Large Scale Generation of Voxelized Terrain

Examensarbete utfört i Informationskodning vid Tekniska högskolan vid Linköpings universitet av

Pierre LeMoine

LiTH-ISY-EX--13/4730--SE

Linköping 2013
Large Scale Generation of Voxelized Terrain

Examensarbete utfört i Informationskodning
vid Tekniska högskolan vid Linköpings universitet
av

Pierre LeMoine

LiTH-ISY-EX--13/4730--SE

Handledare: Jens Ogniewski
ISY, Linköpings universitet

Examinator: Ingemar Ragnemalm
ISY, Linköpings universitet

Linköping, 23 september 2013
Abstract

Computer-aided generation of virtual worlds is vital in modern content production. To manually create all the details which today's computers can visualize would be too daunting a task for any number of artists. Procedural algorithms can quickly generate content, but the content suffers from being repetitive. Simulation of geological processes produce good results but require a lot of resources.

In this report a solution is presented which combines procedural algorithms with geological simulation in the form of erosion. A pre-processing stage generates a heightfield using procedural noise which is then eroded. The erosion is accelerated by being performed on the GPU. A road network is generated by connecting points scattered in the world. The pre-processed world is then used to define a field function. The function is sampled in a grid as needed to produce voxels with different materials. Roads are added to the world by changing the material of the voxels. The voxels are then rendered as textured tiles depending on material.

The generated worlds are varied and interesting, much more so than worlds created purely by procedural methods. A world can be pre-processed within a few minutes and explored in realtime.
Abstract

Computer-aided generation of virtual worlds is vital in modern content production. To manually create all the details which today’s computers can visualize would be too daunting a task for any number of artists. Procedural algorithms can quickly generate content, but the content suffers from being repetitive. Simulation of geological processes produce good results but require a lot of resources.

In this report a solution is presented which combines procedural algorithms with geological simulation in the form of erosion. A pre-processing stage generates a heightfield using procedural noise which is then eroded. The erosion is accelerated by being performed on the GPU. A road network is generated by connecting points scattered in the world. The pre-processed world is then used to define a field function. The function is sampled in a grid as needed to produce voxels with different materials. Roads are added to the world by changing the material of the voxels. The voxels are then rendered as textured tiles depending on material.

The generated worlds are varied and interesting, much more so than worlds created purely by procedural methods. A world can be pre-processed within a few minutes and explored in realtime.
## Contents

1 Introduction .......................... 1
   1.1 Related work .......................... 3
      1.1.1 Stochastic models .......................... 3
      1.1.2 Erosion processes .......................... 8

2 The method ................................ 11
   2.1 Overview ................................ 11
   2.2 Initial height-field ......................... 12
   2.3 Erosion .................................. 14
      2.3.1 Model overview .......................... 14
      2.3.2 Flow calculation .......................... 16
      2.3.3 Calculating velocity .......................... 18
      2.3.4 Calculating sediment capacity ................. 18
      2.3.5 Depositing and dissolving ...................... 19
      2.3.6 Advecting the sediment ...................... 20
      2.3.7 Talus flow and movement .......................... 21
      2.3.8 Applying the erosion ...................... 23
   2.4 Cities and roads .......................... 29
   2.5 Stratas .................................. 33
   2.6 Tile generation .......................... 34
      2.6.1 Tiles from erosion and layers ................. 35
      2.6.2 Tiles from brush .......................... 35

3 Implementation .......................... 39
   3.1 Compute Shaders .......................... 39
   3.2 Texture Representation .......................... 40
   3.3 Channels and Precision ......................... 40
   3.4 Data dependency .......................... 41
   3.5 Handling of edges in erosion ...................... 41
   3.6 Real time rendering of erosion ...................... 42
   3.7 Storage of eroded world .......................... 42
   3.8 Storage of generated tiles ................. 43
   3.9 Interpolation .......................... 43
3.10 Roads ....................................................... 43
3.11 Road representation ............................. 44
3.12 Road Creep ............................................ 44

4 Results ................................................. 47
   4.1 Performance ...................................... 47
      4.1.1 Erosion .................................. 47
      4.1.2 Roads .................................. 50
      4.1.3 Tile Generation .......................... 51
   4.2 Overall Evaluation ............................. 52

5 Summary .............................................. 57
   5.1 Future work .................................. 57

Bibliography .......................................... 59
Introduction

To generate world-like models with computers has long been subject of various research. The applications for these models are various across many different fields stretching across entertainment in movies and video games(fig 1.1), to geographic information systems and even the military which use them for a wide range of purposes, including combat simulation and training. Producing environments which are rich in detail and realism is quickly becoming more important as the rendering capabilities of modern computers continue to increase.

Current methods include the scanning of real world locations to use as templates, generating landscapes and flora with filtered pseudo-random noise, using recursive fractal methods which recursively add more detail to the world, and the simulation of long and complex geological processes. Simulating geological processes is very computational and data intensive, but produces very realistic and detailed terrains.

Combining pseudo-random noise can be relatively fast, but realistic terrains are hard to create because of the inherent self-similarity involved in the process. Using real world locations as templates can produce very impressive results, but requires existing sample data and can't produce terrains with different characteristics than their source samples. Recursively fractal methods are hard to control and to shape into a desired landscape, and also suffer from self-similarity issues.

This report details the results of a project which aims to create a huge voxel-based world for users to explore. The goal is to make a world which is interesting both locally and globally, set in a mountainous area. The resulting world should be voxel-based, and as such not only the surface of the world but also the subsurface of the world should be well defined. The world should be interesting both on a local scale and on a more global scale as well, ie. it should not be too self-similar.
Introduction

Fig. 1.1: A screen-shot from the game Planetside 2, known among other things for its large worlds. Much of the detail in the game come in form of fluvially eroded features, as can be seen on the cliffs in the background.

Water-based erosion should be used to add detail and structure to the world. The world should be made more interesting by adding flora and human artifacts to it.

To summarize, the goals of the project are

- Large voxel-based world
- Surface populated by flora and human population
- Interesting on large and small scale
- The world should contain eroded mountains
- Subsurface of world is defined

To achieve the creation of such worlds, procedural noise is used to create a height field on the CPU. It is then uploaded to the GPU where water-based erosion further shapes the terrain, with the erosion rendered in real-time. Settlements are positioned in the eroded landscape, and a road network connecting the settlements is built by creating cost maps on the GPU. When the user visits a location, the preprocessed eroded landscape and road network is used to fill a volume with voxels. Different material are assigned to the voxels depending on how deep beneath the original uneroded surface a tile is situated. Water and dirt may cover the bedrock as a result of the erosion. The road network is then added to the world by changing the voxels near the surface into an appropriate road material. The voxels are represented and visualized as textured cubic tiles.
The remainder of this chapter details related works, current advancements and techniques for world generation. Chapter 2 goes into detail about the method used. Chapter 3 contains some of the more interesting implementation details. Chapter 4 is an evaluation of the project. Chapter 5 is a summary and place for reflection about the project.

1.1 Related work

Through the years, a lot of effort has been spent on improving models to create and represent landscapes. Common techniques include Perlin Noise- and Diamond-Square-generation of terrain, which are simple to both understand and implement. They are part of the stochastic models, which create landscape by synthesizing terrain with the same stochastic properties as real observed terrain.

Then there are models which emulate the creation process of landscapes. These are often pretty complex both to explain and implement. Among these a pretty popular option is to emulate erosion, and most often fluvial erosion at that. Erosion processes are relatively simple and the resulting terrain is often both spectacular and realistic.

The related works have been grouped together based on these classifications and are ordered by publication date, to make it easier to get an overview of different techniques and how they have evolved with time.

1.1.1 Stochastic models

Fournier describes in [5] the problems associated with representing large virtual terrains. In order to represent macroscopic features as well as close-up detail, one needs to create vast databases of information, even though only a tiny portion of the database is used for rendering. From a perspective where the whole terrain can be visualized, a vast number of polygons are invisible when rendering because of their miniscule size. Up close a single triangle may cover the whole screen and it becomes impossible to perceive the rendered image as that of a terrain. While texture mapping can help alleviate the latter problem to a certain extent, the former problem remains. In rendering and modeling, the process of subdivision is used to recursively refine a mesh to until enough detail is obtained. Building on this idea, an algorithm is proposed which can recursively produce and refine details of any desired level in order to create a stochastic surface. This stochastic surface and algorithm is created in an effort to emulate the stochastic properties of surfaces created with Mandelbrot’s fractional Brownian motion (fBm) introduced in [14]. In one dimension the stochastic properties of the proposed algorithm matches that of one-dimensional fBm, but in two dimensions it does not. However, as the global stochastic properties of the algorithm matches those of two-dimensional fBm and the result is a reasonable approximation, it is preferable to use the algorithm as opposed to having to create a large database representing the terrain with fBm-integration. The devised algorithm is commonly referred to as Midpoint Displacement, since the vertices introduced
with subdivision are placed at the mean location of their surrounding vertices and then offset with a Gaussian variable. The standard deviation of the Gaussian variable is given by

$$\sigma = k \cdot 2^{-iH}$$  \hspace{1cm} (1.1.1)\]

Here $k$ is a scaling factor, $H$ is the fractal dimension and $i$ is the iteration level.

Ken Perlin writes about the Pixel Stream Editor in [19], a tool for developing procedural algorithms that describe the surface and texture of objects. Of special interest is the scalar-valued Noise-function that takes a 3d-vector as argument, which is defined as a stochastic function limited by three constraints: Stochastic invariance under rotation; Stochastic invariance under translation; Narrow band-pass limit in frequency. The Noise-function is a very useful modeling primitive because of its properties and simplicity, which makes composition of more complex functions intuitive and easy. One such composition which is very common is the summing of Noise from different octaves and scale

$$\text{height}(\tilde{x}) = \sum_{i=1}^{\text{octaves}} \frac{\text{Noise}(\tilde{x} \cdot i)}{i}$$  \hspace{1cm} (1.1.2)

This makes it possible to generate terrain in a manner similar to that described by Fournier, since addition of higher frequency noise can be omitted once the Nyquist frequency has been achieved. It also provides more control over the created terrain, since it is very easy to control the nature of the noise added from different frequencies. Perlin also suggests warping of input and output domain of the function to create interesting patterns.

![Fig. 1.2: Figures a and b from [21]. Figure c from [7]](a) Perlins original Noise (b) Perlins improved Noise (c) Perlins Simplex Noise

Perlin later improved his Noise in [21] by removing second order discontinuities in the noise. These discontinuities were an artifact of the function previously used to interpolate values. Another artifact of the noise was the presence of axis-aligned bias, which under certain conditions led to overly high values in the
noise. This since the original implementation depended on mixing precomputed pseudo-random gradients. By removing the random element from these gradients a smoother and more homogenous noise is achieved while still appearing random, as the method for selecting the precomputed gradients to use is already random enough.

In 2001, Perlin published another function for generating noise[20] that conforms to his original noise specification better than the traditional Perlin noise does, with it being more isotropic. It is also faster to compute in higher dimensions. The original Noise has a computational complexity of $O(2^n)$ whereas the new noise is $O(n^2)$ with $n$ being the dimensions. Instead of storing gradients in an euclidean grid, values are stored as densely packed as possible in the given dimension. When the noise is to be computed for a certain point, the point is transformed from a skewed space into an euclidean where one can determine the closest neighboring values to the point. These are then transformed back to the skewed space, where values of the neighbors and the distance between them and the point is used to compute a value for the noise. Because of the nature of the algorithm, the noise is called Simplex Noise. It is not usable as an direct replacement for traditional Noise, since its amplitude and frequency spectrum is different. This is however easily mitigated by scaling the input and output domain of the function such that it matches with those from Noise.

In [16], Miller analyzes the problems inherent in the Triangle-Edge and Diamond-Square implementations of Midpoint Displacement, and introduces a new square-square-method in order to overcome these shortcomings. To illustrate the shortcomings of the existing algorithms, Miller sets the $k$ of equation 1.1.1 to 0 in order to remove the random element from the subdivision. He then examines a test-case where the corners and edges of a square are set to a level height, with the center of the square raised above them. In the case of Triangle-Edge-subdivision, the flaws of the scheme become apparent after only one iteration; As the triangles produced are already planar, further iterations will produce triangles which lie in the same plane. In the case of Diamond-Square-subdivision, dents, bumps, and sudden discontinuities in the surface normal are introduced in an otherwise smooth surface. The center of the surface turns into a sharp point. Both these methods produce a surface which goes through the original control points. Millers proposed algorithm, called Square-Square, sacrifices this requirement in order to obtain a smooth and continuous interpolant. The algorithm does not have the same innate roughness as the previous algorithms, but it is argued that roughness can be introduced by lowering the fractal dimension $H$ or exaggerating the initial control points.

A different approach to modeling landscapes is taken in [2] where Belhadj starts out by creating a skeleton of the key features in the terrain to be generated. Ridges are created by representing a ridge as two particles initially located at the same position and height but with opposite direction to each other. For a number of iterations, these particles travel in their direction while having their
Introduction

(a) Diamond-Square. The dents and discontinuities of normals in the surface are clearly visible

(b) Square-Square. The surface is smooth at the cost of not going through the initial control points

Fig. 1.3: Millers visualized interpolants [16]

direction slightly perturbed. The height of a particle is related to its total traveled distance. If a particle intersects another ridge, iteration is stopped for that particle. The routes that the ridge-particles trace become the skeleton of the ridges. A similar process is done in order to create a river network. River particles are randomly located on ridges and are subject to normal physical properties like mass and gravity, and they travel in a simple height-field made from the ridge skeleton. The routes these particle trace are similarly recorded into a river skeleton. Then by a process called *Inverse Midpoint Displacement* (IMD) the remaining uninitialized height values are generated without modifying the initial values from the skeletons and a plausible landscape is generated.

Giliam de Carpentier combines the advantages of procedural terrain with the artistic skill of level designers in [3]. An application is proposed with which to edit height-fields with procedural brushes. Artists have previously used brushes to elevate, lower or level the height-field within an area. This can be done at interactive speeds and give the artist great control over the produced terrain, but the artist must then create all the details in the terrain manually. While procedural methods are fast, they have not been fast enough to be applied interactively. By accelerating the creation of procedural content on the GPU, de Carpentier overcomes this major hurdle and allows artists to use special brushes which can modify the height-field with procedural content. Common forms of noise are included in the application, as well as two novel forms of noise. One of them is dependent on the direction of the brush by which it is applied and compresses
the input domain in a direction relative to the brush motion. This gives the artist a lot of flexibility, since he can now add terrace-like structure to the side of mountains with the ease of a brush stroke, or create manually create patterns similar to gullies formed by erosion. The second novelty is the introduction of erosive noise. By compressing the input domain of the Noise-function in a direction perpendicular to the gradient of underlying terrain, gully-like features are added to the terrain. The actual implementation is more sophisticated that this simple explanation, and with some additional warping of the input domain additional realism can be added to the noise. However, erosive noise lacks global patterns which emerge in normal erosion trough simulation, such as rivers and drainage networks.

Haase et al. tries to tackle a problem of fractal terrain in [8]. Procedurally generated fractal terrain is often very self-similar and bad at representing multiple
Introduction

1.6: Terrain by Adams, and the sketch from which it was made[1]

Types of terrain. While Multifractals exists which can tackle part of these problems, they are hard to control with unintuitive parameters. To combat this, a RegionTree is implemented to determine how a certain area in the generated world should be formed. The region tree uses a base height fractal together with a temperature- and rain-map in order to determine which fractal should be used to further enhance the terrain. This allows them to use a plethora of different fractals, each tuned to produce a specific kind of terrain. Furthermore they present recursive Brownian distortion(rBm) as a less homogenous alternative to fractional Brownian motion(fBm).

Adams et al. presents in [1] a method to combine the power of midpoint displacement with user-supplied restrictions on the terrain. Traditionally, the midpoint displacement algorithm suffers from being unable to handle user-supplied features like roads or ridge-lines. What makes this possible is their abandonment of the traditional height-map as underlying data-structure. Instead a triangulation is maintained of points with determined height in the world, with delaunay triangulation maintaining the stability of the representation. Fixed terrain features like roads are added to the model at early stages, while unregulated areas of the world are generated in a fractal manner. Filtering of the terrain is done in order to circumvent the artifacts detailed in [16].

1.1.2 Erosion processes

Kelley introduces rivers and streams into the world generation process in [12]. The reasoning behind this is that flowing water is a major part in the process of shaping and evolving terrain, and the characteristics of a landscape can often be described by the patterns in its network of streams and rivers. By using empirical data, a model is created where new tributaries can be added to larger streams in order to refine the network, much as additional data can be generated by fractal methods. A surface is then generated where the stream network imposes ten-
1.1 Related work

Ken Musgrave introduces what later came to be referred to as Hybrid Multifractals and Ridged Multifractals in [17]. Using Perlins Noise-function as a base for terrain generation trough iterative scaling and addition as in eq. 1.1.2, he modifies it so that the scaling of the next octave depends on the value of Noise in the current octave, among other things. This leads to local behavior and characteristics in the produced terrain, such as smooth plains and jagged peaks. He also introduces hydraulic and thermal erosion to shape the fractal landscape. The algorithms for erosion are similar to the ones described in section 2, with the notable differences that Musgraves original implementation ran on a CPU instead of a GPU, and handled polygonal terrain rather than a 2-dimensional height-field.

Xing Mei performs the computationally intensive erosion process on GPU’s in [15] in an attempt to reach interactive erosion speeds. By using shallow water equations and a pipe model[18], the process is parallelized to take advantage of the massively parallel hardware in modern GPU’s. The data values for the computation are stored in a 2D texture. The next iteration of these values are written into another render-target texture by rendering a full-screen quad with a certain shader program. This shader program consists of a pass-through vertex program, and a fragment program which works as the computation kernel. It uses data from previous iterations to compute a new value for every pixel in the output texture. This process is repeated in a number of passes with different textures bound for input and output, as well as different fragment programs.

St‘ava et al adds multiple layers of materials to their erosion model in [22]. Their solution is also implemented to run in parallel on GPU’s, much like in [15]. The
addition of multiple layers of material make it possible to erode more complex features, as each layer can have its own physical properties and thus be eroded at different rates. The different layers of material are represented as stacked height-fields, the sum of which is the height-field representing the surface of the terrain. The erosion of these layers is actually pretty straightforward; Only erode the topmost layer, and only deposit transported sediment to the topmost layer. In the case where a layer is empty, erode the one underneath. A new form of erosion is also modeled, where slow-moving water penetrates the ground and dissolves the underlying material. This regolith is then subject to the shallow-water model, which has the effect of gradually smoothing out the bottom of the water body.

Balázs Jákó expands Mei’s model[15] in his paper[10]. A modification of the equation for determining the sediment capacity of water makes the water velocity have less impact at greater depths. This comes from the observation that the bottom of oceans and seas of great depth do not erode even if there are great currents in them since the currents are closer to the surface. The bottom of streams and rivers on the other hand are always eroded because of their close proximity to the water surface. The impact of flowing water hitting a blocking surface is also increased by more properly taking into account the angles between the terrain and the water flow.
2

The method

2.1 Overview.

My method is split into a few separate steps. Most steps use results from the previous steps to create more and better content for later steps, and every step adds more details to the final world. The process is illustrated in figure 2.1.

The first step is using pseudo-random noise to create an initial height-field which is later processed. The shape of the height-field, along with parameters in later stages, largely determines the final shape of the created world.

This height-field is then used as input in an erosion process which produces a new eroded height-field. Flowing water is simulated in the height-field, which transports soil and weathers rock, creating rivers, valleys, mountain crests and plains. The parameters for erosion along with the distribution of water through rain are the factors which determine what the final terrain will look like, and these have been tweaked to produce a world suitable for exploration in games. Measures of water and flow are by-products of the erosion, and these are stored for later use as they contain much information about the world.

After this, a number of markers for cities are placed in the world by evaluating slope, water levels and other things. These cities are used as starting points when creating roads in the world, as they try to form roads to their closest neighbors while trying to minimize the cost of building roads. Existing roads are preferred to building new ones, while water and steep slopes are avoided if possible. A graph is built of the network, and this can be further manipulated to create bridges, tunnels, or belts of hairpin curves on steep slopes.

Since the world is going to be generated with big cubic-meter voxels, it feels natu-
Fig. 2.1: Process illustrating the workflow of creating a world.

real and important that the world beneath the surface isn't homogeneously made of the same material. As real geological processes are too big and complex to model within the scope of this project, a simpler approximation has been made where the sub-surface world is split into discrete layers which emulates the different stratas of material we encounter in the real world. These layers are of pseudo-random thickness and depth and are made of mostly the same material.

The steps until now are all part of preparing the world and organizing data to enable fast creation of local content. When a volume is to be created every cubic meter is evaluated to determine what material is supposed to be there, be it air, soil, or some kind of rock. When the volume is filled, it is checked for the presence of roads which are applied in a brush-like manner.

There is some theory interspersed in this chapter. This is because the amount of theory is pretty small and it was not deemed worthy of having a chapter of its own.

2.2 Initial height-field

As the initial height-field largely affects the characteristics of the final world, it has been experimentally composed by adding noise of different frequencies and amplitudes together. Besides being used for erosion, the height-field is used to determine how far below the uneroded surface a tile is, in order to determine what kind of material layer it belongs to.

As briefly mentioned earlier, the height-field is the starting point for the erosion
2.2 Initial height-field

process, and as such it has a great impact on the result. It is therefore important to choose a method of height-field generation that together with erosion produces an interesting environment with both mountainous areas and plains. Midpoint displacement does not offer this level of control, which is why the initial height-field is composed of pseudo-random simplex noise\cite{20}. Simplex noise was chosen as the noise basis function because of its very good properties such as being quite rotationally invariant. There are no obvious directional artifacts in simplex noise as compared to the simpler Perlin noise\cite{19}. It is also limited in frequency and amplitude, which makes it easy to compose a function with appropriate properties by scaling the input and output domains.

So to create a correlation between the height value and an input $x, y$ coordinate pair, a function $height(pt_{x,y})$ is defined and used to populate the height-field array.

$$height(pt_{x,y}) = \alpha \cdot \sum_{i=0}^{octaves} \frac{Simplex(\beta \cdot pt_{x,y} \cdot 2^i)}{2^i}$$

(2.2.1)

In this equation $\alpha$ represents a base amplitude, corresponding to the maximum height of the largest features. The value $\beta$ represents a base frequency, and is the reciprocal of the size of the largest features. As can be seen from the formula, noise is added with different amplitudes and frequencies. For every octave of noise, the amplitude is halved while the frequency doubles. This has been shown in earlier works\cite{17} to be a good approach to generating mountains and it has a nice power spectrum. The base amplitude and base frequency are related to how wide and high mountains will be, and the number of octaves determine how many layers of extra detail will be added. A value for the number of octaves can be obtained by determining a limit for when you don’t want additional features of a certain size. For example, if the smallest feature you want is a hill 10 meters wide, it doesn’t add any value to continue adding noise of higher frequencies than that.

By iteratively testing different versions of the above equation through erosion, a modified version which is a mix of eq 2.2.1 and something akin to Ridged Multifractals\cite{17} was developed, where every second octave is regular simplex noise and the rest are ridged. This adds some initial ridges to the world and gives more interesting structure to the eroded versions of the height-field. For a comparison between the two models, see fig. 2.2.

$$height(pt_{x,y}) = \alpha \cdot \sum_{i=0}^{octaves} \frac{Noise(\beta \cdot pt_{x,y} \cdot 2^{-i}, i)}{2^i}$$
The method

(a) First version of the height-field   (b) Second version, with ridges

Fig. 2.2: Initial uneroded height-field

\[
\text{Noise}(p_{x,y}, i) = \begin{cases} 
1 - |\text{Simplex}(p_{x,y})| & \text{even } i \\
\text{Simplex}(p_{x,y}) & \text{odd } i 
\end{cases}
\]

Because of the ridged part which is strictly positive, the final range of the height-field becomes much harder to approximate, whereas eq 1 is roughly centered around 0, the produced height-field is re-located by subtracting the mean of the height-field values, thereby centering it around 0. This makes some later calculations easier to perform.

2.3 Erosion

The erosion of the initial height-field is what gives the world its unique look. Erosion is what forms the grand features in our world by repeatedly exposing rocks and mountains to strenuous forces. Erosion comes in many forms and shapes, such as bacteria working on less dense rocks, chemical interactions which change the composition of rocks, or just plain old force hammering the surface until parts break loose.

This method implements erosion through running water, where soil and rocks are dissolved in water to be transported elsewhere for deposition. Material slippage is also implemented which roughly accounts for some of the other forms of erosion. Material transportation through water combined with slippage allows for some more aggressive settings in the water erosion in order to create impressive environments faster.

2.3.1 Model overview

The model for erosion is grounded on a number of 2-dimensional fields of vectors and scalars. As such the water is treated as vertical columns with fixed base area
and variable height. The height field produced in the previous step is the main input for the erosion. The height field $h$ is interpreted to be composed of rocks and hard stone. A similar field $d$ for dirt, soil and eroded particles lies on top of this field, and eroded material is deposited into this field. On top of this there lies the field $w$ representing the water columns, where every scalar represents the height of the water. See fig. 2.3 for a visualization of this.

There is yet another scalar field $s$ which is harder to put in direct relation to the other three other height-fields, and this is the field which represents how much sediment or material there is in the cell. The sediment or material is modeled as being dissolved in the water and flows with it until the water stops flowing.

Then there is a field $f$ of 4 scalars per cell, which models the flow out of cells. This field is additively updated based on total height difference of water between neighboring cells, and limited by the total amount water available for outflow during a simulation step. A velocity field $v$ composed of 2D-vectors is computed based on the flow field, and is used to determine the sediment capacity for a cell as well as adverting the carried sediment.

Lastly there is an 8-valued talus movement field $t$. Much like the water movement, the talus movement field keeps track of how much material falls into lower neighboring cells. Unlike the water flow, all eight neighboring cells are considered for the talus slip-page.

The process by which the erosion is simulated is as follows

1. Water is added to the model (Rainfall, springs, ...)
2. Water flow is calculated and water is transported
3. The velocity field is updated and material is dissolved/deposited
4. Sediment is transported along the velocity of the water
5. Water evaporates

![Fig. 2.3: Visualization of the water, dirt and rock fields](image)
2.3.2 Flow calculation

The water flow and movement is based on the pipe model, where every cell is connected to their neighbors with a virtual pipe. The height difference between two cells create a pressure difference which accelerates the flow between the cells every simulation tick. The flow out of a cell is limited by the available amount of water in the source cell during the tick, not accounting for any eventual water which enters the cell during the tick. In this model, the pipes and cells only exchange water with the four closest neighbors. The decision to use 4 instead of 8 neighbours for the virtual pipes is based on the findings of Mei in [15], where it is determined that using 4 neighbours is sufficient for satisfactory results.

The equation with which the flow values \( f \) initially were updated with is

\[
  f^i = \max \left( 0, f^i_t + \Delta t \cdot \frac{A \cdot g \cdot \Delta h^i_t}{l} \right)
\]  

In this equation, \( i \) is used to denote the direction north, east, south, or west while \( A, g, \) and \( l \) are constants for cross-sectional area of the virtual pipe, gravity constant, and length of virtual pipe respectively. \( \Delta h^i_t \) is the total height difference between the current cell and the cell in direction \( i \). For a more thorough explanation of this calculation, see [18]. Since we only keep track of flows out from the cell (The neighboring out-flows are by symmetry the in-flow into this cell) we impose the limit that values never be lower than 0. See fig. 2.4 for an illustration of two cells with a virtual pipe and \( \Delta h \) between them. In this and most following equations \( \Delta t \) represent the timestep.

A problem with this model is that it doesn’t have any form of dampening, and will endlessly oscillate or may even form never-stopping vortices in lakes, a property I found undesirable since the flows would lead to water in lakes never deposit-
ing the transported material. Even worse, the vortices and oscillating pools may erode away at the terrain where the water would otherwise lay dormant until external forces put it into motion again. I have therefore introduced a dampening constant $\zeta_f$ into eq. 2.3.1. The range for $\zeta_f$ in my experiments has been $[0.75, 1.0]$. The upper limit is there for obvious reasons and values around the lower limit makes the simulation feel very sluggish. The final equation for calculating the flow is

$$ f^i = \max \left( 0, \ zeta_f \cdot f^i_t + \Delta t \cdot \frac{A \cdot g \cdot \Delta h^i}{l} \right) \quad (2.3.2) $$

If a cell has a steady out-flow while not receiving any new water through in-flow it will eventually run out of water, and when it does it makes sense that the total flow out of the cell does not exceed the amount of water already within the cell. If such a situation is detected, the outflows are scaled so the net result of water in the cell is 0. A simple method to implement this is to create a scaling factor $k$ with which to multiply the flow

$$ k = \min \left( 1, \ \frac{w \cdot d_x \cdot d_y}{(\sum_i f^i) \cdot \Delta t} \right) $$

In this equation $d_x$ and $d_y$ are the breadth and width of the cell. The fraction is a ratio of the volume of water within the cell compared with the volume of water traveling out of the cell.

$$ f^i_{t+\Delta t} = k \cdot f^i_t $$

When the flow field has been completely updated it can be used to calculate the next iteration of the water field. The sum of the flow going into the cell from the neighboring cells minus the sum of the flows out of the current cell, multiplied with the time step gives us $\Delta V$, which is the change in water volume in the cell.

$$ \Delta V = \left( \sum_j g_j(i) - \sum_i f^i_{t+\Delta t} \right) \cdot \Delta t $$

Here $g_j(i)$ is the flow in the direction opposite of $i$ out from the neighbor in direction $j$. As we now have $\Delta V$, we can easily update the water level with

$$ w_{t+\Delta t} = w_t + \frac{\Delta V}{d_x \cdot d_y} $$
With this the calculation and updating of water levels is complete.

2.3.3 Calculating velocity

When the flow and water levels have been updated, we have all we need to calculate velocity and sediment capacity for this tick. In the original implementations the velocity of the water in the positive $x$-direction was calculated with

$$v_x = \frac{\Delta F_x}{d_y \cdot w}$$

Where $\Delta F_x$ is the average flow of water in the cell in positive $x$-direction. I have had problems with this when the water level is very low such as when water slowly seeps into a cell; Just as the water starts to fill the cell, there is a spike in calculated velocity which later subsides as the water level increases. This has caused instabilities in the model as the velocity directly affects how much material may be eroded, causing very little water to dissolve large amounts of material. In the extreme cases more sediment is carried in the water than there is water.

Experiments were carried out to find reasonable causes for this undesired behavior. One revelation was that an uneven distribution of rain water may cause this, which is remedied by distributing water within a larger area with more water near the center of the rain source. This does not solve the underlying problem however. Another solution that was tested was to implement a threshold on the water level; Until the threshold is exceeded the velocity is set to 0. With a good threshold the simulation became stable and erosion nice looking, but a drawback is the need for a certain amount of water. The implications of this is that streams and small creeks do not form until the threshold is reached as opposed to the more natural erosion which would occur everywhere there is running water.

On whim, experiments where performed in which the average flow of water was almost directly interpreted as the velocity of the water in the cell for subsequent calculations. While this entirely disregards the physical correctness of the calculation, the results where very favorable since it produced the most stable and visually appealing erosion.

$$v_x = \frac{f^{east}(west) - f^{west}(center) + f^{east}(center) - f^{west}(east)}{2}$$

2.3.4 Calculating sediment capacity

This model of water transporting material is based on the assumption that a volume of water cannot carry more than a specific amount of material. If a volume of water is saturated with material and the capacity is lowered, material will be deposited. On the other hand, if the water isn’t saturated there is room for material to dissolve into sediment carried by the water. Things which determine the sediment-carrying capacity $C$ of the water include the velocity of the water, the
slope of the terrain ($\alpha$), and a modifier which depends on the depth of the water. A constant $\zeta_c$ is introduced to control the overall capacity of the water.

$$C = \zeta_c \cdot \sin(\alpha) \cdot |\vec{v}| \cdot depth\_Modifier(w)$$

In areas of little slope where $\alpha$ would be 0, the total capacity for the water would become 0. Therefore $\alpha$ is clamped to never go below a certain threshold.

The depth modifier was added to the model in [10] as a way to prevent the bottoms of deep lakes and rivers from being to actively eroded. By making the capacity decrease linearly with the depth of the water until a certain depth where the capacity is 0, sediment which is carried by streams and rivers from the mountains can find a place to rest as they enter the deeper parts of lakes. The $depth\_Modifier$ is thus calculated as

$$depth\_Modifier(w) = \begin{cases} 
0 & w \geq \text{limit} \\
1 - \frac{w}{\text{limit}} & \text{else}
\end{cases}$$

### 2.3.5 Depositing and dissolving

When the sediment capacity has been calculated, its time to determine if material should be deposited or dissolved in the water. As the model has a layer of dirt on top of the rocks, this layer needs to be dissolved before any rock can be eroded by the water. And when sediment comes to rest, it does so as dirt even if it originally came from the rock layer. Dirt and rock have different constants which determine how fast they are dissolved in the water ($\zeta_{dis}$), and there is a constant $\zeta_{dep}$ to govern how fast dirt is deposited. When material transfer occurs, a percentage of the difference between the capacity and actual carried sediment is converted according to these constants.

As the height of the layers are changed when material is converted to or from sediment, the total height of the column of rock, dirt and water changes. This introduces instabilities and does not reflect how water behaves in the real world. To counter this, the height of the water column is changed by the same amount that the dirt or rock layers changed. Thus the total height and volume of the column remains the same, as it would in real life.

To summarize, if the carried sediment $s$ exceeds the capacity $C$ then

$$d_{t+\Delta t} = d + \Delta t \cdot \zeta_{dep}(s_t - C)$$
$$s_1 = s_t - \Delta t \cdot \zeta_{dep}(s_t - C)$$
$$w_{t+\Delta t} = w_t - \Delta t \cdot \zeta_{dep}(s_t - C)$$

Similarly, if the water is not yet saturated then
\[ d_{t+\Delta t} = d - \Delta t \cdot \zeta_{dis}(C - s_t) \]
\[ s_1 = s_t + \Delta t \cdot \zeta_{dis}(C - s_t) \]
\[ w_{t+\Delta t} = w_t + \Delta t \cdot \zeta_{dis}(C - s_t) \]

In the case that the dirt layer is or becomes empty, the rest of the material to dissolve comes from the rock layer instead of the dirt layer.

### 2.3.6 Advecting the sediment

When material has been converted to or from sediment, it is time to move the sediment with the water in an advection step. This is achieved by doing a backwards particle trace from every cell.

\[ s_{t+\Delta t} = s_1(x - v_x \cdot \Delta t, y - v_y \cdot \Delta t) \]

A particle is placed in the middle of the cell. Then using the velocity field \( \bar{v} \), the particle follows a traced path backwards for a specified time. Then an interpolated value is obtained from the sediment field at the particles position, and this value is assigned to be the current cells new sediment value. From experiments it have been determined that the accuracy of the tracing of particles is not of very high importance, and a single Euler step is enough to obtain reasonably good values. The reason for this lack of need for accuracy is attributed to the short time steps and low velocities involved in the simulation. A good property of this way of moving the sediment is that the simulation becomes unconditionally stable.
2.3 Erosion

2.3.7 Talus flow and movement

The second form of erosion is based on material slipping down slopes that are too steep. This emulates many forms of erosion which result in smaller particles and grains breaking loose from the main body of material. Thermal weathering is an example of this, where rapid changes in temperature causes internal forces in the material as it expands or contracts as a result of the temperature. When the particles and grains are made loose, they tumble downwards as a result of the forces of gravity. Eventually they reach a point where they are laid to rest, as the slope they lie on isn’t steep enough for them to continue falling or sliding. As a lot of material is made loose this way it collects at the base of the rock walls they come from and form a slope, reaching up towards the wall. This collection of smaller fragments is often called scree, or talus. An example of this can be seen in fig. 2.5.

This model is based on the observation that such collections of talus are formed with an angle of repose. This angle is the steepest angle the slope can obtain without causing upper material to slip down the slope, and is related to factors such as fragment size and friction coefficients. In computer simulated erosion, the term talus angle is usually used instead of angle of repose. In this model, it is implemented by detecting slopes between cells where the angle exceeds a material dependent talus angle, which is different for rock and dirt. When the angle is exceeded, a portion of the material exceeding the limit is transported to the lower cell. If a cell is surrounded by multiple lower cells, then the total amount of material to transport is normalized across the neighbors.

Since there are two layers of material (rock and dirt) in the model, they are handled one after the other. Rock is handled first, as talus movement in the lower layer may give rise to further movement in the layers above, while the opposite doesn’t hold true.

As the calculation is performed on the GPU it’s not feasible to change both the value of the providing and the receiving cell when controlling the talus angle, which would be possible were it implemented on the CPU. To solved this, a talus...
flow is calculated. This flow works very much like the water flow, in it being defined as the amount of scree slipping out from a cell. The similarity ends there though as the talus flow is not incremented across multiple calculation ticks, and since the material slippage is performed with all 8 neighbors of a cell whereas the water flow is only calculated for the 4 closest neighbors. While it was found that 4 neighbors was enough to simulate water for erosion[15], it is not enough to produce acceptable thermal erosion as the thermally eroded features became noticeably axis-aligned. Figure 2.6 shows the difference between using 4 and 8 neighbors for thermal erosion.

A value $\Delta H$ is calculated, which represents the height difference between two cells including the height of any underlying layers. This value is then compared to the talus limit $t_l$, which is the height of a slope of length $d$ with the talus angle $\alpha$. The value of $d$ corresponds to the distance between cells.

$$t_l = \tan(\alpha) \cdot d$$

The amount of material eligible for transport is then the difference between this value and $\Delta H$. Since we only consider outwards flow, this value is clamped to never become negative.

$$t_j = \max(0, \Delta H - t_l)$$

As mentioned earlier, the initially calculated values are then normalized. If the amount of material available runs out, it is normalized with respect to this value. Otherwise, it is normalized with respect to the highest value of $t_j$. The reason for this is that while separately the values of $t_j$ would cause the height difference between the cells to become $t_l$, together they may remove enough material to
2.3 Erosion

cause the resulting height differences to become less than \( t_j \) and may even cause the cell to have a lower final height than the surrounding cells.

When the flow values have been normalized, the material is ready to actually be transported. This is performed very much like the water is transported, like thus

\[
h = h + \Delta t \cdot \left( \sum_j n(j) - \sum_j t_j \right)
\]

Much like with the water calculation, a notation \( n(j) \) is introduced here to simplify the equation, and it corresponds to the flow-value in the direction opposite of \( j \) in the neighboring cell in direction \( j \).

The slippage and water transport work exceptionally well together. The water transport cuts deep lines in the ground. The walls of the rivers then 'collapse' into the water by the slippage, and eventually the material is transported to other locations by the water. Over long periods of time winding valleys form in the mountains, and ridge-lines emerge which separate them.

2.3.8 Applying the erosion

Now that all parts of the erosion have been explained we can shift focus to how this erosion is applied to the height-field in order to produce the eroded terrains in this solution.

In the beginning the erosion was performed with random raindrops being distributed before each erosion iteration, but this was later replaced with an even distribution of water in every cell in the simulation. This led to what was perceived as a more even and stable erosion process. After the water was distributed and erosion applied, an evaporation step was performed where a percentage of all the water in a cell was removed.

While this produced an interesting terrain to explore with many ridges and valleys, the result looks very uniform in the distribution and size of the eroded features (as in fig 2.7). The areas which are not obviously eroded are very close to their initial form. In order to change this homogenous look, variations of erosion parameters where tested. Eventually it was found that by letting a large initial amount of water erode the height-field without additional rain for a while, an interesting mix of mountains, valleys, and plains form. This also resulted in interesting bodies of water consisting of large lakes and rivers.

A side effect of this is that the world becomes very smooth. The talus movement have transformed all steep inclines into smooth slopes with the same angle, and the plains that form as the water recedes is almost too smooth. Therefore some relatively high-frequent noise is added to the maps, and a second pass of erosion is applied with a small uniform rain. This pass of erosion creates smaller flows of water and adds lot of information and depth to the final world. The whole process is visualized in figures 2.8-2.16 where the images are taken at different
Fig. 2.7: Homogenously eroded terrain. The resulting shapes and sizes ridges and valleys are all very similar to each other.

Fig. 2.8: Uniformly distributed water. The original height-field is clearly visible.
2.3 Erosion

Fig. 2.9: The water flows down and erodes away at the height-field

Fig. 2.10: Larger bodies of water start to form
Fig. 2.11: The water continues to recede, leaving smooth areas behind

Fig. 2.12: What will be future rivers start to emerge
2.3 Erosion

Fig. 2.13: When the water collects and levels rise, banks are breached

Fig. 2.14: The end of the first phase. Mountains and plains are now defined
Fig. 2.15: Start of the second phase. Random soil is added along with more rain.

Fig. 2.16: The final product of the erosion process with these changing parameters.
times in the erosion. This method of eroding terrain is different from the normal way terrain is eroded in that the parameters and water distribution is changed during the erosion process, whereas they are normally kept constant. This is necessary to create the varied and interesting look of the world.

2.4 Cities and roads

When the terrain has reached its final shape and the simulation of water has stopped, it’s time to begin populating the world with things an explorer might find interesting. Rivers, ridges, and distant mountains already provide natural ventures for exploration, but the addition of landmarks create by humans add another layer of interesting features. Roads, like rivers and ridges, are something that piques our interest. Questions such as where does it come from and where does it go or who built it and why motivate us to follow the road instead of wandering back into the wilderness.

The first step in growing a complex and interesting road network is to find suitable places for the roads to connect to. These endpoints correspond to cities or villages, which are semi-randomly scattered in the world. This is achieved by randomly choosing a position in the world, around which an area is examined to find the most suitable place for a settlement. The village score is a function of average slope in the neighborhood, the access to fertile land with good soil, and the proximity to running water. Fertile soil is present where the dirt field $d$ is above a certain threshold. The position is rejected if the score is not high enough or if another city is in close proximity. This is repeated until there is a predefined number of settlements in the world.

When the endpoints have been established the cities that they represent compute a list of their four closest neighbors. This information is then used to form a list of connections between cities, representing a road to be built. The list is sorted based on the cost between the endpoints such that the shortest connection lies first, after which the roads are built. The reasoning behind this sorting is that since the creation of roads take into account any previously built roads, long roads should be able to re-use the shorter roads. It makes more sense that two closely placed villages have a short and optimal road connecting them while the roads connecting far-away cities may take small detours to use these roads, than
The method

(a) Iteration limit = 2 times Manhattan distance

(b) Iteration limit = 5 times Manhattan distance

Fig. 2.18: Cost maps produced with current cost rules, with the best path marked. Both images represent the same area. Black pixels have not received any distance value.

The small village-roads taking detours to connect to the city-connecting highway roads.

In order to determine the path between endpoints \(a\) and \(b\), a distance map is computed on the GPU in OpenGL Compute Shaders. The cost map is initialized with a very high value except for the point \(a\) from which the cost is computed, which is initialized with 0. Then, for a number of iterations, every point in the map is updated by evaluating if the value of a neighboring point plus the score between them is lower than the previous value. After a number of iterations equal to the Manhattan distance between the endpoints there is a guaranteed path found between them. Every iteration thereafter improves the path, until no better path can be found. It is of course hard to determine when this occurs, but experiments have found that after an iteration limit equal to \(1.7 \cdot |a - b|\), most optimal paths will have been found. In the cases where an optimal path has not been found at this point, there is still a guaranteed path which will be pretty good. As the distance between the points increases the amount of pre-made roads increases, and the ratio between the iteration count before no better path is found and the distance decreases. In figure 2.18 and 2.19 different iteration limits have been tested in an extreme case.

When the cost map has been computed, it is downloaded to the CPU and examined to find the best available path. Since the cost map is built around endpoint \(a\), starting at any point and walking towards the lowest neighboring point will
2.4 Cities and roads

Fig. 2.19: The roads without further processing, rendered as lines on the height-field

(a) Limit = 2 times Manhattan distance  (b) Limit = 5 times Manhattan distance

eventually lead us to point \( a \) in the least possible number of steps. These steps are inserted into a graph-like structure, which consists of vertices that represent endpoints, and edges which represent roads. Each of these edges consists of references to the vertices and a list of points through which the road passes. When the cost map is traversed to find the road from \( b \) to \( a \), the new road might intersect previous roads, follow them for a while and then branch of again, and in these cases the existing roads are split at the intersection, where a new endpoint is created to connect the roads. In certain cases where a road is re-used, the new road might split off and join again in a very short distance. These cases are detected and the short alternative routes are pruned from the graph. In figure 2.20a the whole process is well illustrated with a cost map, existing roads being used and a new road forming to connect them where optimal.

When new roads have been added to the graph, the points they correspond to are added to a map representing existing roads for use in the construction of later roads.
(a) Road built from a (green) to b (blue). Endpoints or intersections are colored purple. The yellow segments are newly created while white segments represent previously built roads. As in fig. 2.17 the cost-values are mapped to the red color channel. The large black part represents very high cost values and are clamped to being black.

(b) The road network generated in a displayed on the eroded height-field. Note how additional roads have been generated since fig a was generated.

Fig. 2.20
2.5 Stratas

In most simulations and models of world creation, only the surface of the world is considered. Height-fields are just that, a thin sheet defining the interface between the ground and the sky. Often these are just rendered with a few textures that combine together depending on slope, elevation, or pre-defined areas where a certain texture is dominant. They do not account for what might lie beneath the surface. In a world where subterranean exploration is an option these things must be considered and synthesized.

The model which was adapted here is an attempt to emulate the different layers of rock types that are encountered underground. To simulate the whole process of the creation and distribution of the materials underground is a task too big and daunting to implement in the limited time of this project. Instead, the model simply assumes that there are distinct layers of different material in the world. These layers lay on top of each other, and the thickness of a layer is a function of position and material type.

The types of rock are divided into four main categories, depending on how the rocks are formed. These are sedimentary, metamorphic, igneous intrusive, and igneous extrusive rock types. Sedimentary rocks form as sediment on the bottom of seas are compressed together over long periods of time, and include among others siltstone, limestone, and dolomite. Metamorphic rocks form deep beneath the surface at depths around 15-25 kilometers where materials are forcefully fused together under great pressure and temperatures. Examples of metamorphic rocks include slate and marble. The igneous rock types both form as magma cools down and minerals gradually grow into rock crystals as the temperature decreases, the difference being that intrusive rocks form when the magma cools without break-
ing the surface while extrusive rocks form on the surface. *Granite* and *Diorite* are examples of igneous intrusive rocks, while *Basalt*, *Dacite* and *Obsidian* are igneous extrusive rocks.

Besides these categories, there is a special category which represent horizontal layers of magma. The thickness of these layers are subject to special treatment in order to create relatively thin pipe-like tunnels of magma between the ordinary rock layers, making the magma very sparse in occurrence. The same layer configuration may be set to not contain any magma and instead be filled with air, emulating the cases where internal magma has receded and left behind tunnels deep in the earth.

To determine the order, average thickness, and contents of the layers a simple algorithm is used. The algorithm make the layers near the surface relatively thin, with increasing thickness with increasing depth. This is in order to make it more rewarding and interesting to go exploring near the surface. The category and material of the layer is determined by chance. Each material and category type is linked to their own probability distribution function, which depends on among other things the depth of the layer to place, and the types of the neighboring layers. Thus it is less likely to find sedimentary rock types at great depths, and impossible to find several layers of magma stacked on top of each other.

In order to determine what kind of material and layer a position at \( x, y \) and depth \( z_{\text{depth}} \) beneath the surface corresponds to, a function \( idx(z) \) is defined. As all the layers keep track of their average thickness, this function searches the layers until it encounters a layer with average depth below \( z \). To introduce variation in the stratas, the value which is supplied to \( idx \) is \( z_{\text{depth}} \) perturbed by a noise function with relatively low frequency and amplitude.

\[
\text{index} = idx(z_{\text{depth}} + p(x, y, z_{\text{depth}}))
\]

When we know what layer a position corresponds to, we can perform further calculations to change what layer we end up with. An example would be the magma layers, which use another noise function to return the layer above or below, effectively changing the shape of its own layer. The \( \text{index} \) value can the used to obtain further information about the layer from a sorted array of layers. An image has been produced using this procedure in fig. 2.21 which shows a cross section of the ground.

### 2.6 Tile generation

The final form of the world is represented by tiles of 1x1x1 meter size. The tiles have a discrete material type determining what texture to use when extracting the surface between tiles and world, and are stored in chunks called sectors of 128x128x64 tiles. These sectors are loaded from the hard-drive or generated on the fly as needed.
The process of generating a sector can be divided into two stages. The first stage iterates over every position in the sector to determine what tile to use based on input from the erosion and material system. The second stage applies changes to the terrain in order to add artificial constructs such as roads, which are applied in a brush-like manner.

### 2.6.1 Tiles from erosion and layers

In order to determine what kind of tile a position \( x, y, z \) corresponds to, one first has to determine the relation between the tile and the eroded surface. The height \( z_s \) of the surface at a position \( x, y \) is determined by interpolating and adding the values of the rock- and dirt-height-fields, \( z_r(x, y) \) and \( z_d(x, y) \). This interpolation is made with bi-cubic interpolation, but other interpolation methods can be used to subtly change the look of the world. The value \( \Delta z = z_r + z_d - z \) thus indicates how far below the surface a tile is, and the tile type is determined like

\[
\text{type} = \begin{cases} 
  \text{air} & \Delta z \leq 0 \\
  \text{soil} & 0 < \Delta z \leq z_d \\
  \text{rock} & \Delta z > z_d 
\end{cases}
\]

In the case of a tile being a rock tile, further steps are taken to determine what type of rock it is. To do this we need to figure out how deep beneath the original uneroded height-field a tiles position is. This depth value is then used to look up what layer corresponds to the depth. The depth of the tile can be calculated as

\[
z_{\text{depth}}(x, y, z) = z_o(x, y) - z
\]

Here \( z_o \) is the interpolated height of the original uneroded height-field. Using the value of \( z_{\text{depth}} \) and the \( idx \) function, the correct layer is found. This layer then provides the main mineral type to use, as well as a set of additional alternative types. At this point, further usage of noise and level sets can create veins and blobs of these alternative materials. And with this the first stage of tile generation is done.

### 2.6.2 Tiles from brush

The second stage is based on the notion of applying brushes to the world. Much like one would use a fancy shaped brush in *Microsoft paint* to change the color of tiles in a 2D grid, a 3D brush is swept through the world, changing the type of tiles it comes into contact with. In this project the main thing which is brushed into the sectors are the roads generated in the previous stages. If any road passes through the sector, then the path of the road is obtained by interpolating the positions stored within it. The path is iterated over at short distances, and in each iteration a *mask* is applied orthogonally to the interpolated path, in effect painting a gravel road (in places supported by packed dirt) into the world. Since the rotation of the path is not strictly axis aligned, several strokes must be applied
Fig. 2.22: The generated tiles, with different types of rock. The topmost tiles in the dirt-layer are replaced with grass tiles for dramatic effect. Multiple different rock strata are visible at the surface as a result of the rock face being eroded.
Fig. 2.23: A road of gravel brushed into the terrain, at times supported by brown, packed dirt. The path which the road is interpolated along has been marked with a black stroke. The purple strokes in the foreground apply the road-tiles and are spaced much tighter than illustrated to handle sharp turns and odd angles.

per unit length of path to ensure that all the tiles are changed which should be changed.
This project has been implemented in the D programming language, with parts written in GLSL, the OpenGL Shading Language. The D language is a C-like language with fancy features which makes it easier to write high level code. At the same time it allows working directly with memory and pointers, and it interfaces well with OpenGL and GLSL.

## 3.1 Compute Shaders

The erosion process and the road generating process requires OpenGL version 4.3 to work, as the newer features relevant to GLSL compute shaders are extensively used in both processes. The new image store/load functionality introduced in OpenGL 4.2 offers a very simple way to both read from and write to texture objects. The procedures operate on pixel indices, while the more common texture-procedure operates on normalized texture coordinates. Additionally, writing data to texture objects in OpenGL before imageStore was very cumbersome as one would need to create a framebuffer object and bind the texture to it, and then render a full-screen quad where the output of the fragment shader would be the value to store in the texture. As the compute shaders are part of OpenGL no flushing of states is required before or after computing, as is needed when using OpenGL with OpenCL or other GPGPU solutions.

Parameters are handled as named uniforms, which are assigned proper values before the shader is executed, and only if the values have changed. There is no reason to use Uniform Buffer Objects as the advantages of these lie in the fast switching of multiple uniform values and allowing the use of excessively large uniforms.
3.2 Texture Representation

Since the erosion computation is performed mostly in parallel in the GPU, the different value fields are represented and stored as textures. Since the GPU has very efficient texture look-up and caches nearby texels when making a look-up, this representation is very good for the performance of the computation. Alternatively the data could be represented in OpenGL as Shader Storage Buffer Objects or Uniform Buffer Objects (For read-only data). As SSBO’s are essentially just a syntactic representation of image load/store[9], it does not really matter performance wise which interface is chosen. The texture representation was chosen because it provides an intuitive mapping of data and coordinates, as well as being simple to work with using standard procedures for texture manipulation.

Representing the fields as textures comes with a downside, which is a limitation in size; Most GPU’s do not support texture sizes above \(4096 \times 4096\) texels. This is the main limitation of the size of the produced world in this implementation. Even if one where to use larger textures, current GPU’s do not have enough available memory to represent all the fields. Experiments where performed where parts of the world where ‘paged’ into the memory of the GPU from the host computer, but performance suffered considerably, and additional complexity was introduced in the code. Regardless, the size of the generated world is considerable as the distance between sample points is approximately 25 meters; This leads to a world size of 100 \(\times\) 100 square kilometers.

The data in these texture objects is transferred to and from the GPU and CPU through the use of \texttt{glGetTexImage} and \texttt{glTexImage2D}. In certain situations this can be accelerated with the help of Pixel Buffer Objects, as this allows some measure of asynchronous transfer of data. However in this project the transfer of data can not be accelerated, as there is nothing to do while the data is being transferred.

3.3 Channels and Precision

In order to preserve memory on the CPU, the value fields are not all represented with full precision floats (32 bit floats). Some are represented as half’s, meaning they are stored with only 16 bits of information. The fields which are stored with full precision floats are the \textit{Height} and \textit{Soil} fields. These are represented with the \textit{GL}_R32F internal format, which only contain one channel of data (ie. for every position on the texture there is a single scalar associated with it. A texture with two channels would have two scalars associated with every pixel, etc). The \textit{Water} and \textit{Sediment} fields are represented with the \textit{GL}_R16F format, each one channel of half-precision data. The \textit{Flow}-field is represented with the \textit{GL}_RGBA16F format, with 4 channels of data. The \textit{Velocity} field is represented with \textit{GL}_RG16F, two fields of half-precision data. Then there is the talus movement field which contains 8 values per coordinate; This is realized by using two texture with four channels each, represented with \textit{GL}_RGBA16F.
3.4 Data dependency

Since the calculation new values often require the use of neighboring values care must be taken to not write updated values directly into the buffers when they are computed, as this invalidate the state. Even if one would take care to implement a scheme where one reads/writes in a specific pattern to avoid this, the texture caches would need to be flushed after every write operation, which would degrade performance significantly. In order to mitigate this limitation, several of the fields have been provided with two buffers/textures, one which is used as input and one which is used to store the output of the different erosion stages. Which one is read or written depends on the stage being processed. In some cases the texture objects are swapped after a full erosion iteration when they are read/written an odd number of times. The fields which require two buffers are the Height, Soil, and Sediment fields. The other fields either only require local input of their own previous values, or are not used as input for themselves at all. Examples of this is the Flow field which updates independently of the flow of neighbors, and the Velocity field which only depends on the Flow field.

3.5 Handling of edges in erosion

When accessing and computing values in the erosion process, edge cases need to be handled with special care. When accessing values in the Height, Soil and Water fields, the indices are clamped to lie within the texture before issuing a load. The effect of this is that the values of land, soil and water are copied from the edge, as if accessing a texture with wrap-mode GL_CLAMP_TO_EDGE. When attempting to load flow values (water and talus) outside the world, a value of 0 is obtained. The effect of this is that water and matter can not travel out of the world, and there will be no difference in height to create potential flow out of the world. One might think of the effect as if the world is encased in walls of glass, like an aquarium.
3.6 Real time rendering of erosion

As the erosion process is entirely implemented with Compute Shaders, no data is moved to or from the GPU during the erosion. Any attempt to do so degrades performance by at least an order of magnitude. An early implementation downloaded the computed height, soil and water fields to the CPU, where a height-field mesh was updated, uploaded to the GPU and rendered. This ran with only minor lag on smaller world sizes, but as the project progressed and the size increased it became apparent that the rendering would need to be either removed or rewritten. The ability to observe the erosion process was invaluable in determining correct values for parameters and tracking down bugs, so it could hardly be removed. The solution was to build a static 2D-mesh without any height value, and in the fragment shader use the positions of the vertices as indices into the height, soil and water textures. The obtained values where used to determine height and shading of the vertices, producing a 3D mesh for the fragment shader. As no data is moved between CPU-GPU the main performance bottleneck was removed. The rendering time is very small, almost negligible, compared to the time to do an erosion iteration even though no form of culling or geomipmapping is implemented. The shading employed is a simple lambertian reflection with a twist, where the inclination of the surfaces beneath water is exaggerated in order to make it easier to perceive how the erosion process affects the surface beneath flows and bodies of water.

3.7 Storage of eroded world

When all the erosion calculations are finished, the Height, Soil, Water and Velocity fields are downloaded from the GPU and stored on the hard-drive for further use. The files containing these values are memory mapped in the process, meaning the operating system automagically takes care of writing updated values to the hard-drive and only loading what is currently needed. Previously this would have polluted the address space of the process considerably, but with the advent of 64-bit compilers, processors and operating systems this is no longer a problem.

The Height, Soil and Water fields are converted into 16 bit signed integer values (short’s), in an attempt to reduce the amount of storage space required. As the samples are spaced 25 meters apart and are interpolated in between, the need to store sub-meter height values was deemed unnecessary. The erosion process is computed in full precision however.

The velocity field is stored as 2-dimensional 32-bit float vectors. A more compact representation would without doubt be possible, but this was not deemed necessary as the field is very seldomly accessed and disk space is very cheap.
3.8 Storage of generated tiles

When the user is navigating the world, tiles are generated on-the-fly for areas previously not visited. A section of generated world typically contains 3-4 different types of tiles, and most of the generated tiles share similar settings on generation. For example, all tiles generated with tile-type obsidian will have the same properties of hardness, decay and lighting. As such the tile-data is very susceptible to compression. When the user is leaving an area of the world behind, the generated tiles are compressed with the zlib-compression-algorithm[4] before they are saved to the hard-drive. Besides the obvious advantage that the generated tiles occupy less disk space, storing and retrieving tiles is faster due to the decreased amount of bandwidth required. For tiles placed naturally in the terrain, the compression factor is in the range of 8 to 20 times.

3.9 Interpolation

The values produced and stored in the fields for Height, Soil and Water are as mentioned spaced 25 meters apart. This leaves a lot of space in between the samples where interpolation schemes are used to produce height values. Many interpolation schemes where examined in order to determine which would do the best job of producing believable terrain, but no one scheme was superior to the others. Linear interpolation works well in steep inclines and sudden drops, but since the interpolation is of $C^0$-continuity the edges between the interpolated quads are very sharp and perceptible. Different forms of smooth interpolation such as cosine interpolation and smooth-step which are $C^1$-continuous perform badly as they introduce a saddle point where the surface normal always point directly upwards. Cubic interpolation is $C^2$-continuous and does not suffer from this, but the resulting landscape is devoid of sharp changes in slope making it feel very synthetic.

3.10 Roads

The bulk of the work in road construction is performed on the GPU by creating a cost map centered around one of the designated endpoints of the road. This is done in yet another Compute Shader, and once again the inputs and outputs are represented by texture objects. The inputs are height $H$, water $W$, a previously computed cost map $D$ and a map of computed roads $R$. The height field represents the total height of the rock and dirt fields. All the fields are internally represented as single-channel, full precision float textures, or as OpenGL would have it $GL_R32F$. Before the road network is built the $R$-texture is initialized to 0, and when a road has been identified the pixels it would go through are incremented. The road generation is a lot slower than the erosion process, even though it is a lot simpler; This is because every time a road has been computed, the cost map is downloaded from the GPU to the GPU. This is however necessary in order to build a proper graph representation of the roads and to prevent road creep.
As the values in the cost map are propagated one pixel per iteration, there is no need to process the whole cost map at all times. This is recognized and only a part equal in width and height to $1 + 2 \cdot \text{iteration}$ should be processed. Unfortunately OpenGL's Compute Shaders require recompilation in order to change the local work group size, so the actual number of elements processed is rounded up to the nearest multiplier of 16. As most of the roads are of small to medium length this is a nice improvement.

### 3.11 Road representation

When the cost map has been computed, it is iterated over in order to produce a graph representing the road network. Edges in the graph represent the roads, while vertices represent intersections and end-points of roads. The vertices are associated with a position in the world and perhaps with a place of interest, such as a city or village. The edges hold a list of points through which the road or edge passes. The edges also contain an AABB covering all the road points, in order to accelerate look-up of roads given a position in the world.

### 3.12 Road Creep

Road creep is a side-effect of using previously built roads as an input when computing the cost map. It happens when a long road is to be built between to far away places and a short road already covers a part of the distance. The pixels just beside the roads may receive a lower score than it originally would before there was a road there. The result might be that the new road closely follows the old road, but occasionally leaves the road only to return to the road within a few steps. Typically this introduces small alternate paths which are but one pixel long. If multiple roads are constructed which take advantage of a previous road which has had many such detours added, the end result is that the road might creep towards one side of the road, leaving behind a small delta-like network of alternate paths. In figure 3.1, a cost map has been produced which exhibits exactly this behavior. The pixels with an outline represent pixels where a road already exists. To combat this problem, detours are detected on the CPU when traversing the cost map by taking a few additional probing steps whenever the path leaves a road. If within a few steps the same road is entered again, no road is produced for the short detour.
Fig. 3.1: A cost map where old roads have been given a white border. The new road-segment with yellow border is slightly cheaper than the previous road. Near the blue endpoint multiple such alternate road detours have been previously introduced.

Fig. 3.2: The produced roads with unpruned detours. In the middle and lower right of the image, small alternate paths have been introduced.
This project is somewhat difficult to evaluate in an objective fashion, as the correctness of the resulting world can not be measured. The performance, scalability and memory characteristics of certain parts can be measured and examined, but as a whole the project needs to be considered from a subjective point of view. This chapter has accordingly been split into an objective evaluation of the erosion process and road/tile-generation, as well as a part where the overall results are subjectively evaluated.

4.1 Performance

4.1.1 Erosion

There are many different ways to evaluate the quality of erosion. Comparing it with other algorithms make little sense as the difference in algorithms and properties make the erosion behave differently. It would in fact be startling if two different erosion algorithms did produce the same eroded landscape. This erosion algorithm produces pretty landscapes and is quite configurable trough parameters. The input for the algorithm is a set of parameters, and two height-fields representing rock and soil. The output is a modified rock and soil-field as well as information about water volumes and flows at the end of the erosion. In figure 4.1 the same terrain has been eroded for the same number of iterations, with the only difference being the parameters supplied at the start of the erosion process.

As the erosion is based on the movement of water and the water is implemented with the pipe-model, there are some things typical of erosion which this implementation can not handle or produce. A simple example of something which is
Fig. 4.1: The same terrain eroded with different parameters for the same number of iterations
not possible is the creation of meandering rivers. The process of meanders in nature is attributed to the different speeds of water in a curving river. Heavy particles are deposited on the inner side of a river where the water flows slower, while in the outer side of the river where the water flows faster is eroded. It may be that this process would be reproducible within the pipe-model, but it would require a much higher resolution of the height-field and possibly time-step than is available in this project. At any rate, since the eroded material is not represented as actual particles of varying size in this project but as values in a 2D-field, the deposition of heavier particles in slower water is not emulated. In order to produce this kind of effects of erosion, it'd be necessary to model the environment in full 3D as opposed to height-fields.

Something else which is not done in this project is to use material properties from the underground stratas in the erosion. Using material properties for talus angle and solubility in water would add additional details to the world. It would incur a performance hit, and aliasing when determining which strata to use for material properties could produce undesirable behavior. Another thing to consider could be ice covering high peaks, which seems to prevent erosion[6].

**Speed**  The erosion, being performed on the GPU, is very fast. A version running on the CPU was implemented in the beginning of the project as a reference implementation. The ported GPU implementation which is able to take advantage of the parallel nature of the problem ran about 3 orders of magnitude faster on the initial 25x25 grid. With larger world sizes the speedup increases further. Table 4.1.1 shows the wall clock times for three different world sizes. The values recorded in the table are the average of 5 measurements, each performed on the same random-seed. The difference in time between the measurements was incredibly small. It is interesting to note that the smallest world-size takes unexpectedly long to erode, compared to the larger world-sizes. This non-linear scaling of time is attributed to different layout of the world to erode which might affect branching in the computations, as well as cache-related issues of the hardware.

**Scalability**  Since the algorithm is designed and implemented to run on a GPU, it exhibits poor scaling beyond that of the hardware used. Multiple GPU’s in the same computer will not improve performance, and neither will networking of multiple computers. It is further limited to running on computers with a high enough version of OpenGL and with sufficient memory. Had the erosion been im-

<table>
<thead>
<tr>
<th>World size</th>
<th>Number of elements to process</th>
<th>Time / s</th>
<th>Expected Time / s</th>
</tr>
</thead>
<tbody>
<tr>
<td>655 km²</td>
<td>1024²</td>
<td>39.2</td>
<td>-</td>
</tr>
<tr>
<td>2621 km²</td>
<td>2048²</td>
<td>140.7</td>
<td>39.2 · 4 = 156.8</td>
</tr>
<tr>
<td>10485 km²</td>
<td>4096²</td>
<td>583.2</td>
<td>39.2 · 16 = 627.2140.7 · 4 = 562.8</td>
</tr>
</tbody>
</table>

*Table 4.1.1: Time to erode differently sized worlds*
plementated in OpenCL, the erosion would be able to fall back to being computed on the CPU if no GPU with good enough specs is present. And since the CPU has access to a much larger address space than current generation GPU’s, erosion of larger worlds or worlds with more detail would be possible at a slower pace. An efficient distributed networked implementation might be hard to implement since there are a lot of interdependent steps and neighboring data which needs to be shared, but if larger worlds are of interest it might be an approach worth exploring.

### 4.1.2 Roads

The generated roads are good and add places of interest to the world. The implemented system represents the finished roads as a graph, which makes it very extendable; additional roads and paths may be joined to the roads in a simple manner. The graph can be used on a global scale to accelerate path-finding in the world. Where a road crosses a chasm or bodies of water, parts of the graph may be replaced with more fitting and specialized structures such as bridges, stairs or serpentine roads. A drawback of the system is that the roads are represented as a series of points in an euclidean grid. When the roads are generated, a point may be connected with any of its eight neighbors. This severely limits the number of directions a road may face, and the road network looks very angular. To help alleviate the rigid look of the roads, they are interpolated with a cubic spline from the points when local tiles are generated as in figure 4.2b. In the cases where roads stretch for a long distance in the same direction the roads themselves look very bland as in 4.2a. On a plain this is very boring, but when the road goes along the slope of a gorge it is very interesting.

When the roads are generated, they are generated in the same grid as the sample points from the erosion. Later these height-field values are interpolated to define the surface between the samples. The road generating process does not take this into account, which sometimes leads to roads which appear to be very stupidly located when the height-fields shift dramatically. Likewise as the roads themselves are interpolated with a cubic spline, they are sometimes subject to overshooting which make them curve weirdly before sharp turns.

**Speed** Calculating and building a graph of the road network requires less computations on the GPU than erosion does, but since a lot of data is transferred to and from the graphics card the process is slower than one might expect. It would be advantageous if the road graph could by built on the GPU so that memory need not be transferred, but current GPU hardware is ill suited for the task. The measurement was performed in the same manner as measurement of erosion time and is displayed in table 4.1.2. The network generated in the tests were built between 100 points of interest in each case. In the common case the algorithm for generating a road processes far less than every element in the world, and as such it makes sense that the time increases sub-linearly with regard to world-size.

**Scalability** The algorithm has some potential for being further parallelized. If two roads are to be generated and they are guaranteed to not intersect, their cost...
4.1 Performance

Figure 4.2

(a) A long and straight road
(b) A curving road

<table>
<thead>
<tr>
<th>World size</th>
<th>Time / s</th>
<th>Expected Time / s</th>
</tr>
</thead>
<tbody>
<tr>
<td>655 km²</td>
<td>28.6</td>
<td>-</td>
</tr>
<tr>
<td>2621 km²</td>
<td>82.7</td>
<td>28.6 · 4 = 114.4</td>
</tr>
<tr>
<td>10485 km²</td>
<td>292.4</td>
<td>28.6 · 16 = 457.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>82.7 · 4 = 330.8</td>
</tr>
</tbody>
</table>

Table 4.1.2: Time to generate a road network in the world

Maps could be computed in parallel. Some care might be needed when updating the graph-structure representing the roads. The time required for that is however much lower than that of generating and transferring the cost maps, and may possibly be serialized without any noticeable performance penalty.

4.1.3 Tile Generation

The memory requirements for generating tiles is deterministic, as a buffer is allocated to hold all the tiles within a certain area before the buffer is filled tile by tile. In this implementation, the generation of tiles is performed on the CPU. Currently separate blocks of tiles are generated in parallel on different threads on the CPU, but the generation of blocks could be further accelerated on the GPU.

A downside to representing the world with tiles is that they do not hold any information about the surface normal. Since each face generated is axis-aligned, only 6 different normals are ever present at most. In the case of tiles generated on a slope, the loss of proper shading makes it hard to appreciate the depth and nature of the slope, as can be seen in figure 4.3. A possible solution would be to use a surface extracting technique instead, such as Marching Cubes or Dual Contouring[11]. These algorithms directly produce the polygons required and would have the advantage of only doing material look-ups for positions at the surface, as opposed to the current implementation which does that for every solid tile in the whole volume. Another advantage would be the possibility of producing good Level of Detail-meshes for distant surfaces; The current approach needs to
Fig. 4.3: The side of an undulating hill. The loss of normal information removes the sense of depth that proper shadowing provides.

generate or load tiles before surfaces are made, and at a distance the number of tiles required for this is incredibly high.

**Speed** The speed of tile generation for a block is pretty consistent. The main factor which determines the speed is how many air-tiles the block will contain, as these do not need to perform a material-look-up in the generated stratas.

**Scalability** As the tiles and blocks of tiles can be generated in parallel, there is some room for improvement by generating them on the GPU or with help of networked computers. An implementation in OpenCL would be advisable, as the same code would be able to run on both GPU and CPU without major modifications.

### 4.2 Overall Evaluation

It is somewhat hard to evaluate the results of this project, even in a subjective way. In order to make it easier to determine if this project was successful or not, a set of questions were posed to be answered in this section

- Is the world good enough for use in a product?
4.2 Overall Evaluation

- Is the world realistic enough?
- Is the world varied enough?
- Is the generation quick enough?
- Is the rendering fast enough?

**Is the world good enough for use in a product?** The produced world contains much that is good, but as can be seen in figure 4.5 the lack of vegetation makes the world look barren and empty in its current state. The world needs to be populated with more local features in order to make it more appealing. As the world is presented in a tile-based voxel format which makes it unpleasant to travel vertically, something you do a lot in mountainous terrain. The tiles also present a problem in representing small features such as rocks and grass. Distant terrain is not properly rendered (fig 4.4) as the problem of tiled *Level of Detail* was not part of the project. The generated roads span the world in an interesting way, but due to the behaviour of the splines used to brush them into the world they sometimes have undesirable behaviour such as going straight up steep inclines (where a serpentine road would fit better) or *overshooting* in bends. Both of these flaws are illustrated in figure 4.6. These shortcomings make it unlikely that the generation and rendering would be used as-is without further enhancement in a product. The produced world could be used for other purposes than exploring in first-person view in the implemented engine, such as a strategy game or a management game where these shortcomings are not as much of a problem.

In short the generated world is good enough for use in a product, but the current rendering of the world is not.

**Is the world realistic enough?** Ignoring the fact that the world is a square of approximately 100 square kilometers surrounded by invisible walls, the world is somewhat realistic. It does not have any faults lines or other results of tectonic activity which is quite unrealistic, so finding tilted or folded stratas in the world is very unlikely.

As a result of the erosion process just stopping abruptly, large bodies of water may become *stuck* on a slope as can be seen in figure 4.7. Had the erosion continued for a while, the water would have settled further downstream.

A related issue is the sometimes seemingly unexplainable sources of water, as water can seem to spring up from nowhere. In the model, rain deposits a uniform amount of water everywhere on the map, but this is not made apparent with weather effects in the exploration mode.

**Is the world varied enough?** The world is certainly varied. In figure 4.8 which represents a generated world of only 25² kilometers we can see larger bodies of water with relatively flat land next to it, lots of small and large streams running from the mountains. We can see both large and small ridge formations with accompanying valleys. In a full-sized world, the diversity is even greater as there is a larger area which is not affected by the boundary conditions (Which can be
Fig. 4.4: This rendering does not make justice to the distant terrain due to lack of proper LOD system.

Fig. 4.5: The lack of vegetation makes the world look somewhat barren. The effect is more apparent as one moves around in the world.
Fig. 4.6: A road on a very steep hill, with a small overshoot instead of a smooth bend

Fig. 4.7: A huge body of water frozen on a slope as the erosion process stopped before the water reached a place to settle.
Fig. 4.8: The world contains many distinct landmarks such as lakes, flatland, valleys with rivers and mountain ridges

seen clearly in the lower right corner if figure 4.8).

The worlds typically don't contain any iconic mountain peaks or incredibly sudden and steep drops which one may expect from mountainous areas, as these are quickly eroded away by water and talus movement.

**Is the generation quick enough?** The pre-processing of a world is fast enough that an end-user could generate a world for exploration if so desired, and offline generation by a producer is fast enough to allow quick iteration cycles for adjusting world parameters. The filling of tiles in volumes and surface extraction is also fast enough to allow seamless exploration of the world as long as the explorers speed is limited to realistic values. If the user teleports or loads an entire new area there is a slight load time, but well within reasonable limits.

**Is the rendering fast enough?** The rendering of the generated terrain is certainly fast enough. The rendering of polygons is extremely fast on modern hardware, so any bottleneck lies in generating the polygons to use.
In this report we have presented a project that generates mountainous worlds by combining procedural techniques and geological processes. First a height-field is generated with procedural noise and used as input for an erosion process. Erosion occurs on two layers of material with different properties. To simulate water in the height-fields the pipe-model is used. The water erodes the height-field and transports eroded sediment. Thermal erosion collapses sharp inclines and banks. A road network is grown in the world to connect places of interest by generating cost maps. The world consists of different stratas of material, which are exposed by the erosion. The world is visualized by generating chunks of tiles from which a surface is extracted and rendered. The methods are accelerated by executing data-parallel computations on the GPU where possible. The generated worlds are visually pleasing, and the diversity makes it interesting to explore.

5.1 Future work

As parts of the project required more attention than planned, features such as forests and vegetation where dropped. Information produced by the erosion, such as running water and where soil has accumulated, could be used to produce realistic and interesting spread of flora.

The information about stratas at different depths could be exploited in the erosion process, by using local material properties when dissolving and transporting material. Problems with aliasing when determining what material to use needs to be solved before this will produce any worthwhile results.

A hybrid solution with both the pipe-model and particles would be interesting to consider, as it might produce more realistic rivers. Representing some of the
dissolved material as particles with known material and weight might also yield interesting results, as they might settle in slow regions or help break down things they collide with.

Since the distance between sample points in the generated height-fields is quite large, local detail from erosion is sadly lacking. Adding a layer of Erosive Noise[3] to the surface would help alleviate this issue. The two methods would complement each other as macroscopic structure is missing in erosive noise.

The current method of generating a volume of tiles and then producing a polygon surface from them removes normal information from the surface. Switching to another surface extraction technique would retain the information and might even accelerate the process.

To circumvent the size-limitation of the world imposed by the memory of the GPU, the erosion could be ported to OpenCL and executed on the CPU at the cost of speed. Doing this would also make it easier to transition to a model which would allow distributed computation on networked computers.


[9] Christophe Riccio Graham Sellers Bruce Merry John Kessenich Jeff Bolz,


Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns detlösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Pierre LeMoine