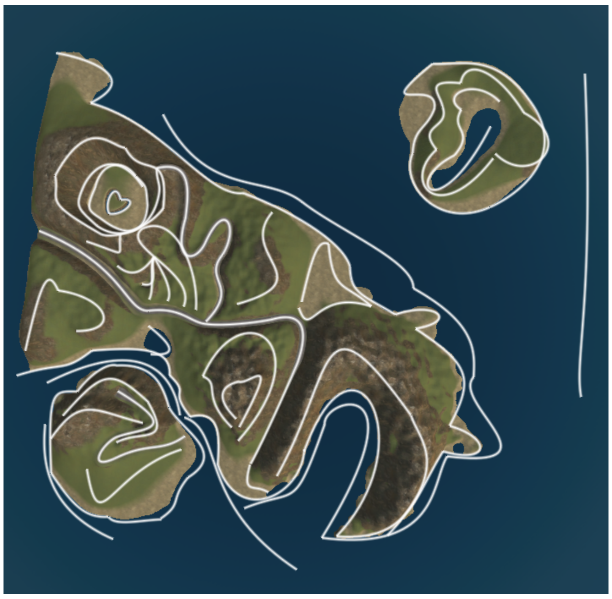


Figure 5.2: Fantasy terrain with feature splines in white

Figure 5.2 shows the positions of the splines used to generate the terrain.

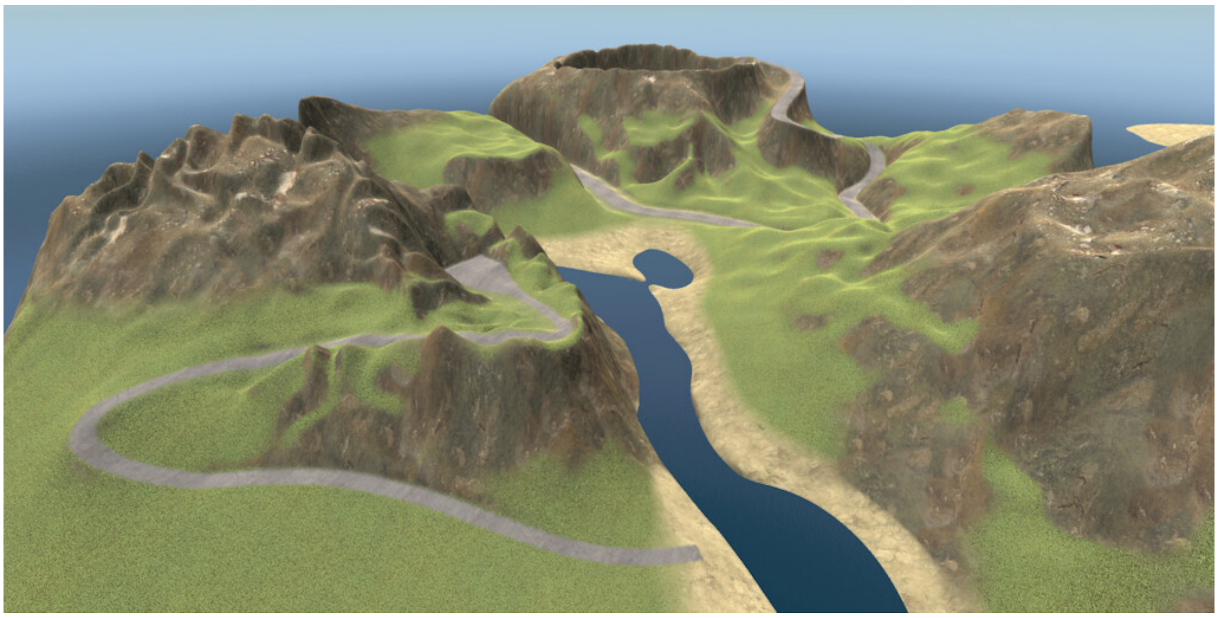


Figure 5.3: Fantasy terrain - river

Figure 5.3 shows the roads, river, and volcano. We can see a road in the foreground going from the river up the hill where the road becomes wider before it stops. This wider road represents a point of interest and shows how one can use the meta points to change the width of the spline. On this island, we added warping to make the noise look different from the rest of the terrain. The river is created with three splines, one for each riverside and one for the middle. The circular water body in the middle of the image is created with one spline simply lowering the terrain. Above this body of water, we used seven splines to make the side of the volcano uneven. Additionally, there is a road going up to the volcano which we look closer at in Figure 5.4.

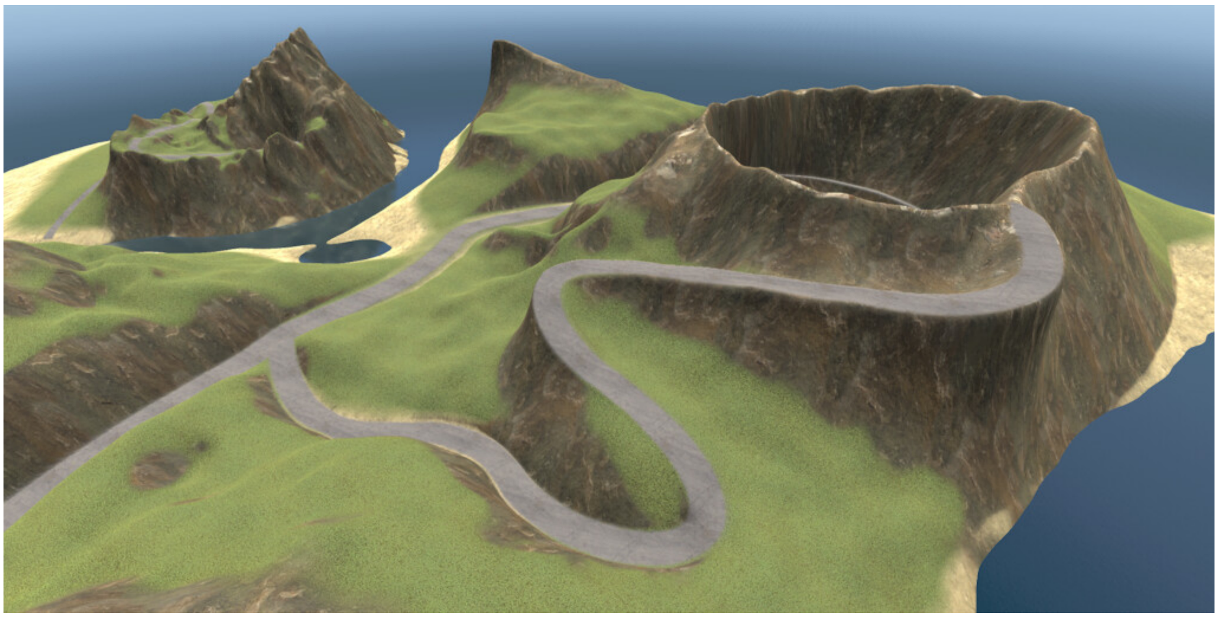


Figure 5.4: Fantasy terrain - volcano path

The prominent path in Figure 5.4 is only defined at grid levels 8-12 which results in very steep sides of the road. By using only the high grid levels one can easily create features in the terrain which do not look natural, but rather look like humans have been there and modified the terrain to their liking. In this case by creating a path to the volcano. The next two images are examples of this.

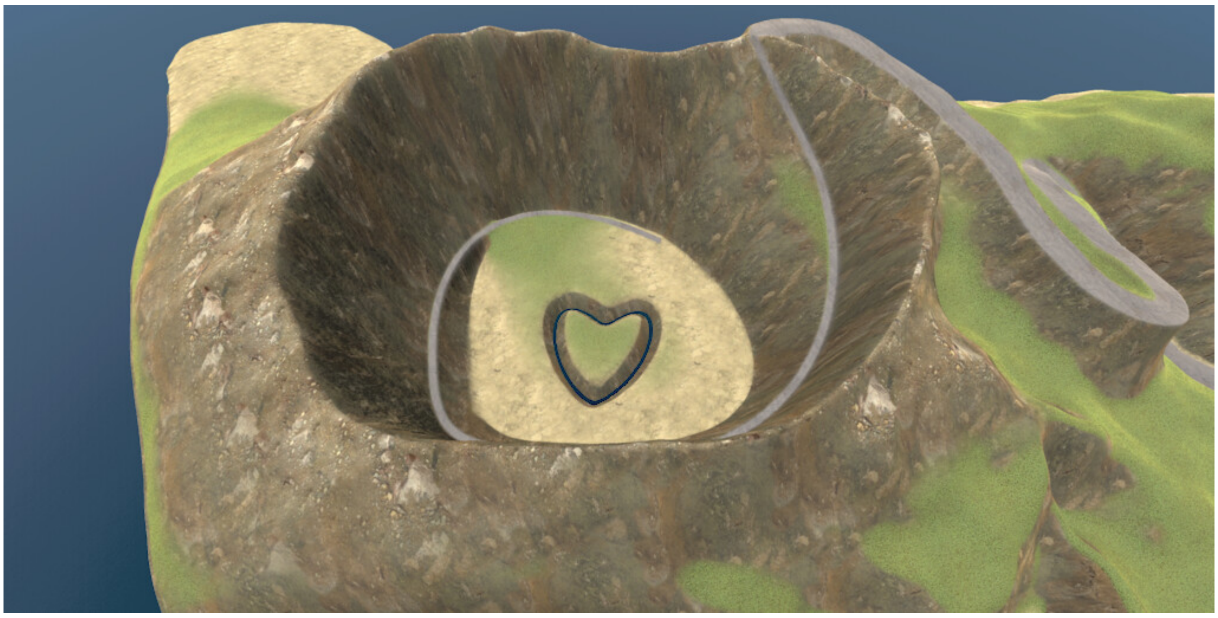


Figure 5.5: Fantasy terrain - volcano center

Figure 5.5 shows the inside of the volcano where the path from Figure 5.4 continues down to the bottom. At the bottom, we formed a heart from a spline which is only defined on the grid levels 10-12. Since the spline is only defined on the two largest grid levels it does not affect the terrain inside of the heart.

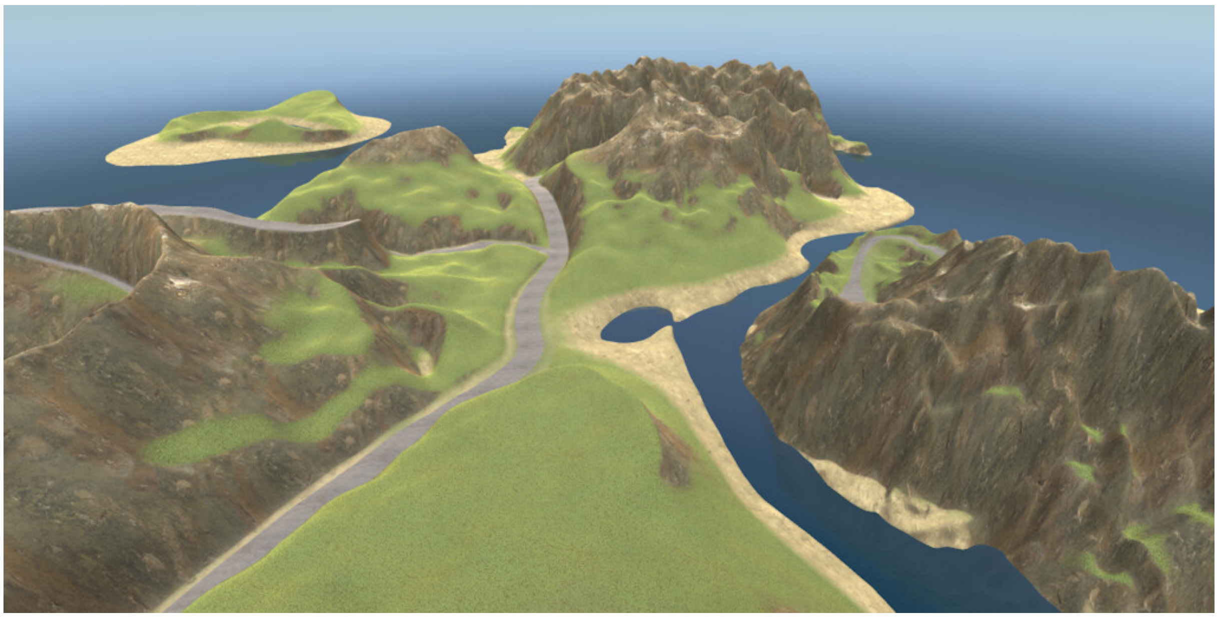


Figure 5.6: Fantasy terrain - the road

We wanted to make it look like humans created this road by carving out a path in the terrain.

Figure 5.6 shows the path towards the bay. The spline creating the road has a downwards gradient of both sides close to the camera which slowly disappears. There is a lot of noise added on both side of the road after the gradient disappears. This noise is not warped and looks quite repetitive without erosion. Figure 5.7 shows the same area after heavily eroding the terrain. Most of the repetitive noise is eroded and hydraulic erosion details are added.

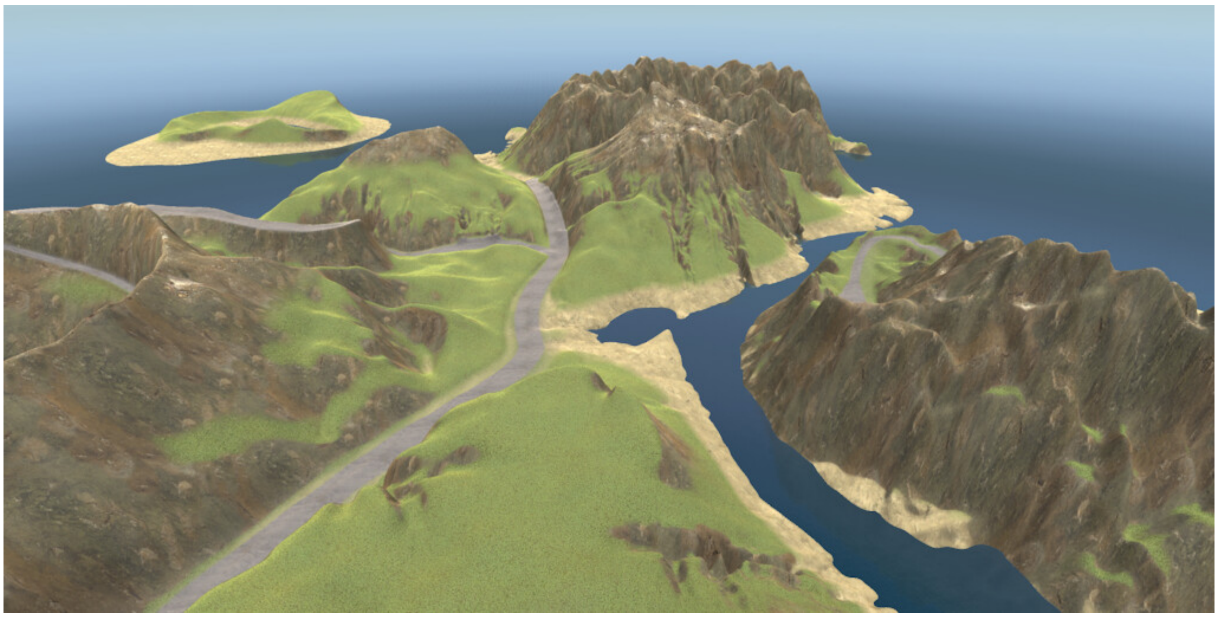


Figure 5.7: Eroded fantasy terrain - the road

We simulated a lot of erosion to most of the terrain. We eroded the main island the most and reduced the erosion on the other islands and on the road. The road is not eroded, but has received some sediment. To avoid eroding the roads we did not erode on grids levels smaller than 9 because the roads were not added before grid level 8.

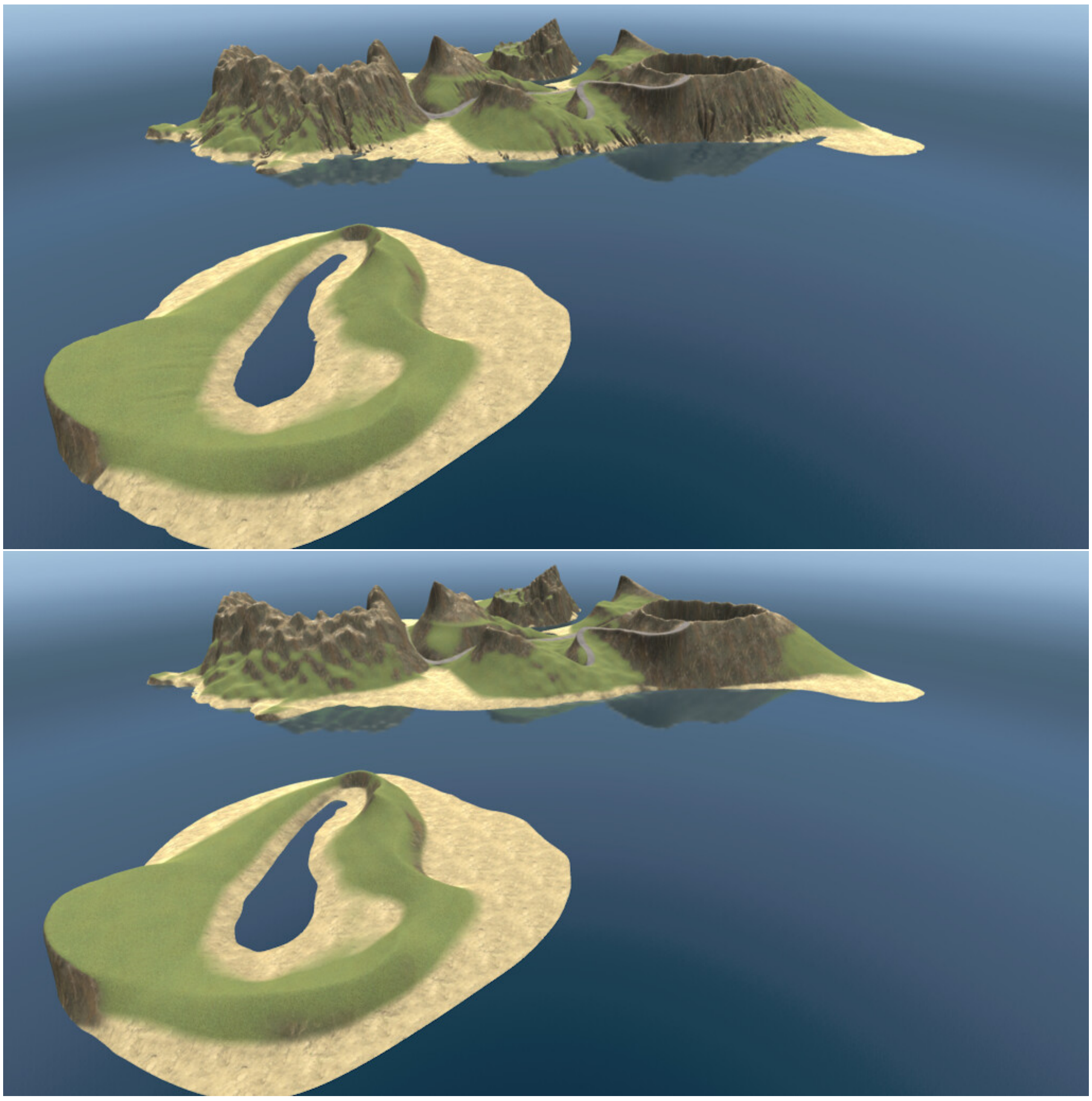


Figure 5.8: Fantasy terrain with and without erosion from another angle

Figure 5.8 and Figure 5.9 show the difference with and without erosion of the terrain. The second image has a lot of repetitiveness in the mountains surrounding the bay which is mostly mitigated by the erosion. There is no erosion applied to the smaller islands, the roads, and inside of the volcano. Figure 5.10 shows the eroded terrain from four angles.

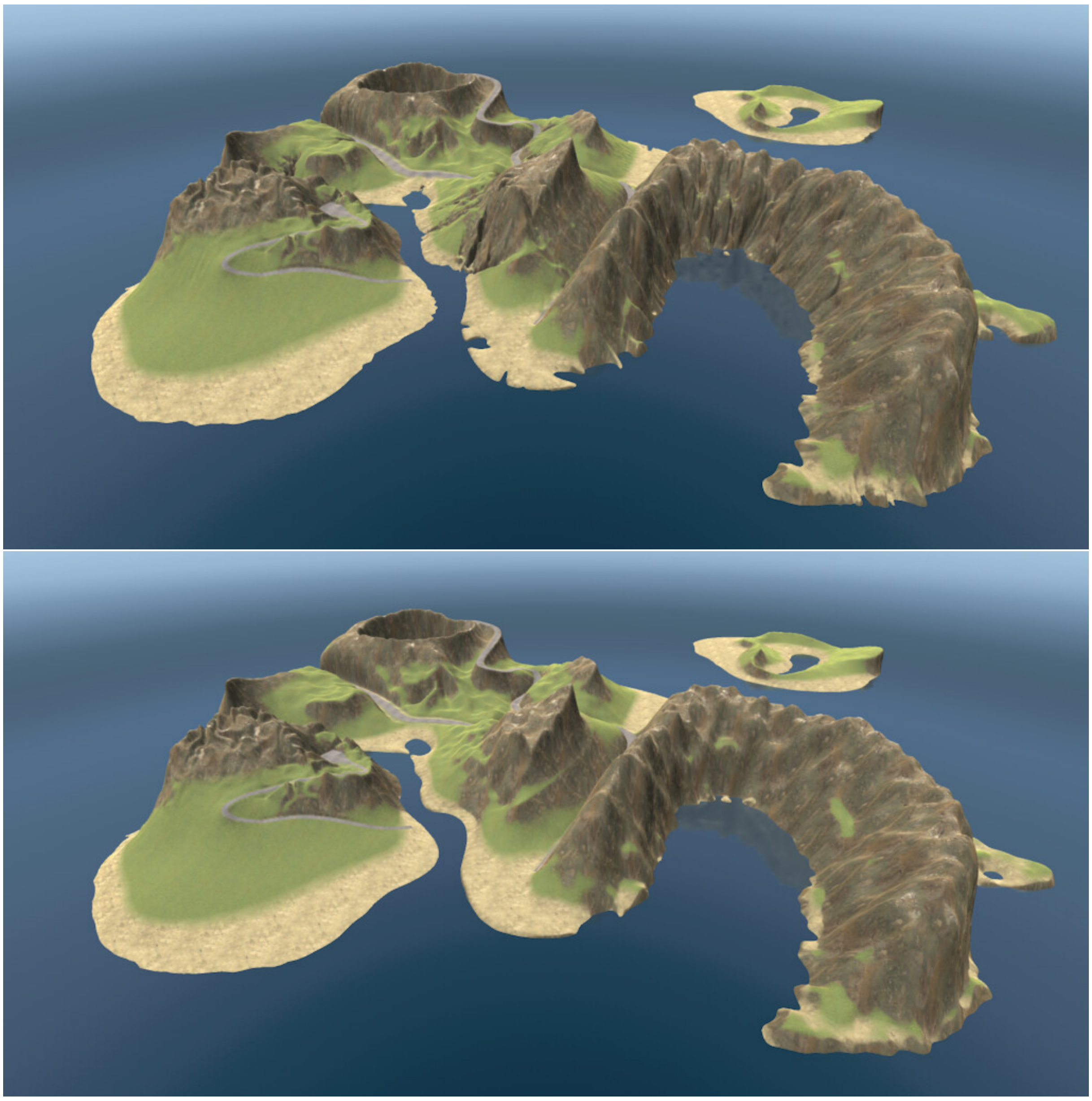


Figure 5.9: Fantasy terrain with and without erosion

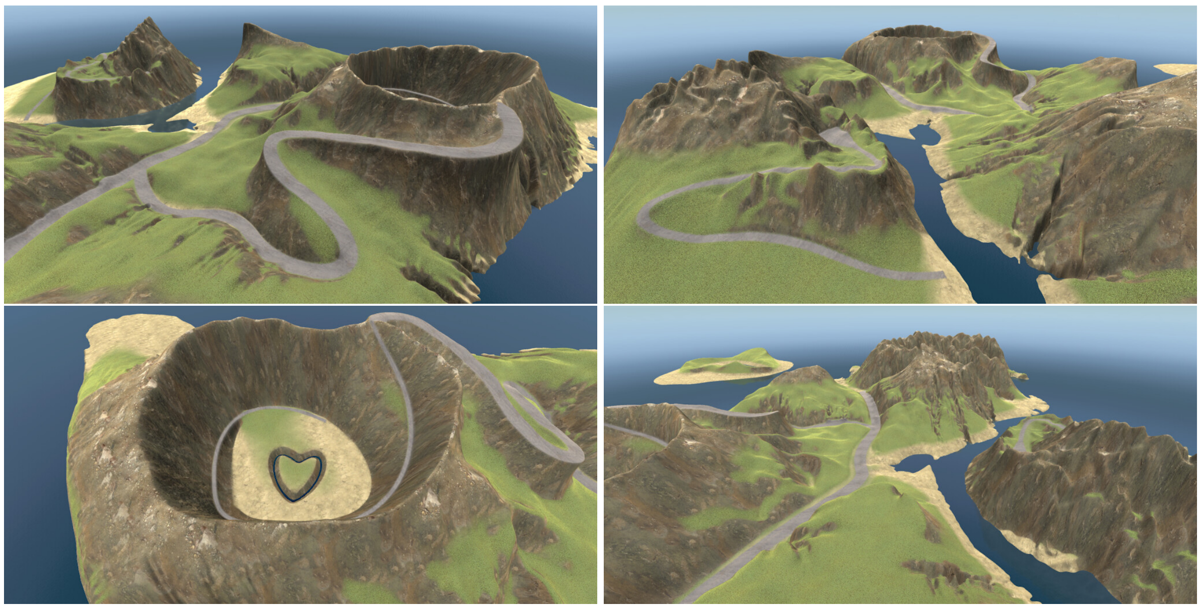
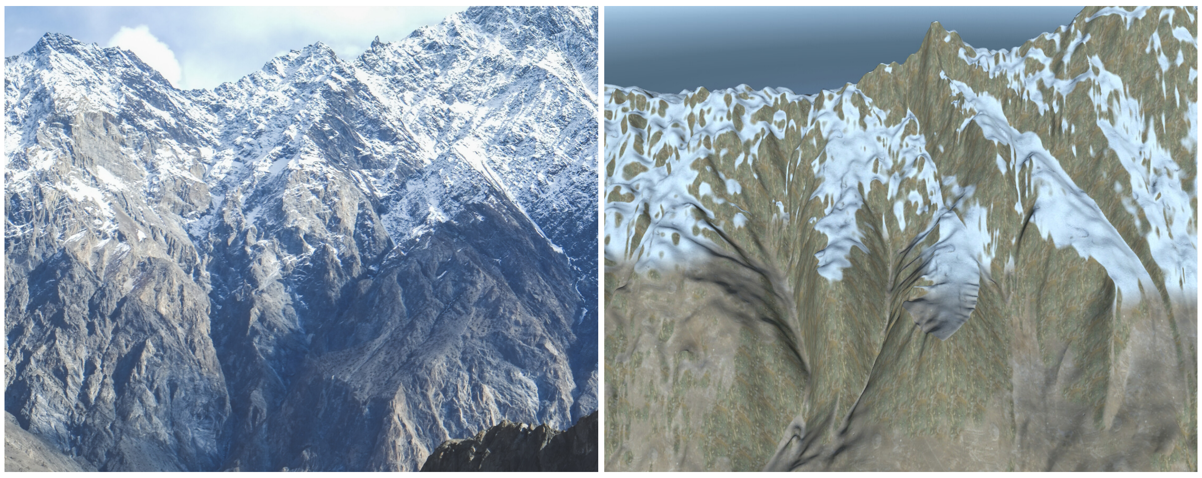


Figure 5.10: Eroded fantasy terrain from different angles

5.3 Mountain range

Our second terrain is modeled after an image of a mountain range. As previously stated we have not designed terrains before nor consider ourselves good at design. The proportions of the features are not similar to the features in the image. Arguably this is not a limitation of the method, but rather our lack of design skills. Figure 5.11 shows our inspiration next to a rendering of our eroded terrain.



(a) Mountain range photography

(b) Our terrain

Image (a) by suolzone100491 at Vecteezy.com

Figure 5.11: Inspiration for the terrain

This terrain shows how one can recreate images and use erosion to greatly enhance the results. In the middle of the image in Figure 5.11, we can see the effects of erosion where some bulks of mass have been moved from the higher altitudes. Figure 5.12 shows a rendering of the full mountain range with and without erosion. To better see the effects of erosion we rendered the terrain with erosion on different grid levels in Figure 5.13. We created this mountainside by using only 27 curves which use about 0.8 kB of efficient storage space. Most of the erosion happened at grid level 8. We did this to get larger erosion details. We added a lot of water to this simulation which initially caused unnatural features. To remove these features we used low values for our erosion parameters, w_i and s_i, on the lower feature splines and higher values on the higher splines.

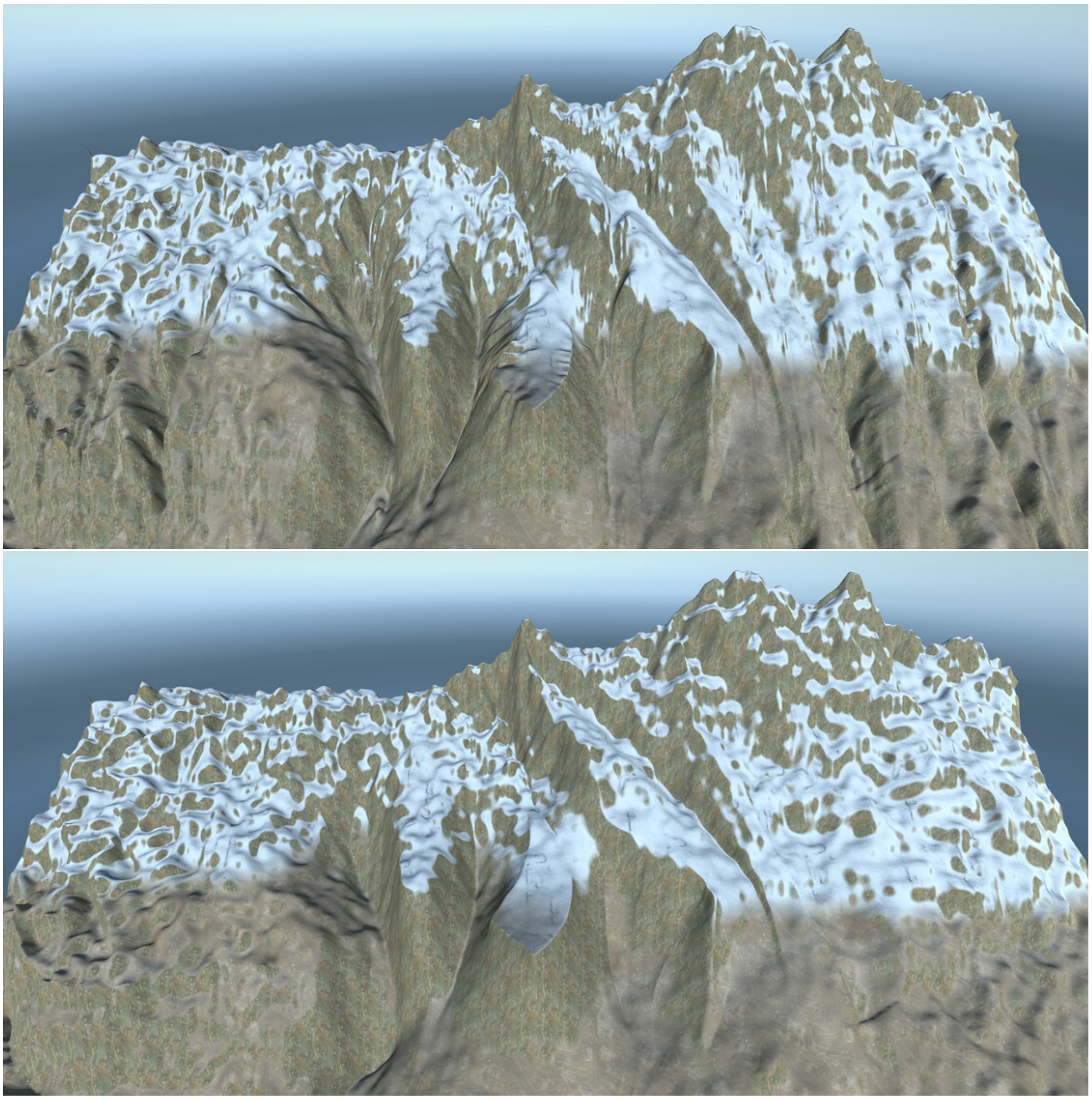
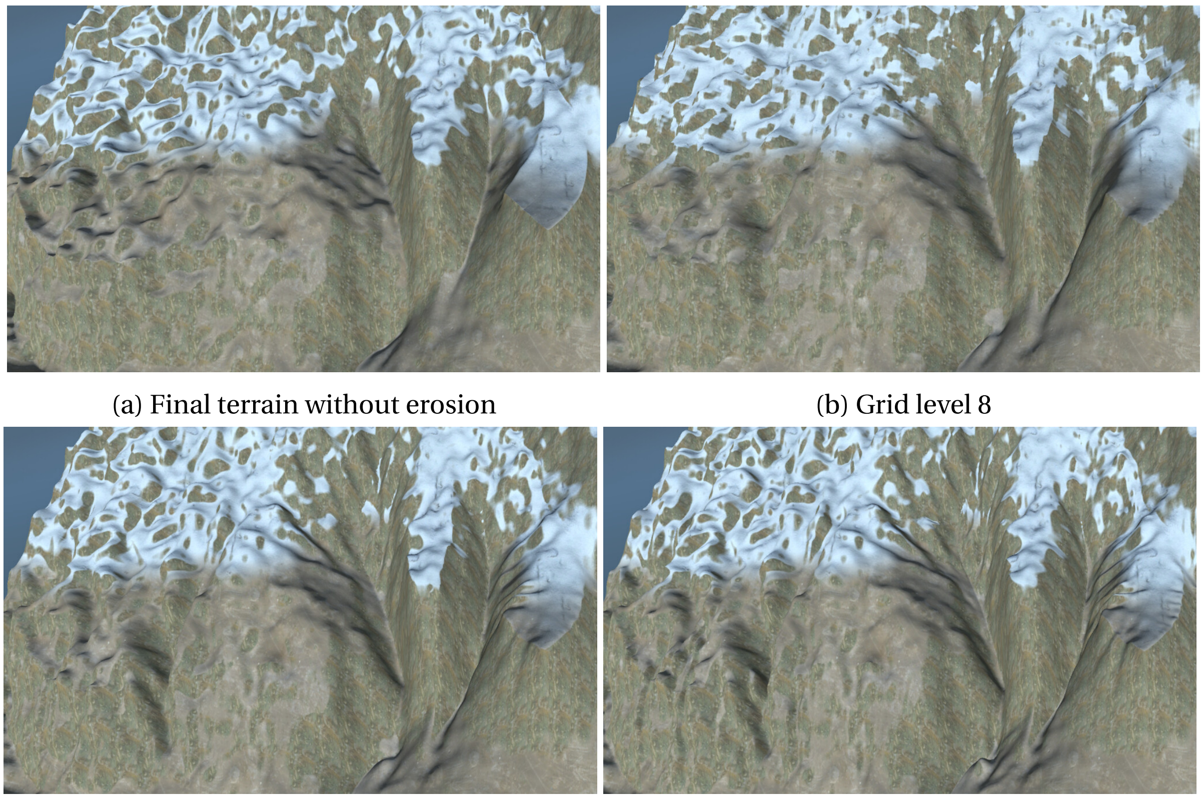


Figure 5.12: Mountain range with and without erosion



(a) Final terrain without erosion (b) Grid level 8

(c) Grid level 9 (d) Grid level 10

Figure 5.13: Lower resolutions of mountain range

Note: Lower resolutions are automatically scaled up by Unity 3D terrain to 2048^2

5.4 Mountain river

The terrain in Figure 5.14 is designed to be comparable to other methods for generating terrain. We modeled it with a similar motif, scale, and features as the other terrains we use for comparison in section 5.8. Though this is not what feature-based terrain excels at, it is something simulation methods are great at. Because of this, we created a very simple terrain with our feature splines and focused on erosion. We made this terrain in less than an hour with most of the time spent on erosion. It has the most erosion with 5500 erosion iterations in total over all the grid levels.

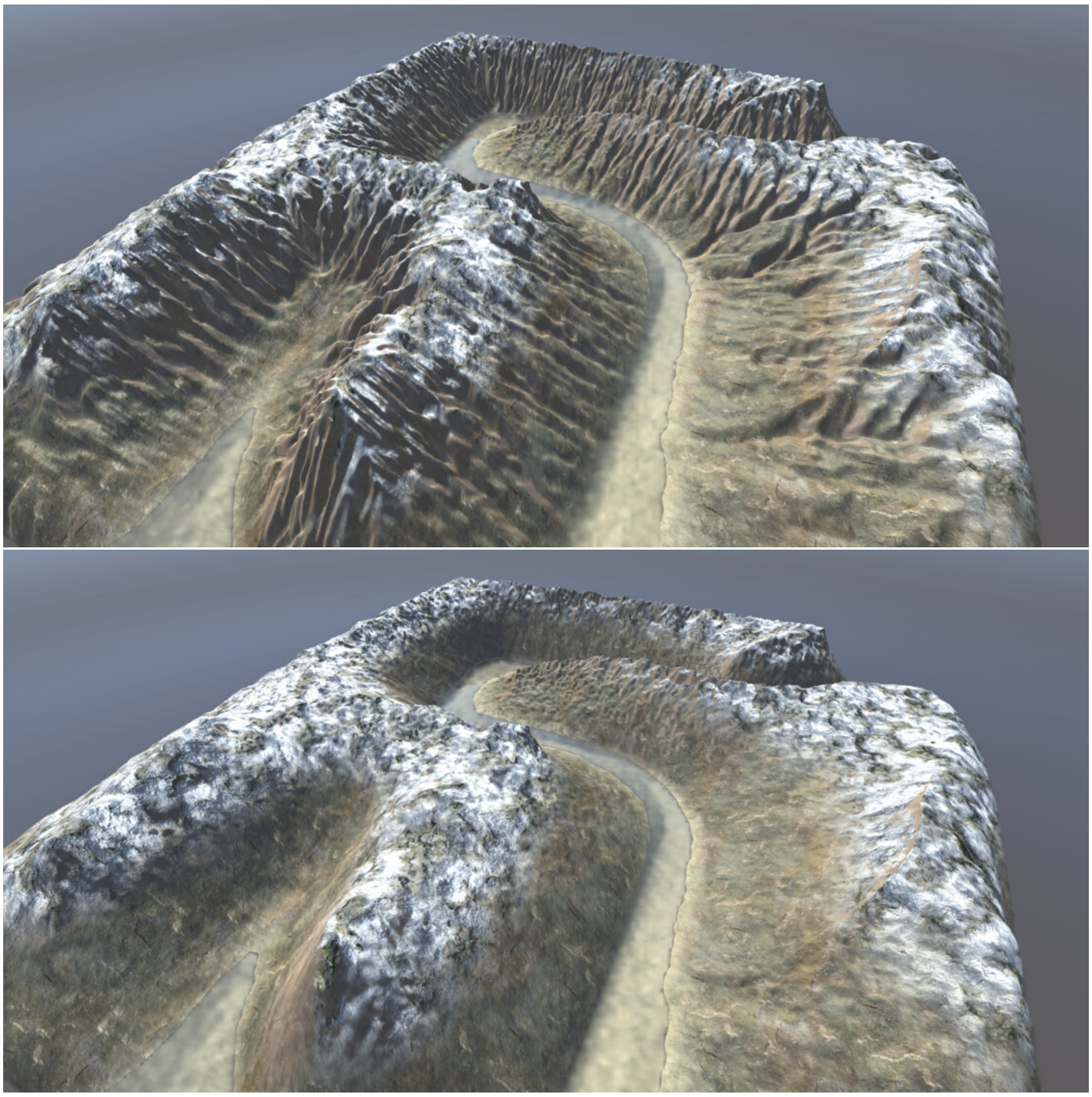
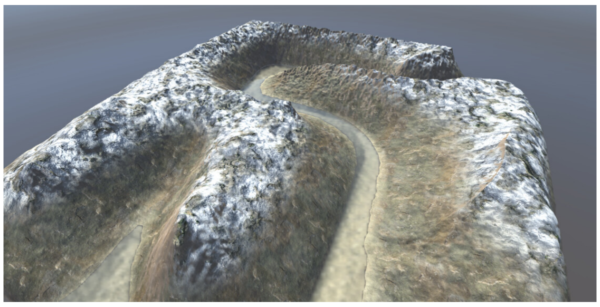
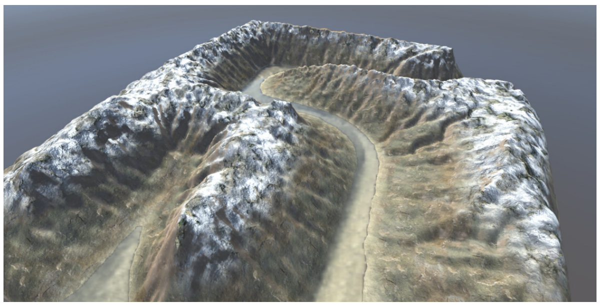
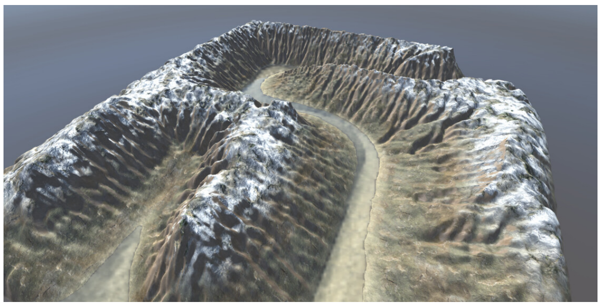


Figure 5.14: Mountain river with and without erosion

In Figure 5.15 we can see that more erosion details are gradually added with each grid level. We simulated 2000 iterations of erosion on grid levels 8 and 9 so this is where the terrain changes the most. Then we ran 1000 iterations on grid level 10 and another 500 iterations on grid level 11 before evaporation. This took almost 35 seconds to simulate.


(a) Final terrain without erosion


(b) Grid level 8


(c) Grid level 9

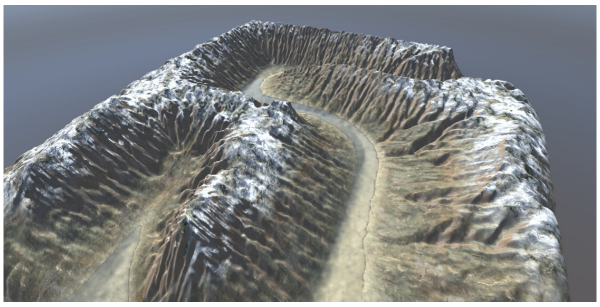

(d) Grid level 10

Figure 5.15: Erosion on lower grid levels of mountain river

Note: Lower resolutions are automatically scaled up by Unity 3D terrain to 2048^2

5.5 Combined terrain

Our combined terrain in Figure 5.16 is designed to have two different types of terrain divided by a river. On one side there is a mountain range and on the other side, there are flat grasslands. This is to showcase control over noise and erosion. We split the grasslands into two parts where half is slightly eroded and has a few splines to make the terrain less smooth. Close to the camera in Figure 5.16 we have the slightly eroded grasslands with some spline details and further from the camera we have the smooth grasslands.

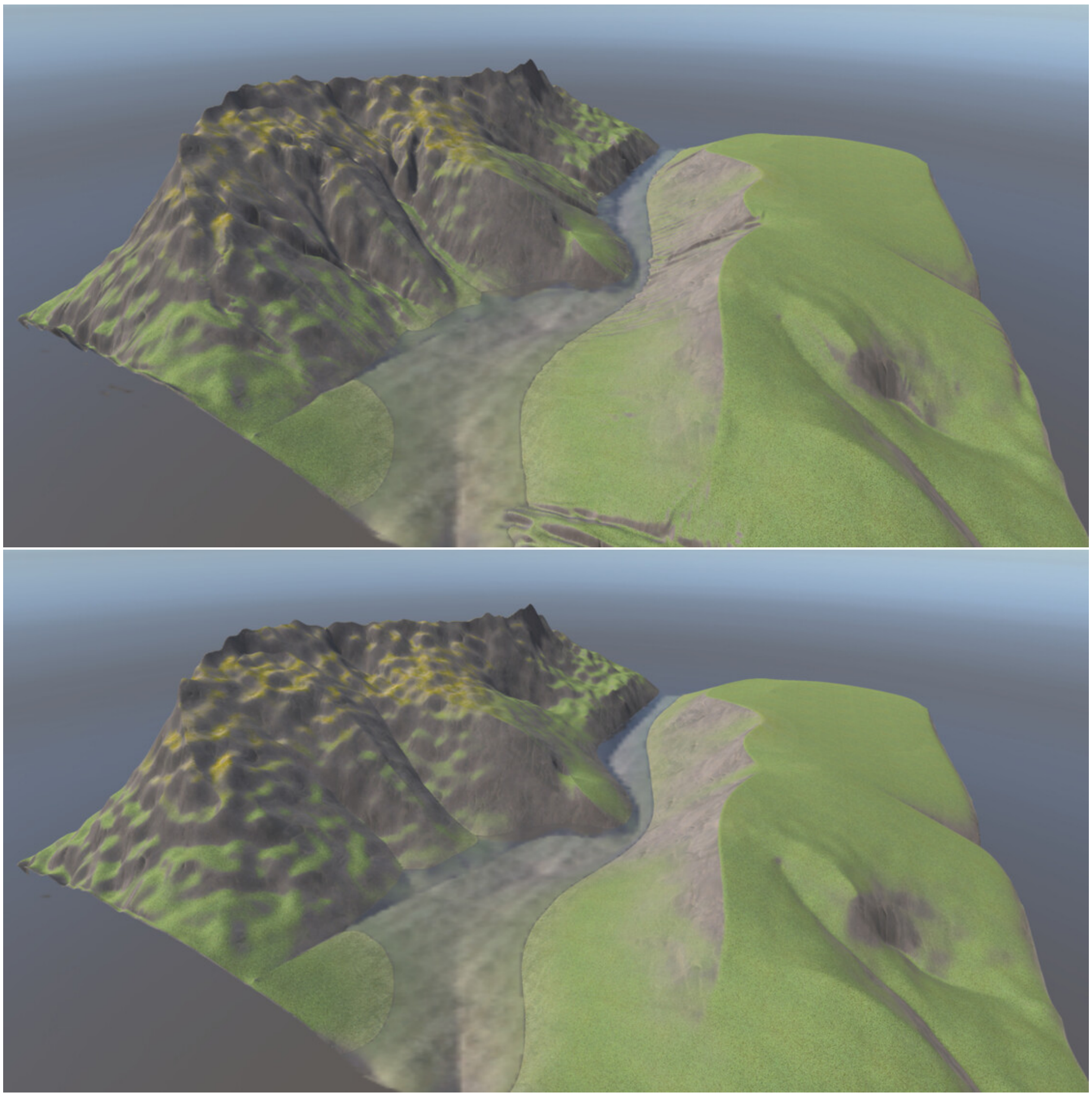


Figure 5.16: Combined terrain with and without erosion

We wanted a very visible deformation from erosion on the mountainside. Our process for achieving this was first placing most of the feature splines. Then we added meta points with erosion parameters and modeled the erosion on the 256x256 grid. Then we adjusted the erosion for bigger grids and added noise in the end. We designed this terrain in 80 minutes, half the time was used placing the feature curves, 25 minutes on erosion, 10 minutes on noise, and 5 minutes on final touches. A screen recording of this process is available as an attachment to this thesis.

This terrain is a good example of how parameter diffusion can be used in terrain generation.

The erosion and noise are mostly limited to the mountains with small lines on the hills close to the camera and no erosion in the river. By using multigrid erosion we could quickly create large-scale erosive features in the terrain before adding finer details. As most of our erosion happens on the smallest grid our diffusion and erosion take 4.6 seconds before evaporating the water which takes 2.8 seconds.

Finally, this terrain was created without limiting the splines to certain grid levels. This results in visible unnatural edges where the feature splines are located. In Figure 5.16 we can trace about half of the splines used to generate this terrain. Some of these create sharp edges or unnaturally smooth curves in the terrain. Figure 5.17 shows the splines used to generate this terrain.

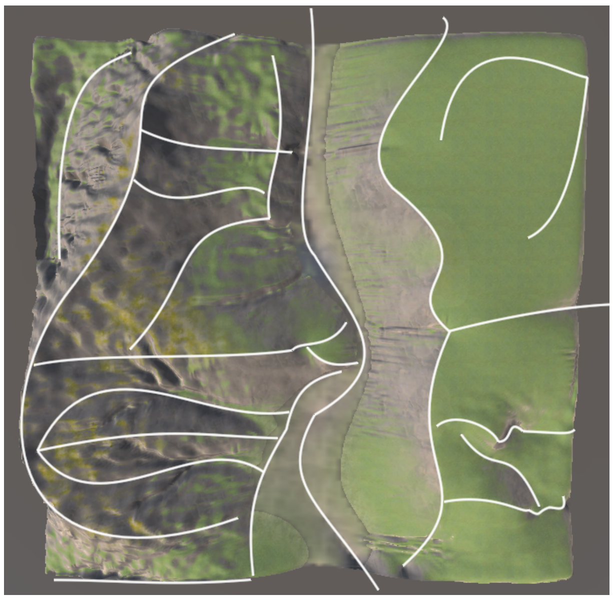


Figure 5.17: Splines used to generate this terrain

5.6 Pathway

This terrain is designed to showcase noise warping and limiting splines to grid levels. We rendered the final terrain with and without warping in Figure 5.18. To showcase the noise we also rendered the smooth diffused terrain and the warped noise separately in Figure 5.19. We used five splines to generate this terrain and none of the splines were defined on all grids. Two of the splines were used from the smallest grid to grid level 9, one was used from the smallest grid to grid level 6, one was used from grid levels 4-9 and the last spline was defined from 0-11. The final resolution is 2048x2048, grid level 11, which means the largest grid had no restrictions and was used only for smoothing. This is why there is almost no trace of the splines in the final result. We used a lot of warped noise in this terrain as well as high amplitude and low noise scale to make the noise very prominent.

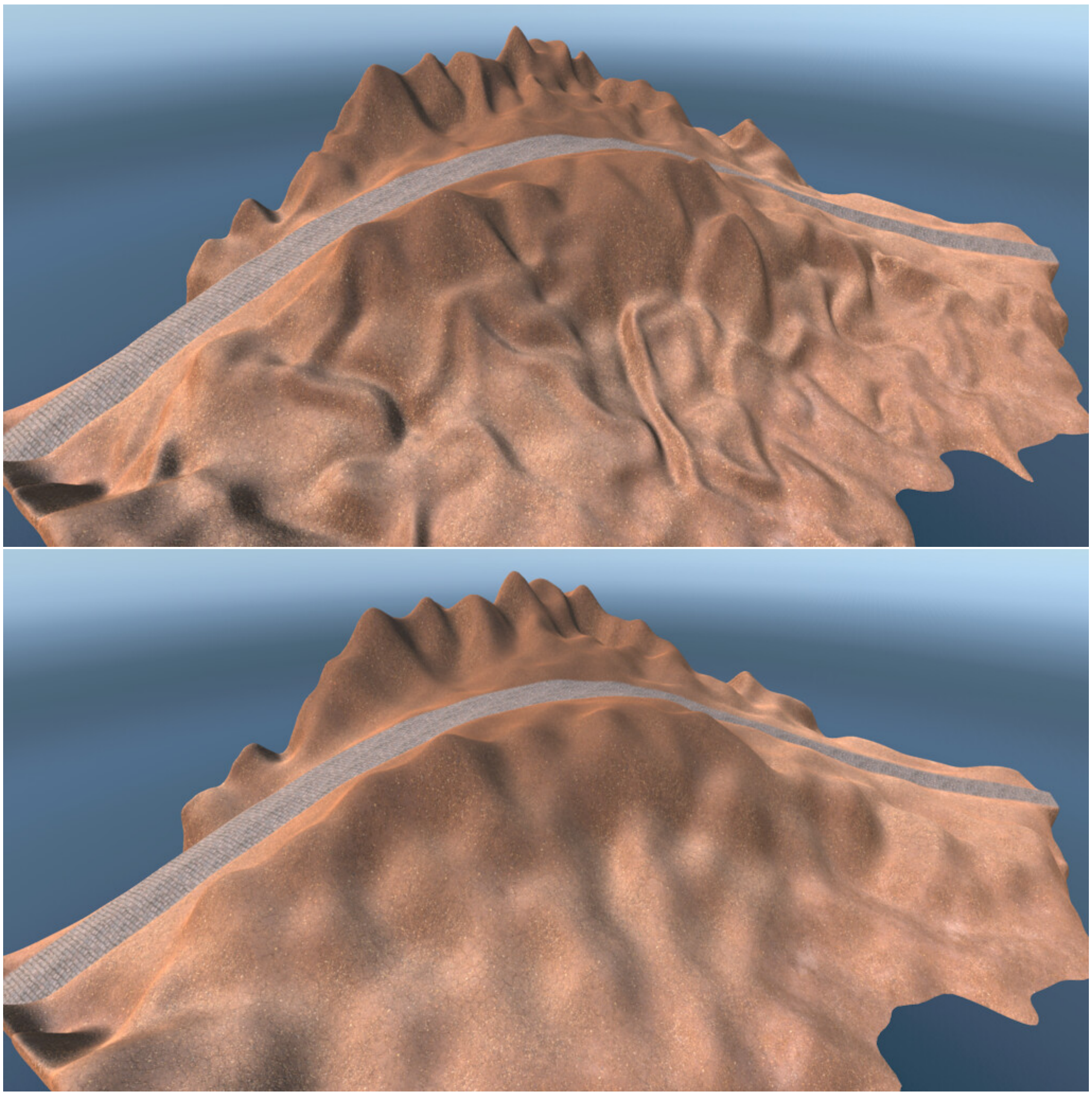
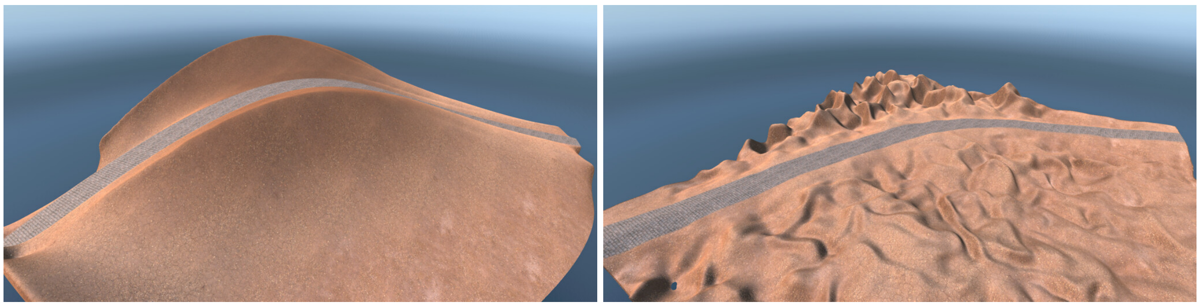


Figure 5.18: Pathway with and without warping



(a) The diffused terrain (b) Only noise

Figure 5.19: Warped pathway diffused terrain and noise separately

5.7 Generation parameters

We provide the parameters used to generate the terrains as well as the generation time as tables below.

Table 5.1 shows the parameters we used for the diffusion and noise when generating each terrain. *Noise amplitude* and *Noise scale* are multiplied with the diffused parameters as mentioned in section 4.4. *Break on level* is used in section 4.3 to calculate how many Jacobi iterations to do on each level in the multigrid solver. *Spline samplings* is described in section 4.2 where it refers to how many samples are taken from each Bézier curve when rasterizing. A higher number gives more smooth curves but takes more time.

Terrain parameters	Fantasy	Mountain range	Mountain river	Combined	Pathway
Noise Amplitude	23.5	30	12	20	62.2
Noise Scale	4	4.5	8.6	2.6	1
Diffusion Iteration Multiplier	2	3	0.89	3	1.5
Spline samplings	300	300	300	300	100

Table 5.1: Terrain Parameters

Table 5.2 shows the parameters we used in our erosion simulation. These parameters are explained in Table 3.2 in section 3.7. We do not include *pathway* in tables 5.2, 5.3 or 5.4 since no erosion was used to generate this terrain.

Erosion parameters	Fantasy	Mountain range	Mountain river	Combined
Pipe Area	1	1	1	1
Gravity	4.36	3.48	6	2
Pipe Length	1	1	1	1
Cell Size	1	1	1	1
Sediment Capacity	0.9	1	1	0.9
Suspension Rate	0.7	0.525	1	0.69
Deposition Rate	0.882	0.812	1	0.46
Evaporation	0.12	0.15	0.15	0.1
Max Rain Intensity	0.4	0.5	0.18	0.36
Min Rain Intensity	0.24	0.276	0.08	0.2
Max Rain Size	32	35	50	29.7
Min Rain Size	20.8	15	16.4	15.4
Time Delta	0.2	0.2	0.25	0.1

Table 5.2: Erosion parameters

We show the number of iterations on each grid level in Table 5.3. This refers to the user-defined amount of iterations mentioned in subsection 4.5.1.

Grid level	Fantasy	Mountain range	Mountain river	Combined
8	0	500	2000	1000
9	547	500	2000	200
10	1000	500	1000	100
11	0	0	500	0

Table 5.3: Erosion iterations

Though we did not focus on optimizing for speed in our implementation we provide the time it took to generate the terrains with and without erosion in the tables 5.4 and 5.5.

Time to create	256	512	1024	2048
Fantasy	0.38s	1.7s	10.7s	13.8s
Mountain range	0.54s	1.7s	6.8s	10.3s
Mountain river	1.8s	5.6s	14.8s	34.7s
Combined	0.78s	1.45s	3.14s	6.7s

Table 5.4: Generation time for different grid sizes with erosion

Time to create	256	512	1024	2048
Fantasy	0.33s	0.4s	0.65s	1.5s
Mountain range	0.14s	0.23s	0.62s	1.9s
Mountain river	0.09s	0.15s	0.33s	0.93s
Combined	0.17s	0.30s	0.67s	1.9s
Pathway	0.05s	0.12s	0.35s	0.94s

Table 5.5: Generation time for different grid sizes without erosion

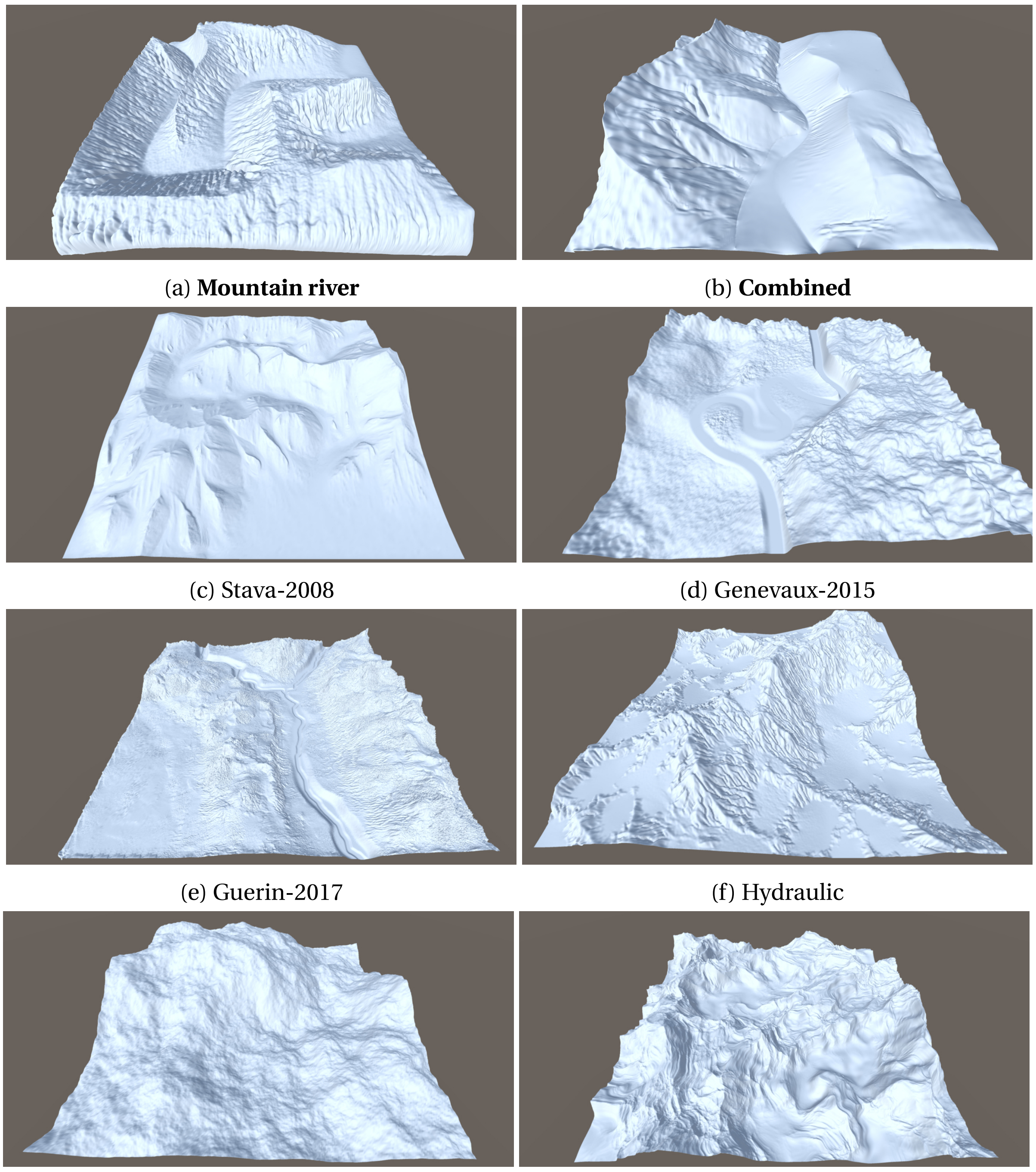
The experiments were run on a computer with AMD Ryzen 5 3600X and Nvidia GeForce RTX 2060 Super. We ran each experiment a few times and took the average time from rasterization to creating the finished texture as visualized in Figure 3.1. This time was measured after reading back one pixel to the CPU of the finished texture. This ensures the GPU has finished all the work and minimizes the cost of reading back the texture to the CPU as this is a slow operation that is not needed for rendering.

5.8 Comparison

In this section we compare two of our terrains to the results of other papers. All terrains compared in this section are provided by Galin *et al.* [7] and rendered by us shown in Figure 5.20. Table 5.6 gives some context to the terrains rendered in Figure 5.20. Our terrains *(a) Mountain river* and *(b) Combined* are marked with bold text.

Terrain Name	Author	Method
(a) **Mountain river** (b) **Combined**	This thesis	Based on feature-based generation and hydraulic erosion
(c) Stava-2008	Šťava *et al.* [34]	Based on hydraulic erosion with multiple layers of material
(d) Genevaux-2015	Génevaux *et al.* [19]	Based on a construction trees which is a type of feature-based generation
(e) Guerin-2017	Guérin *et al.* [20]	Machine learning
(f) Hydraulic	Musgrave *et al.* [27]	Hydraulic erosion
(g) Simplex (h) Warped-noise	Galin *et al.* [7]	Based on hydraulic erosion with multiple layers of material

Table 5.6: Short description of terrains in 5.20



(a) **Mountain river** (b) **Combined**

(c) Stava-2008 (d) Genevaux-2015

(e) Guerin-2017 (f) Hydraulic

(g) Simplex (h) Warped-noise

Figure 5.20: Renderings of terrains made with different methods

Note: The intended height of the models was not provided so we chose something we thought was appropriate for each terrain

We can see that *(a) Mountain river* has the distinct hydraulic erosion features that *(f) Hydraulic* has. *(d) Guerin-2017* has different levels of noise on either side of the river which is quite flat. This is comparable to our *(a) Mountain river* except you can not see the primitive features as well. The erosion on both methods is based on hydraulic erosion, *(c) Stava-2008* and *(f) Hydraulic*, have more even erosion than we were able to create. This is probably caused by different erosion methods. *(d) Guerin-2017* is very detailed and even has small erosive details on the mountainside.

(b) Combined has both large scale and small scale features from erosion, small scale similar to *(f) Hydraulic* and large scale more similar to *(c) Stava-2008*. Though *(b) Combined* has very repetitive noise on the mountain side, *(a) Mountain river* has less repetitive noise due to the low scale erosion.

Chapter 6

Discussion

In this chapter, we will discuss our results presented in chapter 5. We compare the results to previous papers, discuss our four novelties and finally discuss our two research questions.

6.1 Multigrid diffusion

Multigrid diffusion is solving a Poisson equation on multiple grids. It is previously used in several computer graphics papers [1], [14], [22]. We improved the multigrid solver by changing our Poisson equation on each grid level. This lets us decide which grid levels each spline should be included in. Our multigrid solver enables us to deconstruct splines as Figure 3.8 shows. The cost of the possibility to deconstruct splines is the time it takes to rasterize our data onto new grids compared to restricting the grids.

We restricted most of our splines in three of the five results. Fantasy 5.2, mountain river 5.4, and pathway 5.6 restricted most of the splines so they were not included in the largest grid. This removes the sharp edges which the splines create on the largest grid. In our other two results, mountain range 5.3 and combined 5.5, many of the splines are visible which in many cases are unwanted and could even create artifacts. This problem can be seen in the previous work by Hnaidi *et al.* [1] and their results shown in Figure 2.4. Another method for solving this is by smoothing the final terrain. Though this can remove some of the artifacts created by the splines it also removes some of the features of the terrain. Our methods let the user remove the artifacts only.

Fantasy terrain 5.2 utilized the multigrid diffusion both for creating a smooth mountain range and a heart with a single spline each. As we mentioned in chapter 5 this would have been very difficult to do with more splines or gradients. Gradients are only suitable for forming the terrain in the immediate area around a spline or when the splines are far from each other. The gradients are dependent on the normal vectors of the splines and they vanish when there are more splines in the terrain. This makes the splines with gradients dependent on the splines close by so adding a new spline could change the behavior of the gradients in unexpected ways. For this reason we found the gradients difficult to work with and used them sparsely. We can avoid using gradients to create hills and mountains with our multigrid diffusion which is another advantage of our multigrid diffusion.

6.2 Multigrid erosion

Multigrid erosion is a new concept we introduced in chapter 3. This concept can be described as simulating erosion on a small grid, interpolating the state to a larger grid, and continue to simulate erosion with higher resolution. This gives results similar to simulating erosion on different scales as done by Argudo *et al.* [18]. Our method runs the simulation on smaller grids which uses less time compared to scaling down the simulation with parameters, though it might be less accurate. By comparing our results, especially mountain river, to hydraulic erosion by Mei *et al.* [26] we can see similar erosive features. We included one of their results in Figure 2.6. Our terrain also has similar erosive features as the hydraulic terrain in Figure 5.20. Our method does not have as much deposited sediment as the hydraulic terrain and we discuss the reason for this in section 6.3. We could see in Figure 5.15 how this allows us to create erosive features of different sizes and combine them to create heavily eroded terrains. This is an efficient method for simulating erosion on multiple scales.

6.3 Constrained erosion

Constrained erosion is limiting the erosion in some way. Argudo *et al.* [18] used this term when they counteracted erosion by applying procedural uplift in certain parts of a terrain. We use this term to describe the effect of our diffused erosion parameters from section 3.7. Lower values of s_i reduce the sediment capacity and constrain the erosion to only deposit sediment. Lower w_i values reduce the amount of water which affects how far the water flows before evaporating. When creating our results we depended on these values to limit erosion in certain areas. We also used it in the fantasy 5.2 and combined 5.5 terrain to not erode certain areas. This method is based on the same method as the noise is from Hnaidi *et al.* [1] which makes the constrained erosion and noise easy to combine.

The constrained erosion also solves an important problem in our mountain range terrain 5.3. This terrain has some sharp edges and areas where a lot of water flows in from multiple directions. This is a problem with the method proposed by Mei *et al.* [26] which is the basis of our method. The problem with this method has to do with sediment transportation. Though Mei *et al.* [26] claim the sediment transportation is unconditionally stable this is not true for all states. There are two possible states where sediment is lost because of what they describe as "...taking an Euler step backward in time". The first state happens initially when water is added for the first time. When a cell has an inflow of water but not yet an outflow the sediment suspended is partly lost. The second state happens when a cell has an inflow from multiple cells at the same time. If a cell has an equal inflow from two opposite directions the velocity of the water is zero. In this scenario, this cell does not move any sediment as it has no velocity while the neighboring cells can have a velocity and overwrite the sediment that was supposed to be moved to the original cell. This second scenario affected our mountain range terrain a lot and quickly eroded deep unnatural holes in the terrain. To compensate for this we lowered the sediment capacity in these areas with our constrained erosion which resulted in reduced sediment loss. There were also parts of the terrain that we did not want as much eroded, so we lowered the amount of water added with w_i. This illustrates the flexibility of our approach and the ability to adjust the input locally to remove artifacts.

6.4 Diffused warped noise

Warped noise is a noise distorted by some other noise and is used in terrain generation. We combined simple warped noise from Carpentier *et al.* [14] with diffused noise from Hnaidi *et al.* [1] to create our diffused warped noise. The warp effect we implemented might be enhanced. We based our pathway 5.6 on warped noise and we used noise warping on a part of our fantasy terrain 5.2. The warped noise looks unnatural and it is as repetitive as normal noise on a lower scale. It did make the warped pathway more supernatural and made the noise on the island in fantasy terrain 5.2 sharper. However, it did not have remotely the same impact as erosion had on the results. Carpentier *et al.* [14] used a more sophisticated warped noise combined with directional vectors to create what they called erosive noise. For further research one could use the diffused normal vector scalar field as directional vectors to warp the noise in a direction relative to the splines. We have shown that warping does not create artifacts in the terrain so it should be possible to generate terrain with similar warping as Carpentier *et al.* created. As Carpentier *et al.* [14] stated "...defining such a function is probably as much an art as it is a science" so we leave it for artists to explore these possibilities.

6.5 The scale of terrain features

Our first research question was "How can one design a consistent terrain with a combination of fine-scale and large-scale features?". With this method, we introduced two ways to generate such terrain. Our multigrid diffusion lets the user decide how much of the surrounding area each spline should affect. By affecting a minor area one can design fine-scale features such as a heart inside of a volcano as shown in section 5.2. The user can also create a mountain range with a single spline by defining it only on lower grid levels as shown in section 5.2. By combining these features, the user can start a process by creating the large features of the terrain, and then incrementally adding finer details. Our second method is multigrid erosion. The multigrid erosion can erode the terrain at different scales creating features of both large and fine-scale as shown in Figure 5.15.

6.6 Noise repetitiveness

Our second research question was "How can one mitigate the repetitiveness of the noise added in feature-based terrain created with diffusion?". We tried to mitigate this repetitiveness by adding erosion and warping the noise. Our results show that eroding the terrain can remove most of the repetitiveness of the noise. Erosion simulation is computationally expensive and could hinder the modeling process. With our method, one can erode at lower resolutions first to see a rough version of the terrain, created in half the time or even less. This can be considered as a draft of the final terrain and includes enough information to improve the model on a higher grid level. One can also attempt to mitigate the repetitiveness by warping the noise. Our results show that it is not easy to do so with our diffused warped noise algorithm 3.11. It does however show that warping can be added to the noise 5.2,5.6 and a further developed warping method could in the future be a comparable and significantly faster method for mitigating the repetitiveness of the noise than erosion simulations.

Chapter 7

Conclusions and future work

7.1 Conclusion

In this thesis, we have presented four novelties to current methods for generating terrain to answer our two research questions. Our results show examples of terrains generated with this new method by inexperienced terrain designers. Our multigrid diffusion gives the designer increased control over the process which proved very useful in the design process. It also improves the results by smoothing over certain splines so that they are not pronounced. Further, we improved hydraulic erosion from a previous paper by applying erosion at multiple grids to create erosive features of different scales. To better control the erosion we let the user control where water should be added as well as the hardness of the terrain. Finally, we modified our noise function by warping the noise. Our paper has shown how erosion, diffused warped noise, and feature-based terrain generation can be used together to generate terrain in an innovative way. Erosion and diffusion work well on all sizes with different levels of detail while our diffused warped noise did not prove as effective. Though our diffused warped noise, Equation 3.11, did not clearly improve on the repetitiveness, we discovered ways to modify the noise function to create more advanced noise which could do this at a low computational cost. Our novelties pave ways for adding features of different scales to a virtual terrain as well as a way to reduce the repetitiveness of the noise.

7.2 Future work

This method would be more feasible to use for terrain modeling if one could create larger terrains by seamlessly connecting multiple grids. Creating terrains with larger resolutions than 4086x4086 takes a very long time, uses a lot of memory, and is currently not supported by Unity 3D terrain. To create larger terrains one can connect multiple terrains in a grid. Our method does not focus on making it easy to create two terrains that are adjacent to each other. This is possible by placing a spline on the edge of both terrains to force the edge on both terrains to be of equal height. An improved method could automate this by enforcing the edge of the second terrain to fit the edge of the first terrain. There might be even better methods to combine such terrains and this is worth exploring further. Generating larger terrains is also a challenge when using erosion. The erosion currently has strict border boundaries so no water can escape the grid. Without any water flow between the grids, the erosion could be unnatural and create artifacts in the terrains where they meet.

As erosion proved useful at mitigating repetitiveness we believe a state-of-the-art hydraulic erosion or multi-layer erosion method could improve our results. Such methods could create new and more realistic erosive features in the terrain. As performance was not in our scope one could improve the performance of our method by optimizing our erosion, diffusion, noise generation, and rasterizing.

Finally, we encourage anyone to experiment with different types of noise to generate terrains with our method. There are a lot of variables that could be changed when adding noise. It is possible to change some of the variables to generate a wide variety of noise. Some of these variables are the number of noise layers, the scale between each layer, the type of noise, and the warping method. It is possible to introduce variables from diffused textures into the noise function. This could result in very realistic or very surreal terrains and give each terrain a unique appearance.

Bibliography

[1] H. Hnaidi, E. Guérin, S. Akkouche, A. Peytavie, and E. Galin, “Feature based terrain generation using diffusion equation,” *Computer Graphics Forum*, 7th ser., vol. 29, pp. 2179–2186, Sep. 2010. DOI: 10.1111/j.1467-8659.2010.01806.x. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01381590.

[2] I. Joshua Yehl. (2019). How jon favreau directed the lion king inside a video game, [Online]. Available: https://www.ign.com/articles/2019/05/30/how-jon-favreau-directed-the-lion-king-inside-a-video-game (visited on 05/28/2021).

[3] W. W. (Brandon Bestenheider and D. D. (Brad Baruh, *Disney Gallery: The Mandalorian - Technology*, 2020. [Online]. Available: https://www.imdb.com/title/tt12258086.

[4] Blizzard Entertainment, *World of warcraft*, 2004.

[5] Hello Games, *No man's sky*, 2016.

[6] W. Chris Higgins. (2014). No man's sky would take 5 billion years to explore, [Online]. Available: https://www.wired.co.uk/article/no-mans-sky-planets (visited on 05/28/2021).

[7] E. Galin, E. Guérin, A. Peytavie, G. Cordonnier, M.-P. Cani, B. Benes, and J. Gain, “A review of digital terrain modeling,” *Computer Graphics Forum*, vol. 38, no. 2, pp. 553–577, 2019. DOI: https://doi.org/10.1111/cgf.13657. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13657. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13657.

[8] Unity Technologies. (2021). The leading platform for creating interactive, real-time content, [Online]. Available: https://unity.com/ (visited on 05/28/2021).

[9] K. Perlin, “Improving noise,” *ACM Trans. Graph.*, vol. 21, no. 3, pp. 681–682, Jul. 2002, ISSN: 0730-0301. DOI: 10.1145/566654.566636. [Online]. Available: https://doi.org/10.1145/566654.566636.

[10] B. Chan and M. Mccool, “Worley cellular textures in sh,” p. 18, Jan. 2004. DOI: 10.1145/1186415.1186437.

[11] A. Lagae, S. Lefebvre, G. Drettakis, and P. Dutré, “Procedural noise using sparse gabor convolution,” *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)*, vol. 28, no. 3, pp. 54–64, Jul. 2009. DOI: 10.1145/1531326.1531360.

[12] K. Perlin, “An image synthesizer,” in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’85, New York, NY, USA: Association for Computing Machinery, 1985, pp. 287–296, ISBN: 0897911660. DOI: 10.1145/325334.325247. [Online]. Available: https://doi.org/10.1145/325334.325247.

[13] K. Perlin and E. M. Hoffert, “Hypertexture,” in *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’89, New York, NY, USA: Association for Computing Machinery, 1989, pp. 253–262, ISBN: 0897913124. DOI: 10.1145/74333.74359. [Online]. Available: https://doi.org/10.1145/74333.74359.

[14] G. J. P. de Carpentier and R. Bidarra, “Interactive gpu-based procedural heightfield brushes,” in *Proceedings of the 4th International Conference on Foundations of Digital Games*, ser. FDG ’09, Orlando, Florida: Association for Computing Machinery, 2009, pp. 55–62, ISBN: 9781605584379. DOI: 10.1145/1536513.1536532. [Online]. Available: https://doi.org/10.1145/1536513.1536532.

[15] I. Parberry, “Modeling real-world terrain with exponentially distributed noise,” 2015.

[16] T. Hyttinen, E. Mäkinen, and T. Poranen, “Terrain synthesis using noise by examples,” in *Proceedings of the 21st International Academic Mindtrek Conference*, ser. AcademicMindtrek ’17, Tampere, Finland: Association for Computing Machinery, 2017, pp. 17–25, ISBN: 9781450354264. DOI: 10.1145/3131085.3131099. [Online]. Available: https://doi.org/10.1145/3131085.3131099.

[17] K. Golubev, A. Zagarskikh, and A. Karsakov, "Dijkstra-based terrain generation using advanced weight functions," *Procedia Computer Science*, vol. 101, pp. 152–160, Dec. 2016. DOI: 10.1016/j.procs.2016.11.019.

[18] O. Argudo, E. Galin, A. Peytavie, A. Paris, J. Gain, and E. Guérin, "Orometry-based terrain analysis and synthesis," *ACM Trans. Graph.*, vol. 38, no. 6, Nov. 2019, ISSN: 0730-0301. DOI: 10.1145/3355089.3356535. [Online]. Available: https://doi.org/10.1145/3355089.3356535.

[19] J.-D. Génevaux, É. Galin, E. Guérin, A. Peytavie, and B. Benes, "Terrain generation using procedural models based on hydrology," *ACM Trans. Graph.*, vol. 32, no. 4, Jul. 2013, ISSN: 0730-0301. DOI: 10.1145/2461912.2461996. [Online]. Available: https://doi.org/10.1145/2461912.2461996.

[20] É. Guérin, J. Digne, É. Galin, A. Peytavie, C. Wolf, B. Benes, and B. Martinez, "Interactive example-based terrain authoring with conditional generative adversarial networks," *ACM Trans. Graph.*, vol. 36, no. 6, Nov. 2017, ISSN: 0730-0301. DOI: 10.1145/3130800.3130804. [Online]. Available: https://doi.org/10.1145/3130800.3130804.

[21] M. Becher, M. Krone, G. Reina, and T. Ertl, "Feature-based volumetric terrain generation," in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '17, San Francisco, California: Association for Computing Machinery, 2017, ISBN: 9781450348867. DOI: 10.1145/3023368.3023383. [Online]. Available: https://doi.org/10.1145/3023368.3023383.

[22] A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin, "Diffusion curves: A vector representation for smooth-shaded images," in *ACM SIGGRAPH 2008 Papers*, ser. SIGGRAPH '08, Los Angeles, California: Association for Computing Machinery, 2008, ISBN: 9781450301121. DOI: 10.1145/1399504.1360691. [Online]. Available: https://doi.org/10.1145/1399504.1360691.

[23] J. McCann and N. S. Pollard, "Real-time gradient-domain painting," in *ACM SIGGRAPH 2008 Papers*, ser. SIGGRAPH '08, Los Angeles, California: Association for Computing Machinery, 2008, ISBN: 9781450301121. DOI: 10.1145/1399504.1360692. [Online]. Available: https://doi.org/10.1145/1399504.1360692.

[24] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Second, ser. Other Titles in Applied Mathematics. SIAM, 2003, ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003. [Online]. Available: http://www-users.cs.umn.edu/%5C~%7B%7Dsaad/IterMethBook%5C_2ndEd.pdf.

[25] W. Briggs, V. Henson, and S. McCormick, *A Multigrid Tutorial, 2nd Edition*. Jan. 2000, ISBN: 978-0-89871-462-3.

[26] X. Mei, P. Decaudin, and B. Hu, "Fast hydraulic erosion simulation and visualization on gpu," in *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, 2007, pp. 47–56. DOI: 10.1109/PG.2007.15.

[27] 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*, ser. SIGGRAPH '89, New York, NY, USA: Association for Computing Machinery, 1989, pp. 41–50, ISBN: 0897913124. DOI: 10.1145/74333.74337. [Online]. Available: https://doi.org/10.1145/74333.74337.

[28] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys, "A multigrid solver for boundary value problems using programmable graphics hardware," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05, Los Angeles, California: Association for Computing Machinery, 2005, 193–es, ISBN: 9781450378338. DOI: 10.1145/1198555.1198784. [Online]. Available: https://doi.org/10.1145/1198555.1198784.

[29] Microsoft Corporation. (2018). Direct3d architecture (direct3d 9), [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d9/direct3d-architecture (visited on 05/25/2021).

[30] Microsoft Corporation. (2018). Graphics pipeline, [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline (visited on 05/25/2021).

[31] Microsoft Corporation. (2018). Compute shader overview, [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader (visited on 05/25/2021).

[32] N. Ashu Rege. (2008). An introduction to modern gpu architecture, [Online]. Available: http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf (visited on 05/25/2021).

[33] F. Sans and R. Carmona, “A comparison between gpu-based volume ray casting implementations: Fragment shader, compute shader, opencl, and cuda,” *CLEI Electron. J.*, vol. 20, 2017.

[34] O. Šťava, B. Beneš, M. Brisbin, and J. Křivánek, “Interactive terrain modeling using hydraulic erosion,” in *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA ’08, Dublin, Ireland: Eurographics Association, 2008, pp. 201–210, ISBN: 9783905674101.

Appendix A

A.1 Additional results

We rendered more images of the terrain than it was useful to showcase in the results. Here are some of them:

A.1.1 Mountain river

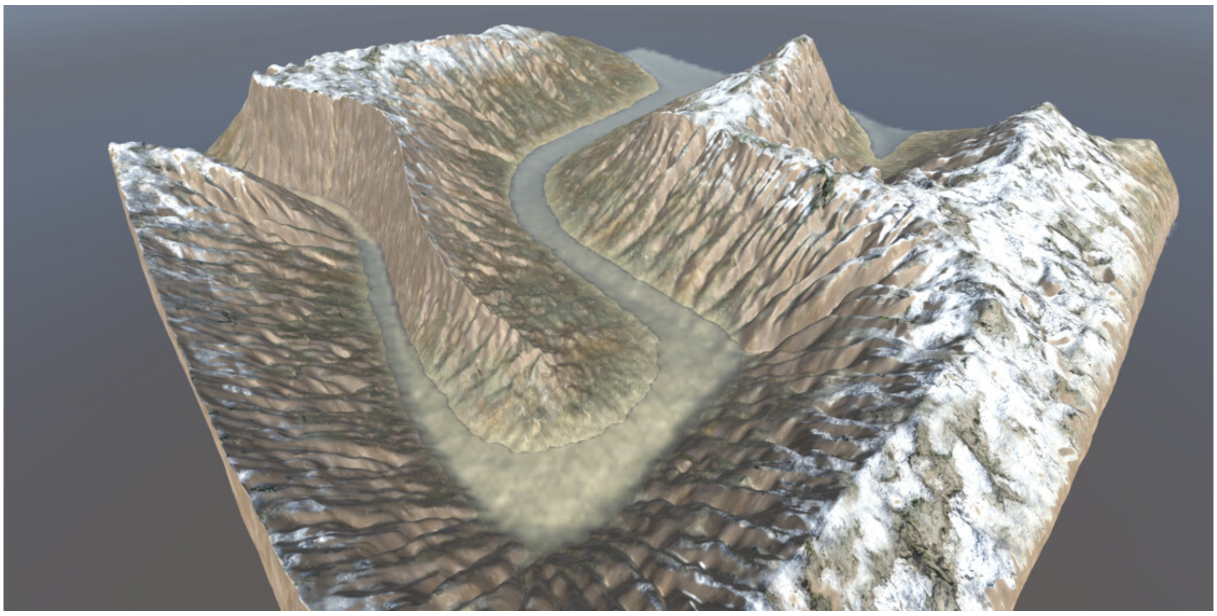


Figure A.1: Mountain river camera 2

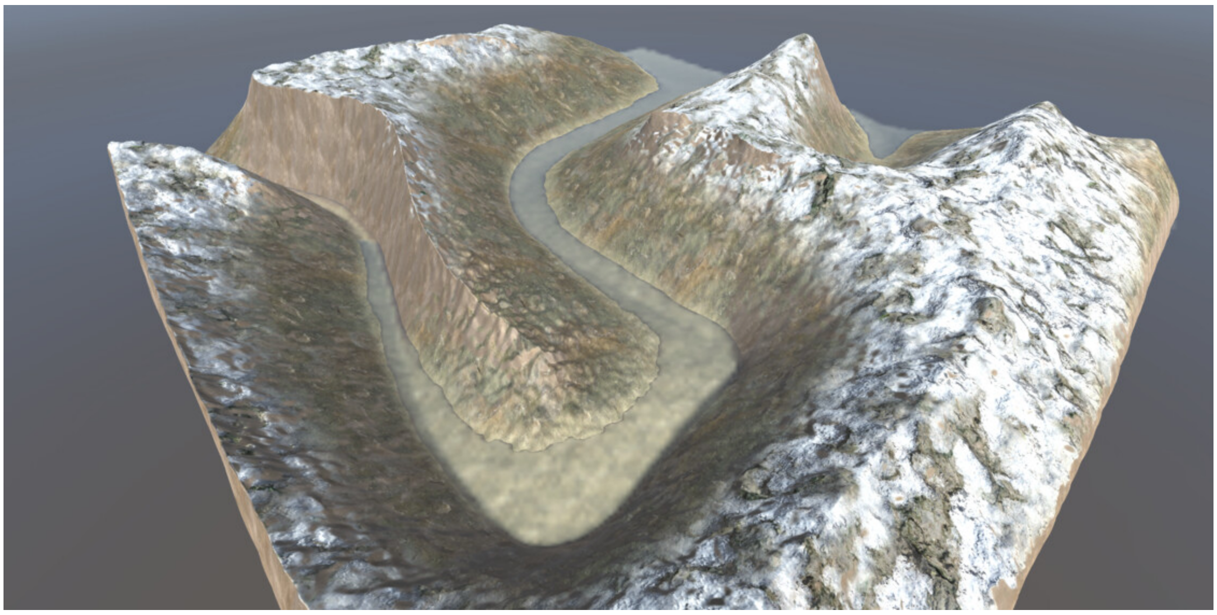


Figure A.2: Mountain river camera 2 without erosion

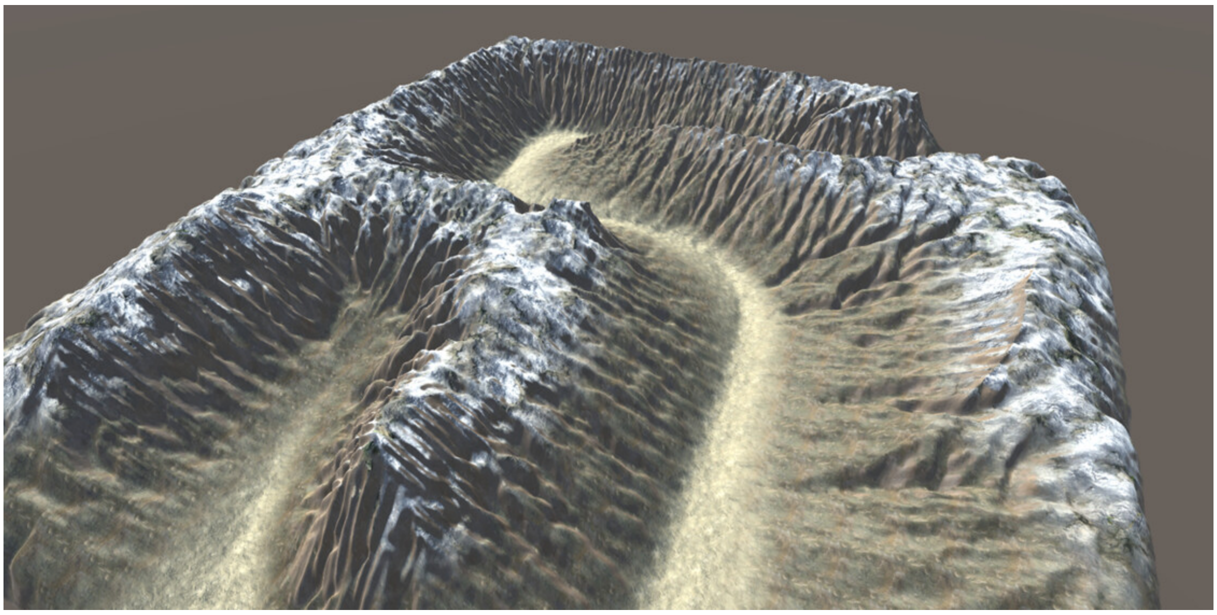


Figure A.3: Mountain river without water

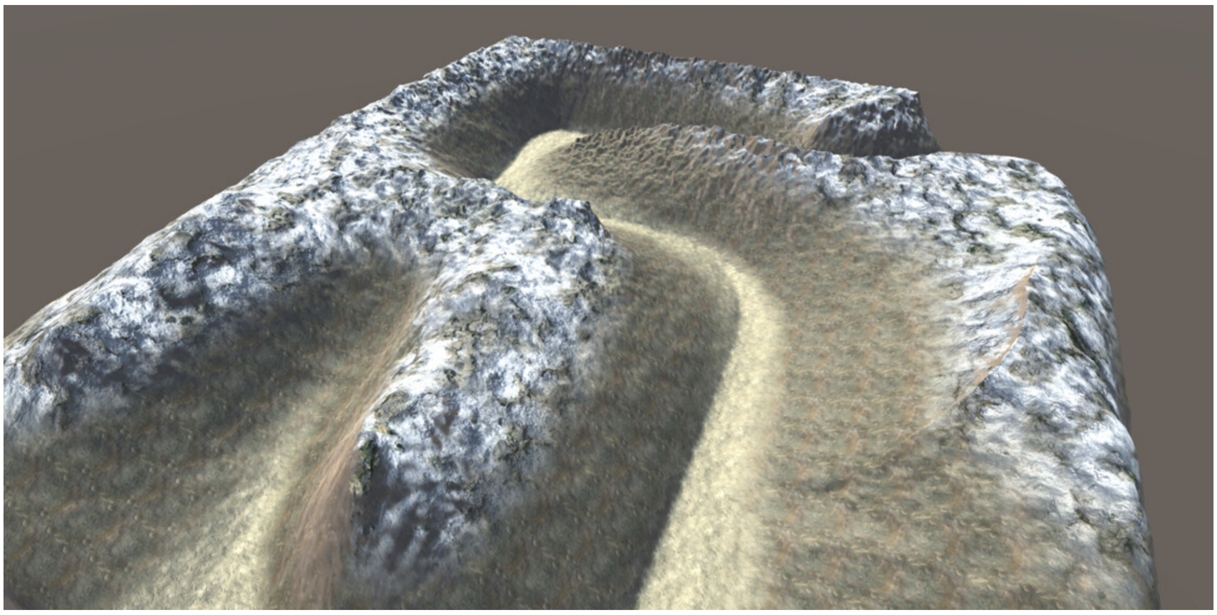


Figure A.4: Mountain river without water or erosion

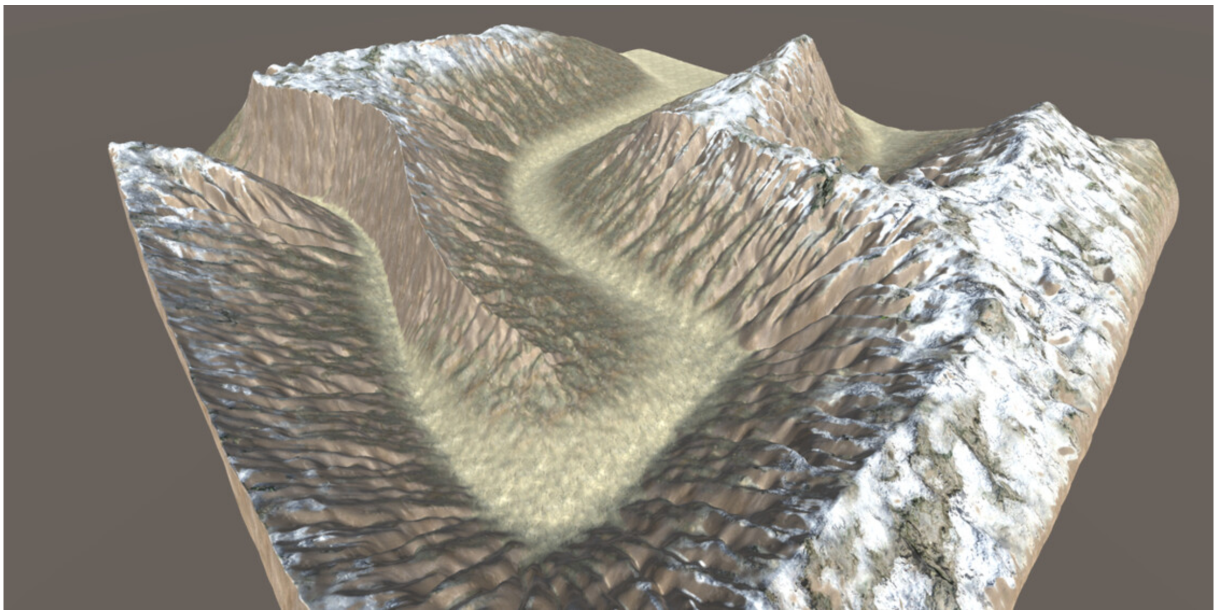


Figure A.5: Mountain river camera 2 without water

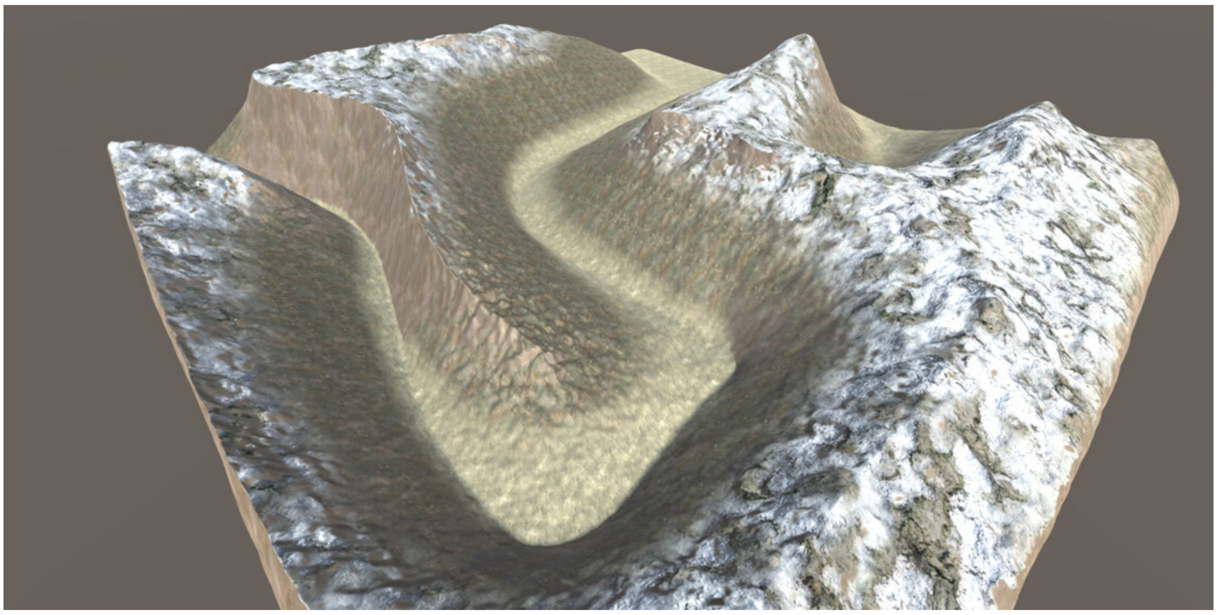


Figure A.6: Mountain river camera 2 without water or erosion

A.1.2 Combined terrain

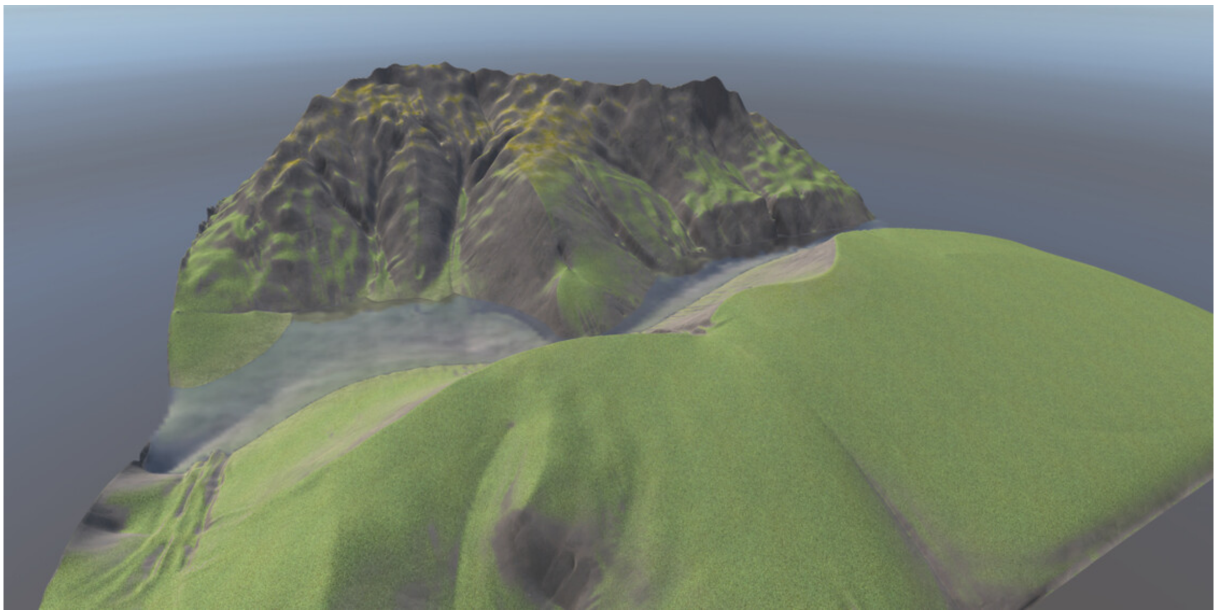


Figure A.7: Combined terrain camera 2

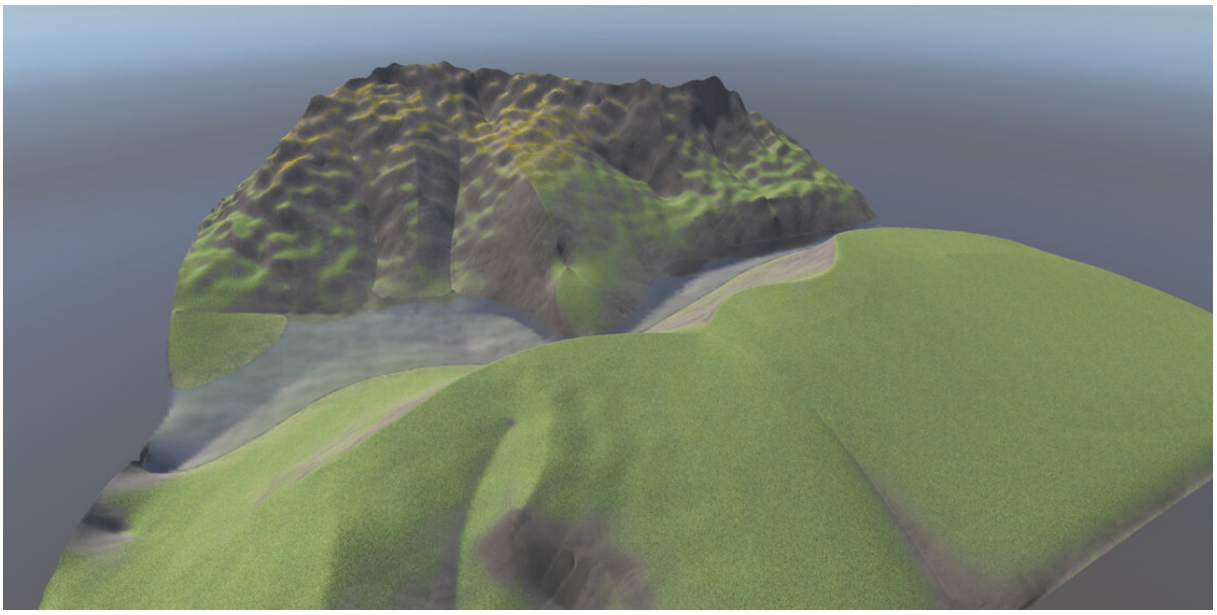


Figure A.8: Combined terrain camera 2 no erosion

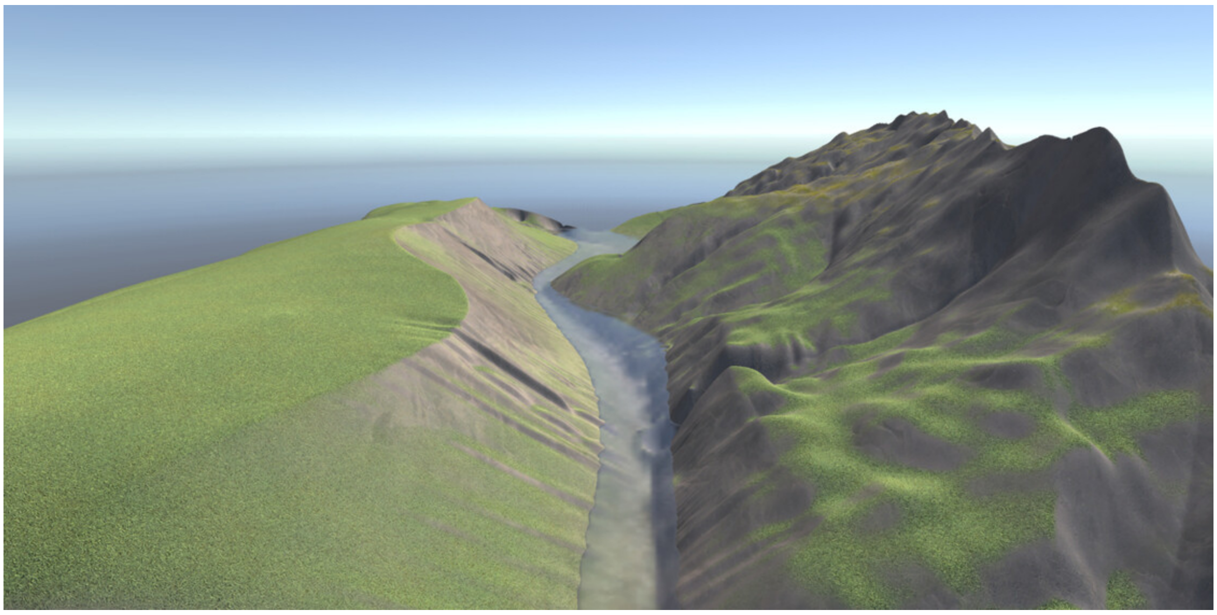


Figure A.9: Combined terrain camera 3

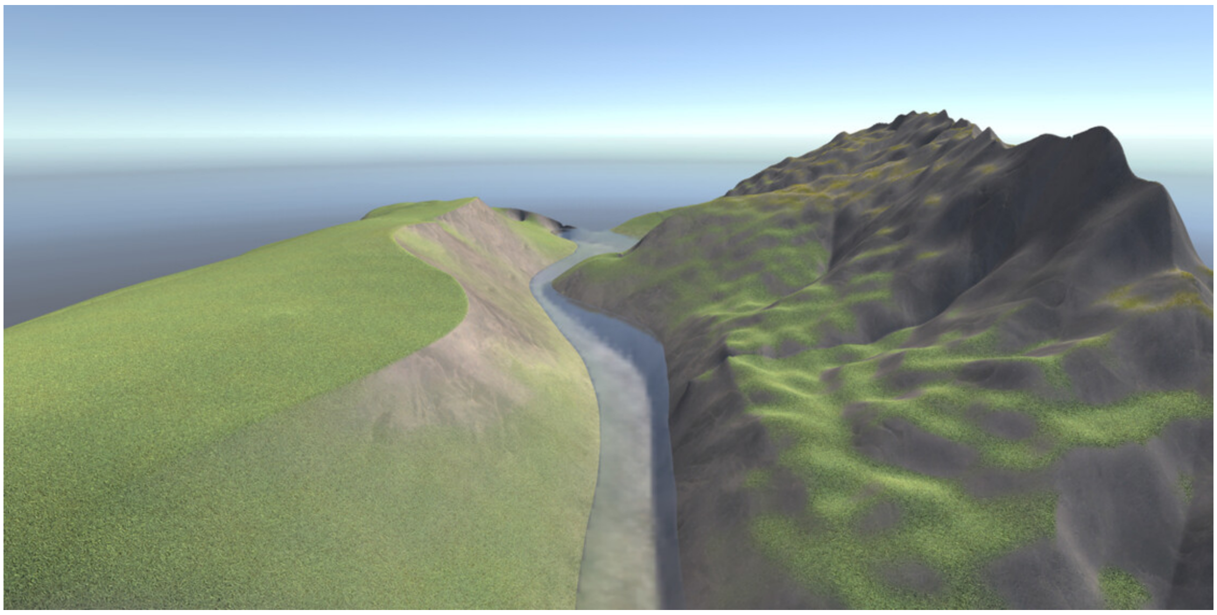


Figure A.10: Combined terrain camera 3 no erosion

A.2 Different kernels

One can use many different kernels for the Jacobi relaxation. We used the nine-point Laplace kernel in Equation 3.4 for our method, but the following kernels would also work with a minor modification to the method. The first equation below was used by Hnaidi, Guérin, Akkouche, *et al.* [1]. To use one of the three first kernels listed below the Jacobi relaxation terms needs to be redefined to Equation A.5.

$$A = \frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{A.1}$$

$$A = \frac{1}{100}\begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 4 & 8 & 16 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix} \tag{A.2}$$

$$A = \frac{1}{84}\begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 4 & 8 & 0 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix} \tag{A.3}$$

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{A.4}$$

$$v_{i+1} = \alpha A v_i + \beta B v_i + \beta G \tag{A.5}$$

A.3 Data optimization

As our implementation uses ARGB textures there is a lot of room for optimizations. Firstly one could remove all unused channels reducing the number of channels from 36 to 26 channels. Then one could move all masks into one channel using bit operations reducing the number of channels needed further down to 24. One could impose a local restriction to the guided diffusion $\alpha = 1 - \beta$ if $\alpha + \beta \neq 0$ where the *if* part could be stored as a mask reducing the number of channels needed to 23. This is one channel less than six full textures.