Generating Terrain Models using the Stream Power Law

Liam Bell

Bachelor of Science in Computer Science
The University of Bath
2024

Generating Terrain Models using the Stream Power Law

Submitted by: Liam Bell

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

The study of how landscapes form and evolve has been studied in the field of geomorphology for many years and has applications in environmental planning, entertainment and beyond. In this project, a software system was implemented which applies the stream power law from geomorphology to a graph-based terrain model. The system produces geologically consistent and realistic terrain models through the simulation of fluvial erosion. The realism of the produced terrains was then evaluated. The effect of the parameters of the stream power law on terrain generation were also analysed and the performance of the system over different parameters was evaluated.

Contents

Acknowledgements

I would like to thank to my supervisor, Mac, for his support and belief in my project.

Chapter 1

Introduction

Terrain modelling is the representation of the topographical and geological features of natural landscapes, modelling the landforms and features of physical terrains.

1.1 Applications

Digital terrain models are a key part of any domain involving the natural world and have a broad range of applications in geomorphology, environmental planning, entertainment and beyond. Geomorphology is the study of how landscapes are formed and change over time. Digital terrain modelling can be used to predict how landscapes will change through computer simulation. In the domain of environmental planning, predicting the changing landscape is key in urban planning and disaster management. In entertainment, films are often set in cinematographically difficult settings or even in fictional worlds. Digital terrain modelling can be used to solve this problem, without the requirement to film in physical locations, by synthesising fictional terrains using computers.

1.2 Project Aim

The aim of this project is to create a software tool which allows users to author and generate realistic synthesised digital terrain models.

1.3 Project Objectives

1. Research existing algorithms used for procedural terrain generation.
2. Research existing algorithms for geomorphological simulation.
3. Choose a state-of-the-art algorithm to implement based on a review of the literature.
4. Implement and extend the algorithm to create a software system which meets the project aim.
5. Evaluate the system on the variety of landforms it can generate, the realism of the terrains, the system's performance and the system's degree of authoring.
6. Identify the limitations of the system.

Chapter 2

Literature, Technology and Data Survey

A literature search was undertaken on the area of digital terrain modelling, looking at different methods for terrain representation, procedural terrain generation, simulation-based generation methods and example-based methods. The following is a review of the work found.

2.1 Terrain Representation

In this section, an overview of the different ways digital terrain models can be represented is presented.

2.1.1 Elevation Models

Terrain can be modelled as having an elevation described by a function $h : \mathbb{R}^2 \to \mathbb{R}$, which describes the altitude of a point on the terrain (Galin et al., 2019). This could be represented as an exact closed form mathematical expression, but in practice it is discretised.

The ubiquitous method for representing discrete elevation models are heightfields, also known as Digital Elevation Models (DEMs). Heightfields are a 2D array of altitudes representing the height of the terrain at points on a regular grid. The terrain surface can be constructed by interpolation of grid points to form a mesh.

2.1.2 Volumetric Models

Elevation Models can only describe the surface of a terrain, and do not represent the internal structure of the terrain or allow for the modelling of caves and overhangs. Volumetric models can be described by a function $\mu : \mathbb{R}^3 \to M$, where M represents the material at that point in space. This allows for complete modelling of the terrain in three dimensions.

The most common discretisation of volumetric models are voxels. Voxels are analogous to 2D pixels, but for 3D. Space is partitioned into a regular grid and each cell in the array given a material. It has a much increased cubic space complexity, but allows for more realistic modelling than heightfields, as subsurface structures can be modelled, such as caves. A voxel representation is used by Ito et al. (2003) to model rocky terrains with cracks or joints between

the rocks. Beneš et al. (2006) uses a voxel representation to simulate hydraulic erosion in three dimensions.

2.1.3 Layered Representation

Layered representations are similar to heightfields, but are not limited to a single height value for each point; it can have many values which represent layers of different materials of different depths. (Musgrave, Kolb and Mace, 1989) use material layers to model different sediments, but the idea was properly introduced in (Beneš and Forsbach, 2001), and is a compromise between heightfields and voxels. Using voxels requires lots of data storage, but using a normal heightfield means information about material below the surface is not represented. Peytavie et al. (2009) presents a framework for modelling landforms such as caves, arches and overhangs using a representation using layers of material, which includes air layers.

2.2 Procedural Methods

Procedural terrain generation methods aim to reproduce the effects of geomorphological phenomena, without simulating the real processes that create the terrain. They instead use procedural algorithms to create terrain models, which can be large scale terrain models, or models of specific landforms.

Many terrains have fractal properties at different levels of detail, for example, coastlines or eroded mountains looking self-similar. One of the main procedural techniques for generating terrain is using fractional Brownian motion (fBm) (Mandelbrot and Ness, 1968). This can be implemented using Perlin noise (Perlin, 1985), by combining noise functions of varying frequencies and amplitudes.

A popular procedural method for producing terrain models is subdivision schemes. The midpoint displacement subdivision algorithm introduced by Fournier, Fussell and Carpenter (1982) takes a heightfield and creates terrain by vertically displacing a point between two points by a random amount from their average. Variations on this have been explored including the square-square scheme introduced by Miller (1986).

Another procedural algorithm used for terrain generation is faulting, introduced by Mandelbrot (1982) and extended in (Voss, 1991) and (Ebert et al., 2003). The algorithm works by taking a random line on a heightfield and raising up the points on one side and lowering the points on the other to reproduce the effect of the cracking of the Earth's crust.

Stachniak and Stuerzlinger (2005) present an algorithm which deforms fractal terrain using certain constraints. For example, a terrain can be deformed to fit a specified path.

A method for generating river networks using the midpoint-displacement algorithm was introduced by Kelley, Malin and Nielson (1988) by modelling the formation of drainage networks formed by streams and their tributaries. The river network can then be used to generate terrain. A more sophisticated approach to creating river networks by Génevaux et al. (2013) uses a grammar which is constrained by the branching properties of the graph of the river network. This is used to construct watersheds and classify the rivers using the Rosgen classification (Rosgen, 1994). A different approach is taken by Peytavie et al. (2019), where instead of generating terrains from water courses, the water courses are identified from existing terrain and then are used to carve a river network into the terrain.

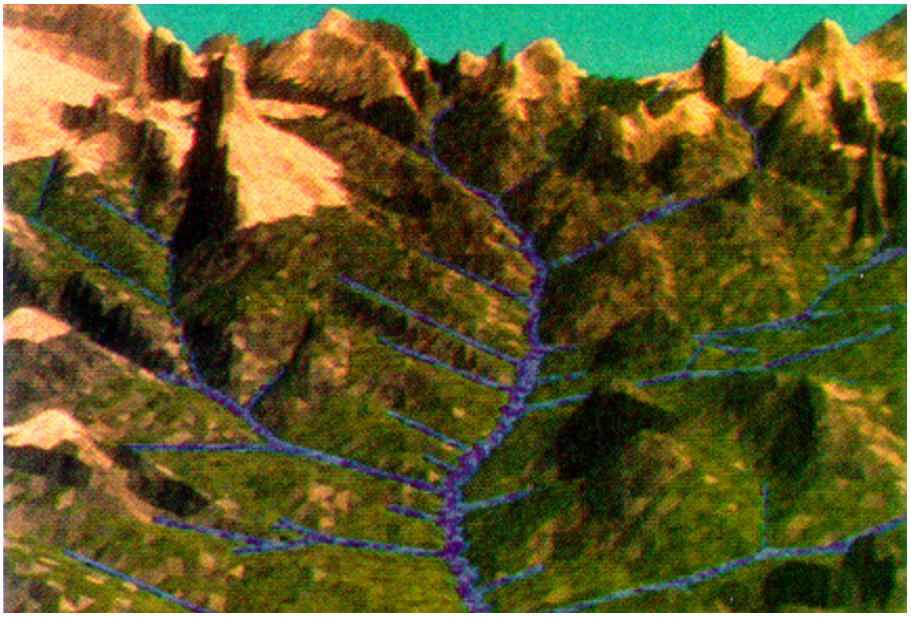


Figure 2.1: Example of a terrain generated by the method described in (Kelley, Malin and Nielson, 1988).

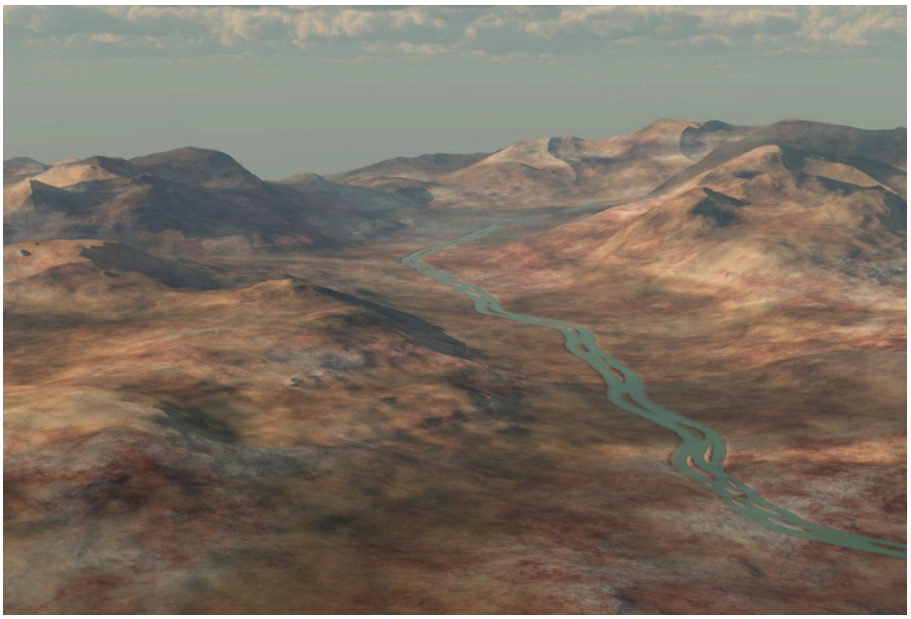


Figure 2.2: Example of a terrain generated by the method described in (Génevaux et al., 2013).

2.3 Simulation Methods

In contrast to procedural techniques, simulation methods model not just the effects of geomorphological processes (terrains), but the causes too. The terrain model produced is emergent from the simulation.

Most of the terrain simulation methods explored in computer graphics are based on erosion. Erosion is the action on a surface which:

1. *Detaches* material from the surface.
2. *Transports* the material.
3. *Deposits* the material at another location.

The creation of terrain models from erosion simulation models is achieved through the computation of the changing boundary between the terrain model and the erosion process (Galin et al., 2019). This is a function of many different characteristics of the environment such as slope, elevation, water flow, environmental factors, etc.

2.3.1 Thermal Erosion

Thermal erosion is caused by expansion of water in cracks inside rocks. This can cause material to break off and fall, eroding the terrain. The simulation of thermal erosion was introduced by Musgrave, Kolb and Mace (1989) The transportation of the material is dependent on the talus angle, which is is the angle where the deposited material has sufficient friction to be stable Galin et al. (2019). The thermal erosion process can be simulated using relaxation, iteratively moving some amount of material to be distributed among neighbouring areas. The specific landforms produced by thermal erosion are typically scree slopes under crags. Thermal erosion was also used as a technique to model the table mountain shapes found in mesas in (Beneš and Arriaga, 2005) using a terrain model composed of a hard and soft material.

2.3.2 Aeolian Erosion

Aeolian erosion is caused by wind. It is common is deserts where the wind erodes sand which further erode material by abrasion. The simulation of Aeolian erosion is explored in (Onoue and Nishita, 2000), where two models are presented: the creation of sand dunes and the rendering of wind-ripples on the dunes using bump mapping.

2.3.3 Hydraulic Erosion

Hydraulic erosion is the phenomenon caused by water movements against bedrock .This includes processes such as abrasion (material being transported wears away at a surface over time), corrasion (abrasion of the river bed) and saltation (material is transported but not suspended).

Hydraulic erosion methods can be classified by how the erosion model is represented and can be classed into Eulerian, Lagrangian and graph-based approaches.

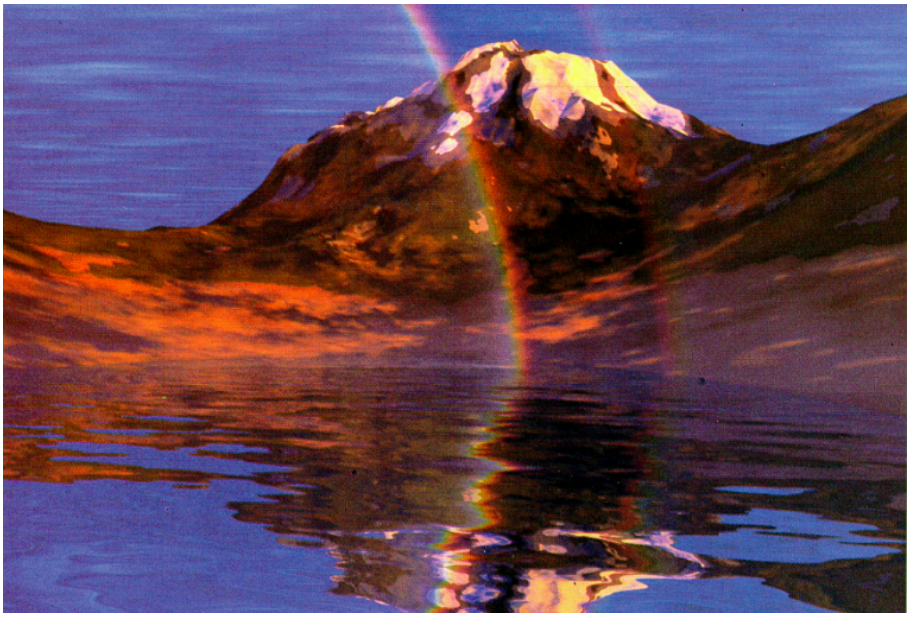


Figure 2.3: Example of a terrain generated by the method described in (Musgrave, Kolb and Mace, 1989).

Eulerian Approaches

Eulerian approaches use a discrete grid-based model, simulating the amount, pressure and velocity of water in each cell. Musgrave, Kolb and Mace (1989) present one of the first erosion simulations using an Eulerian approach to simulate hydraulic erosion on a heightfield. Their simulation deposits water on the heightfield, detaches some material and then the water and any suspended sediment material is transported to lower, neighbouring cells in the grid. The amount of erosion which occurs is a function of water volume, amount of sediment and the gradient of the terrain.

Roudier, Peroche and Perrin (1993) extend on this, presenting an hydraulic erosion model which also considers the properties of the strata of the bedrock, such as rock softness, permeability and vegetation cover.

Nagashima (1998) proposes another approach which takes as input, the initial water courses of the terrain. These are created using the midpoint displacement method (Fournier, Fussell and Carpenter, 1982). He focuses on the generation of valleys from streams, only simulating erosion on rivers with rapid flows.

Beneš and Forsbach (2002) introduces a new algorithm for hydraulic erosion, focusing on visual plausibility over physical correctness. The layered data structure introduced in (Beneš and Forsbach, 2001) is used to represent the terrain model and a diffusion model is used to simulate water moving to lower neighbouring cells. The sediment in the water cannot exceed a certain saturation level, and this causes the deposition of material, due to water evaporation.

Neidhold, Wacker and Deussen (2005) presents a new method for simulating erosion through fluid simulation that can run at iterative rates (in real time). More accurate water transport is simulated whilst maintaining interactivity.

All previous erosion simulations used a height map or a layered representation, but Beneš et al. (2006) presents a hydraulic erosion simulation which is modelled fully in three dimensions (3D).

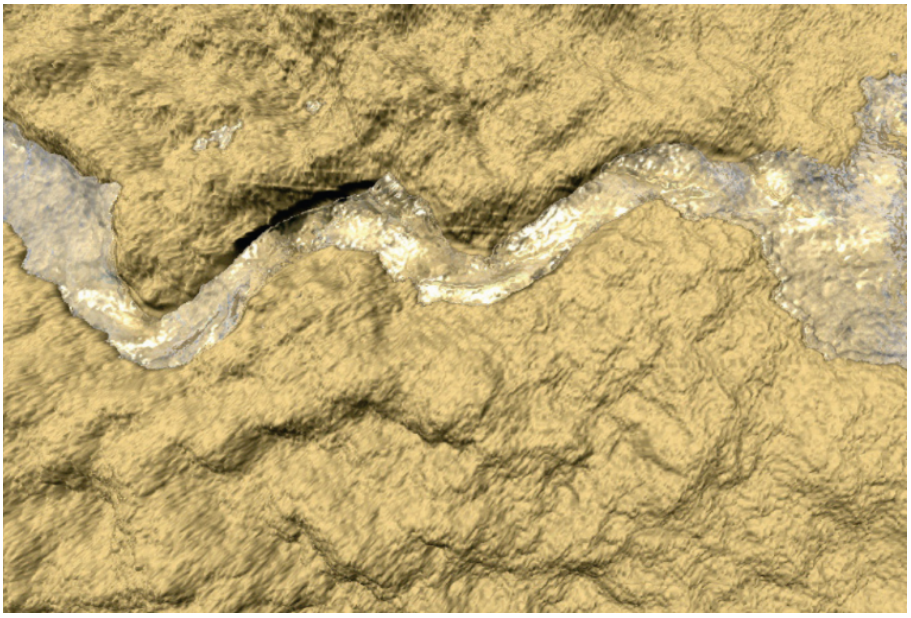


Figure 2.4: Example of a terrain generated by the method described in (Krištof et al., 2009).

The model they present works with a voxel-based representation and can simulate phenomena such as receding waterfalls, meanders and springs. The algorithm involves approximately solving the Navier-Stokes equations to compute the pressure and velocity field for each voxel. Then hydraulic erosion is performed. The large downside of this method is it is too computationally expensive for real-time applications.

Beneš (2007) presents a real-time hydraulic erosion simulation which models dissolved material as a fluid with a higher viscosity than water. The terrain uses a heightfield representation and uses the shallow water equation as a simplification of the Navier-Stokes equations. When the water evaporates, the dissolved material is deposited.

Lagrangian Approaches

Lagrangian approaches model hydraulic erosion using particle simulation.

Chiba, Muraoka and Fujita (1998) use velocity fields of water particles to erode a heightfield. The velocity field represents the motion of many independent water particles. The simulated erosion occurs from the effect of the erosive forces from the surface runoff of water. However, interactions between the particles are not considered. Sutherland and Keyser (2006) extends this, adding the concept of pooling, where slow or old particles are pooled into a water pool, allowing deletion of particles to save computation.

Krištof et al. (2009) presents an approach to simulating hydraulic erosion using Smoothed Particle Hydrodynamics (SPH) in 3D at interactive rates. The approach uses approximations to the Navier Stokes equations to simulate the fluid dynamics. The erosion of the terrain is achieved through using boundary particles which handle the interactions of detachment and deposition between the particle-based fluid and the terrain model.

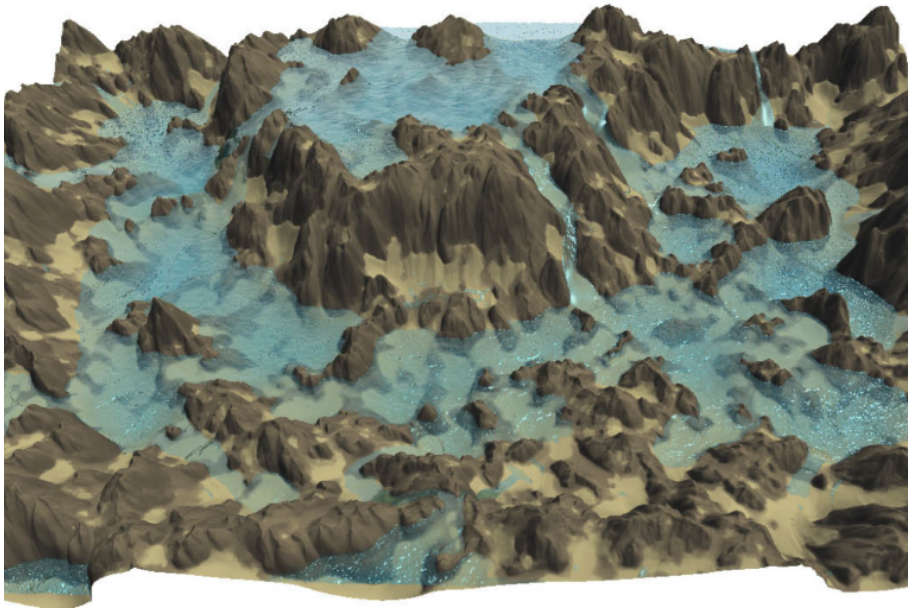


Figure 2.5: Example of a terrain generated by the method described in (Šťava et al., 2008).

Parallel Implementations on GPU

Hydraulic erosion is a computationally expensive process, meaning many algorithms are slow, or only run on limited terrain sizes. One approach to solving this problem has been to parallelise the simulation process, typically using General-Purpose Graphics Processing Units (GPGPUs). Mei, Decaudin and Hu (2007) present a method of simulated hydraulic erosion in parallel using velocity fields. The velocity field is updated using a shallow water model adapted from (O'Brien and Hodgins, 1995). In the same time, Anh, Sourin and Aswani (2007) also proposed a implementation on the GPU based on the method presented in (Neidhold, Wacker and Deussen, 2005). Combining methods from (Mei, Decaudin and Hu, 2007), (Beneš, 2007) (O'Brien and Hodgins, 1995), another parallel GPU implementation is proposed by Šťava et al. (2008) which runs of interactive rates.

2.3.4 Tectonic Uplift and Fluvial Erosion

Interactions between tectonic plates causes terrains to fold, which create mountain ranges. This happens through a phenomenon called uplift, is the raising of terrain vertically upwards, working against the processes of erosion. Michel, Emilien and Cani (2015) introduce a method for procedurally generating a fold map from a user inputted vector map. The tectonic plates are approximated from the peaks on the map and terrain folds are generated, from which a heightfield is formed.

A method introduced into computer graphics by Cordonnier et al. (2016) allows for the generation of large scale terrains using simulated tectonic uplift and fluvial (hydraulic) erosion. Streams are modelled as a set of trees covering a planar graph over the terrain. The combination of these trees with a user-specified uplift map is used to model the combination of uplift and erosion using the stream power equation Whipple and Tucker (1999).

This is extended by Cordonnier et al. (2018) who present a method to generate an uplift map from the simulation of tectonic plates. The authoring is achieved through defining the movement of tectonic plates through user gestures, allowing for real-time authoring of the

Figure 2.6: Example of a terrain generated by the method described in (Cordonnier et al., 2016).

terrain. The Earth's crust is modelled as an incompressible viscous material which is used to generate an uplift field from the folding processes and then fluvial erosion is simulated considering the different strata in the rock.

Schott et al. (2023) also combines the stream power equation with uplift modelling to create a tool which facilitates interactive large scale terrain authoring. The interactivity comes from the use a fast parallel drainage area approximation.

2.3.5 Glacial Erosion

Certain landforms have not been modelled through computer graphics simulation such as U-shaped valleys, hanging valleys and fjords, which are the effects of glacial erosion. In a recent breakthrough by Cordonnier et al. (2023), a solution which simulates the formation of glaciers allows the generation of such landforms. The simulation uses a deep learning based estimation for ice flows. Glacial erosion is combined with hydraulic (fluvial) erosion to model the combination these two process have on generating realistic terrain which has undergone ice ages.

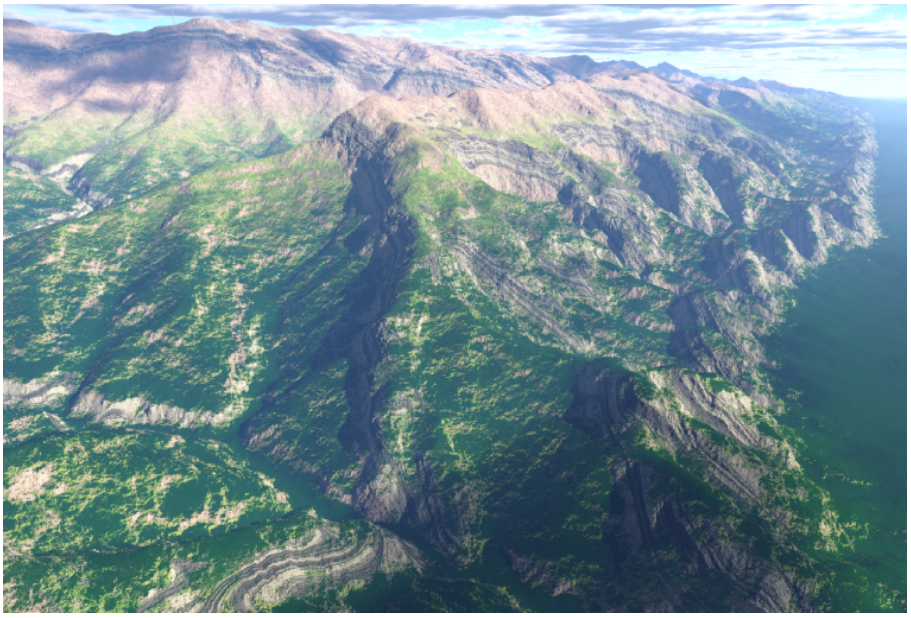


Figure 2.7: Example of a terrain generated by the method described in (Cordonnier et al., 2018).

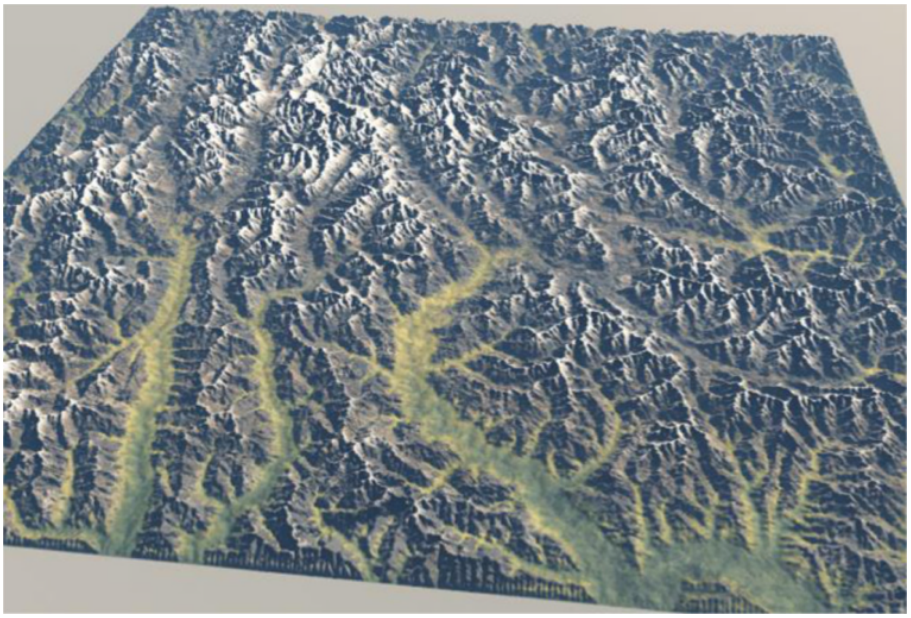


Figure 2.8: Example of a terrain generated by the method described in (Schott et al., 2023).

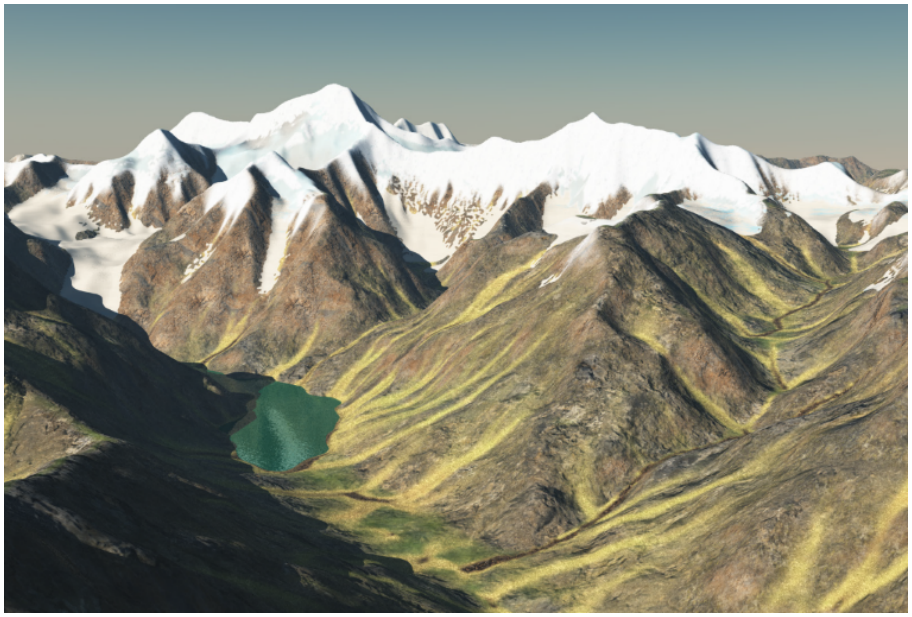


Figure 2.9: Example of a terrain generated by the method described in (Cordonnier et al., 2023).

2.4 Example-based Methods

Example-based methods use and transform real-world data to create digital terrain models. New terrain model are created by combining data from existing terrains. An advantage of example based terrain synthesis is that as the method is based on real data, the output should ideally be highly realistic. This all relies on the quality of the input data.

2.4.1 Texture Synthesis

Many approaches to terrain generation with example based techniques use a method called texture synthesis, which is takes input images and creates an output image of a similar kind (Galin et al., 2019). A heightfield terrain is similar to a grey scale image so texture synthesis techniques can be applied, such as in (Zhou et al., 2007). They include the ability for a user to sketch a feature map that specifies where terrain features occur in the resulting synthetic terrain and then samples patches from input terrain to create new terrain.

2.4.2 Machine Learning

(Guérin et al., 2017) uses Conditional Generative Adversarial Network models to created synthetic terrains. Data used to train the model is pairs of real-world terrain data and user sketches of that terrain. A trained model can then use new sketches of terrains, to synthesise new terrain models.

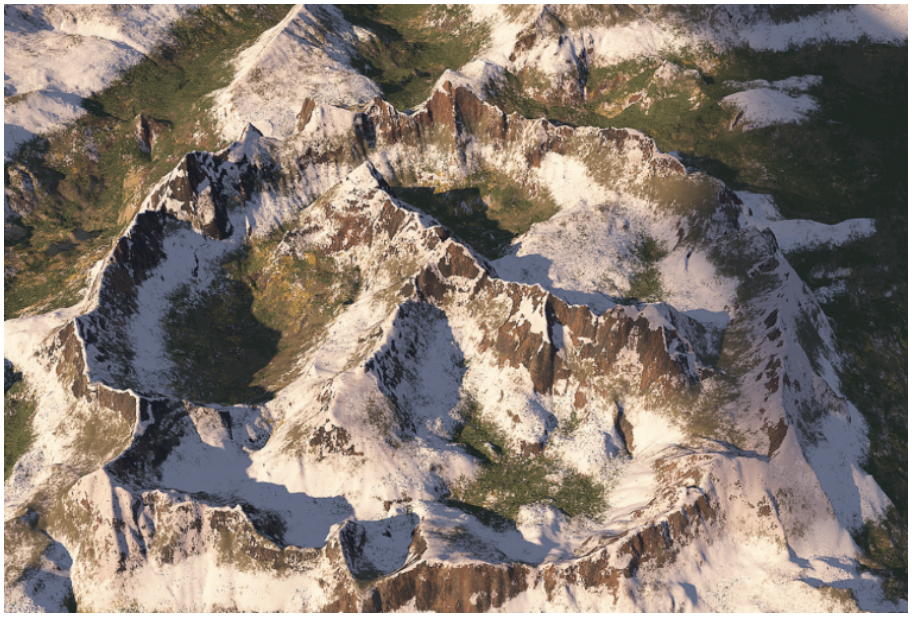


Figure 2.10: Example of a terrain generated by the method described in (Zhou et al., 2007).

2.5 Rendering Terrain Models

After modelling a terrain using any of the previously described techniques and terrain representations, it would be nice to actually see the terrain model generated. The terrain needs to be rendered. Often this process in split into two variations: one low-quality rendering for interactive visualisation; and a higher-quality rendering process for producing final images of terrain.

2.5.1 Basic Rendering

Rendering Heightfields

To render a heightfield, each point can be connected to form a three-dimensional polygonal mesh. This can then be rendered quickly using rasterisation for visualising the terrain.

Rendering Voxel-based Models

To render a volumetric terrain model which uses voxel, the marching cubes algorithm (Lorensen and Cline, 1987) can be used to convert the terrain model into a three-dimensional polygonal mesh, which can then be rendered.

2.5.2 High-Quality Rendering

For producing high-quality renders of terrain, more computationally expensive rendering techniques can be used as interactivity is not the goal, but rather the image quality. Ray tracing is an obvious choice as accurate lighting, reflections, refractions and shadows can be calculated. Extra detail can also be added to the model to improve the realism of the terrain, such as water and vegetation.

2.6 Programming Languages

The implementation of the system should be fast, so using C/C++ would be suitable as a good compiler can produce very efficient code, which is important for running geological simulations as quickly as possible.

2.7 Rendering Tools

For rendering the high quality images of any terrain models, a rendering engine could be used. The Terragen™ software might be a good choice as it is made specifically for rendering terrains.

Chapter 3

Methodology

3.1 Choice of Model

During the literature survey, the area of research was narrowed down to simulation methods. They are a suitable and interesting method for study. They have the potential to produce more realistic terrains than procedural methods due to their grounding in geology. Example-based methods can be very realistic but simulation methods were preferred due to personal interest in the physical geological processes which form terrains.

However, a significant disadvantage of simulation methods, compared to procedural methods is the large computation time it takes to generate terrains. This is especially true for Eulerian and Lagrangian methods, which makes generating large terrains using these methods computationally unfeasible.

Generating large-scale terrain models was an area of interest, but Eulerian and Lagrangian methods, which are already have large time complexities, are not feasible for generation of large terrains. The other main simulation approach is iteratively applying geomorphological equations to generate geologically consistent terrain models. These methods allow the generation of large, but also geologically consistent terrains which was important as the project aims to generate realistic terrains. Within this area, the project was now focused on the generation of large-scale terrains through simulated tectonic uplift and fluvial (hydraulic) erosion.

3.2 Geological Background

Real-world terrain is the result of the actions from many interconnected geological processes such as erosion and tectonic activity, as well as chemical and biological weathering processes.

3.2.1 Tectonic Uplift

Tectonic uplift is the rate at which the Earth's surface rises due to subsurface tectonic activity. When tectonic plates collide, the crust compresses, which forces the terrain to be pushed upwards, eventually resulting in the formation of mountains, after millions of years, along the boundary of the plates.

3.2.2 Fluvial Erosion

Fluvial erosion is an hydraulic erosion process where material from the underlying terrain is detached and transported by streams and rivers. Many terrains are formed through the competing interaction of the rising of the terrain due to tectonic uplift, against the eroding of the terrain due to fluvial erosion. The rate of change of the terrain height h $[L]$ at a point with uplift u $[LT^{-1}]$ can be simply modelled as so:

$$\frac{dh}{dt} = u - \epsilon \tag{3.1}$$

The fluvial erosion rate ϵ $[LT^{-1}]$ at a point in a stream can be modelled by using geological model called the stream-power erosion model (Whipple and Tucker, 1999):

$$\epsilon = kA^m S^n \tag{3.2}$$

where A $[L^2]$ is the drainage area of the point, S is the stream's topographic gradient, m and n are positive exponents and k $[L^{1-2m}T^{-1}]$ is the erosion constant.

The exponents m and n depend on many factors, such as rock strength and climate. The exact values vary depending on region, but the ratio is normally $m/n \approx 0.5$ (Whipple and Tucker, 1999). A survey (Lague, 2014) discusses the knowledge of setting these parameters.

The drainage area, A is the planar area of the terrain which is upstream of the point. It is a good proxy for the volume of water flowing through that point of the stream, as any rainfall in a stream's drainage area will collect to that point.

3.3 Overview

The system developed for this project works by applying the stream power erosion law to a planar graph to create geologically coherent digital terrain models based on a user-defined uplift map. The approach taken is based on (Braun and Willett, 2013) and (Cordonnier et al., 2016). A brief overview is shown here, with detail of the algorithm explained in the following sections.

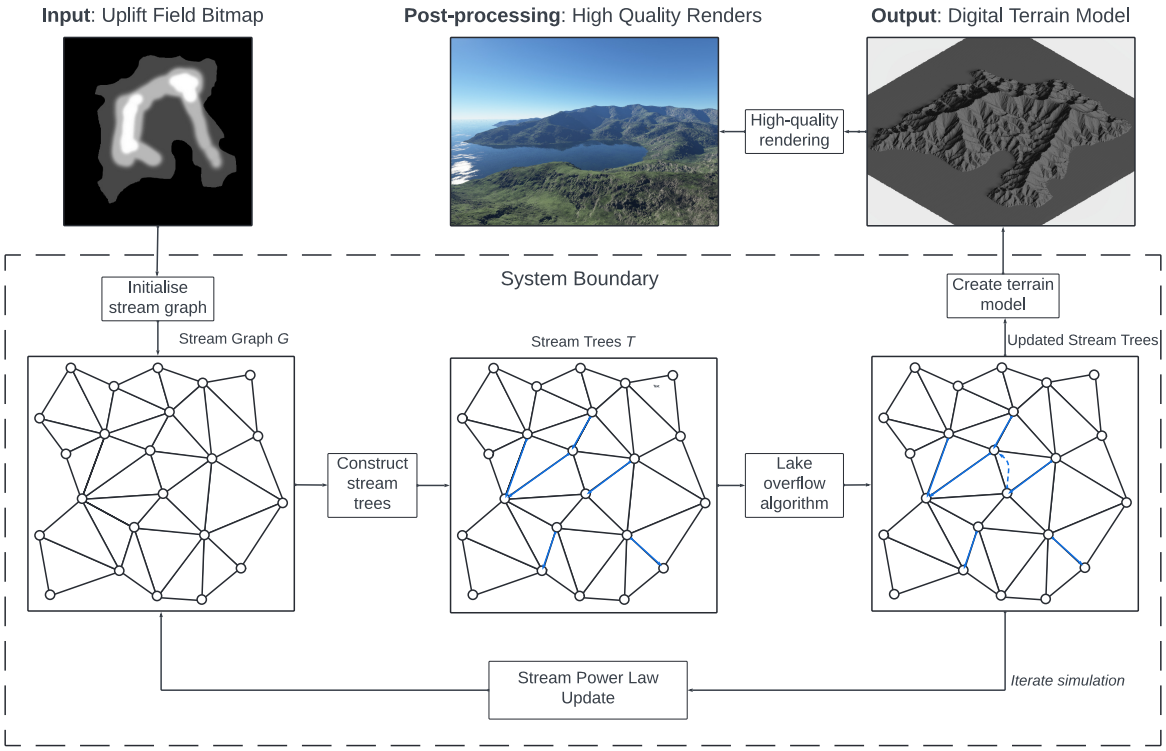


Figure 3.1: Overview of the method.

3.3.1 Input

:

- Uplift field $U : \mathbb{R}^2 \to \mathbb{R}$

3.3.2 Output

:

- Digital elevation model $M : \mathbb{R}^2 \to \mathbb{R}$

3.3.3 Stream Graph Initialisation

1. Create a random planar graph, $G = (V, E)$ called the stream graph.
2. Positions p_v of vertices $v \in V$ are randomly uniformly distributed across the plane of the domain.
3. The set of edges E is defined as a Delaunay triangulation of V.
4. Each vertex $v \in V$ is assigned an uplift value, u_v which is sampled from the uplift field U.
5. For each vertex $v \in V$ its, elevation, h_v, initially set to zero.
6. For each vertex $v \in V$, the immediate drainage area, a_v is calculated

3.3.4 Erosion Simulation on Stream Graph

The main simulation loop below is iterated a maximum number of time steps or may stop at an earlier time step i if convergence threshold is met: $\forall v \in V(|h_v^{i+1} - h_v^i| < c)$ where c is a set constant percentage.

1. A covering of G is made with a set T of trees called stream trees. A edge in a stream tree will connect a vertex with a higher elevation from a neighbouring vertex in G with the lowest elevation. The root of each tree, either represents an outflow at the domain's boundary, or a lake.
2. T is modified by connecting some stream trees to others to model the overflow of lakes into other stream.
3. The drainage area A, for each vertex $v \in V$ with children $C(v)$, is calculated.
4. The topographic slope, S_v, for each vertex $v \in V$, with parent vertex w is calculated.
5. The updated elevation h_v^i for each vertex $v \in V$ with parent vertex w after time step δt is updated using the stream power erosion law.

3.4 Stream Graph Initialisation

3.4.1 Vertex Creation

First, a random undirected geometric graph G, called the stream graph is created. G is geometric as it has vertices V which have a position in 3D space. Positions of vertices are separated into their (x, y) positions in the horizontal plane, which are fixed at the initialisation of G, and elevation values which vary in the positive direction perpendicular to this plane.

The horizontal positions p_v of vertices $v \in V$ are randomly distributed using Poisson disk sampling. This ensures that the distance between points is always greater than a specified value.

3.4.2 Edge Creation

The set of edges E is defined as a Delaunay triangulation of V. The edges represent potential flows for stream trees in the stream graph. A triangulation subdivides the convex hull of the points in the horizontal plane of vertices in V into triangles. A Delaunay triangulation is a triangulation for which every triangle's circumcircle does not contain any points in the triangulation. One reason a Delaunay triangulation was used is because it avoids creating so-called sliver triangles with small angles, ensuring the sides of the triangles have similar lengths.

The combination of using Poisson sampling to generate the positions of the vertices and the edges being a Delaunay triangulation means that the lengths of the edges in the stream graph are of similar lengths. This is important as the resolution, or level of detail of the terrain can be controlled directly by changing the number of vertices in the graph (Cordonnier et al., 2016).

3.4.3 Vertex Properties

Each vertex $v \in V$ is assigned an uplift value, u_v which is sampled from the uplift field U. The uplift for a particular vertex is constant for the entire process. For each vertex $v \in V$ its, elevation, h_v, is initially set to zero meaning no stream trees will be formed on the first iteration, until the varying uplift values are added to the heights.

3.4.4 Immediate Drainage Area

The stream power equation is a function of the drainage area of a point in a stream. To model this, the drainage area for a stream tree is calculated as the area of the points in the domain which are 'closer' to that stream tree than any other tree. In this initialisation stage, only the immediate drainage area for a particular vertex is considered, as the stream trees are yet to be constructed. A Voronoi diagram (Figure 3.2) is created as the dual graph of G; the circumcentres of the Delaunay triangles are the vertices of the Voronoi diagram. This is used to calculate the immediate drainage area, a_v for a vertex $v \in V$ as the area of the Voronoi cell corresponding to that vertex.

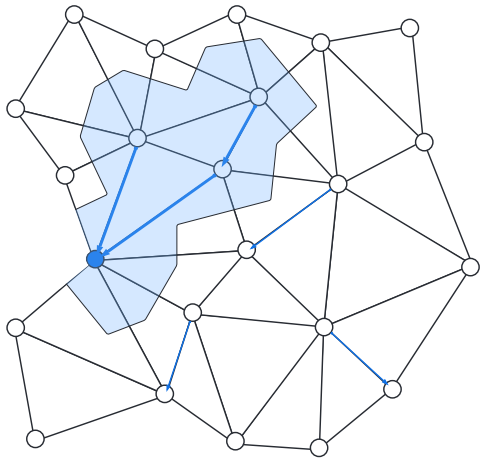


Figure 3.2: The Voronoi cells shaded are used to calculate the drainage area of the stream.

3.5 Construction of Stream Trees

A covering of G is made with a set of directed rooted trees (arborescence), T. A directed edge in a stream tree will connect a vertex with a higher elevation from a neighbouring vertex in G with the lowest elevation. These trees are the representation of a stream in the model. The root vertex of each tree, either represents an outflow at the domain's boundary, or a lake.

3.6 Lake Overflow

Simply using these stream trees will not always work, as vertices which represent lakes (root vertices of trees which are not outflows of the domain) have no way of ever connecting to the

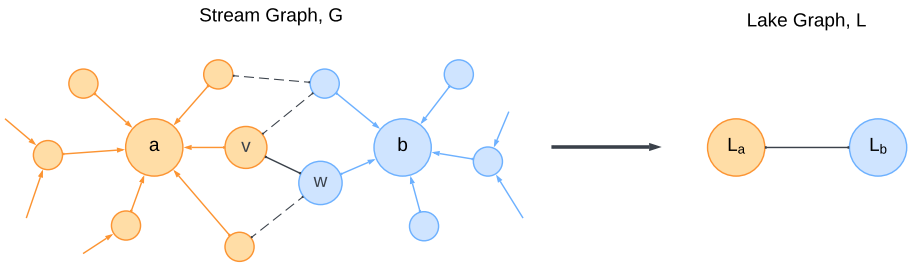


Figure 3.3: Creation of the lake graph from the stream graph.

wider stream network, leading to a very large number of lakes which can never connect. This means that the algorithm is likely to converge on a local minimum (Braun and Willett, 2013). To connect lakes together, the approach based on the solution from (Cordonnier et al., 2016) is used.

3.6.1 Creation of Lake Graph

A new graph L, called the lake graph is created. Every stream tree in T is represented by a single vertex in L. The lake a vertex v belongs to is identified by $L(v)$. A **pass** is defined at the lowest pair of neighbouring vertices $v, w \in G, (v, w) \in E$ which are not are not in the same stream tree $v \in T_1, w \in T_2, T_1 \neq T_2$.

(Cordonnier et al., 2016) defines the condition for a pass (v, w) between lakes l_1 and L_2 as:

$$\forall i, j,\ L(i) = l_1, L(j) = l_2\ (max(h_v, h_w) <= max(h_i, h_j))\ where\ max(x, y) = \begin{cases} x & x > y \\ y & otherwise \end{cases} \tag{3.3}$$

This 'pass' in the model represents a col between real mountains.

3.6.2 Creation of Lake Trees

A covering of the lake graph with lake trees is then created. Unlike the stream graph, lake vertices do not have a elevation, so the set of trees T_L is constructed using the following algorithm from (Cordonnier et al., 2016);

1. Initialise $T_L = \{l \in L, l\ is\ an\ outflow\}$
2. Create a set of candidate directed edges $(a, b) \notin T_L$ where $a \notin T_L$ and $b \in T_L$
3. Choose the candidate edge with the lowest pass height to add to to T_L, as well as the vertex a.
4. Remove unused candidate arcs from L.
5. Repeat steps 2-4 until no candidate edges remain.

3.6.3 Updating the Stream Trees

The set of stream trees T is modified by using the information from the set of lake trees T_L. A new edge $(v, w) \in E$ is added to T for each $(a, b) \in T_L$ where v is the root of the stream tree corresponding to lake a and w is the pass vertex for lake b. This guarantees that each $t \in T$ still retains it tree structure, but the number of trees with reduce. The modified trees will all now have root verticies at outflows from the domain, with no interior lakes, solving the issue of local minima.

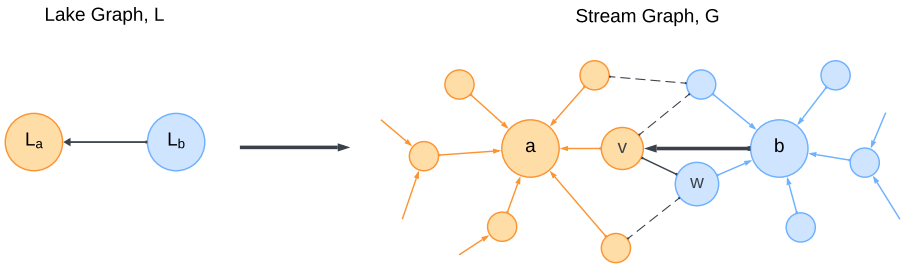


Figure 3.4: Adding new edges to the stream graph using pass information.

3.7 Applying the Stream Power Law

3.7.1 Calculating Drainage Area

The drainage area A, for each vertex $v \in V$ with children $C(v)$, is calculated as:

$$A_v = a_v + \sum_{u \in C(v)} A_u \tag{3.4}$$

As this is a recursive formula, each computation can be done from the leaves (the source) of the stream trees, down to the root of the tree (the stream outflow).

3.7.2 Calculating Topographic Slope

The topographic slope, S_v, for each vertex $v \in V$, with parent vertex w is calculated as:

$$S_v = \frac{h_v - h_w}{||p_v - p_w||} \tag{3.5}$$

3.7.3 Solving the Stream Power Equation

The updated elevation $h^{t+\delta}$ for each vertex $v \in V$ with parent vertex w after time step δt can be updated implicitly using the stream power erosion law (assuming $n = 1$):

$$h_v^{t+\delta} = \frac{h_v^t + \delta t(u_v + h_w^{t+\delta}(\frac{kA_v^m}{||p_v - p_w||}))}{1 + \delta t(\frac{kA_v^m}{||p_v - p_w||})} \tag{3.6}$$

3.8 Thermal Erosion

The stream power models stream incision well, but it has an effect of creating sharp peaks in areas of low drainage (Cordonnier et al., 2016). To correct for this, a model of thermal erosion from (Musgrave, Kolb and Mace, 1989) is used to limit steeper slopes. The term thermal erosion is used to describe the collection of erosive processes which cause steep hill slopes to erode material into the stream below. After the height of a vertex is updated from the stream power law, it is change from thermal erosion based on a specified talus angle α.

$$\frac{dh_v}{dt} = \begin{cases} -k(S_v - tan(\alpha) & S_v) > tan(\alpha) \\ 0 & otherwise. \end{cases} \tag{3.7}$$

3.9 Creation of Terrain Model from Stream Graph

The stream graph G can be used to create a digital elevation model $M : \mathbb{R}^2 \to \mathbb{R}$ using this method:

1. Create a heightfield initialised to zero.
2. For each vertex $v \in V$ apply a Gaussian filter to the heightfield centred at each vertex scaled by h_v.
3. Normalise the heightfield by the sum of the values the Gaussian filter kernels.

Chapter 4

Algorithms and Implementation

4.1 Overview

The system was implemented as a computer program developed in C++. Development initially started in C, and changed to C++ to utilise the higher-level features of C++ such as object-oriented programming. The machine used for development was a laptop computer running Windows 11. However, development was done on Ubuntu (version 20.04) by use of The Windows Subsystem for Linux. Code was compiled using gcc version 9.4.0.

Full code listings for the system can be found in Appendix C.

4.2 Implementing Auxiliary Functions

Before implementation of the main algorithm, some auxiliary functions were needed to perform tasks such as creating heightfields, outputting heightfields as images and meshes, and implementing noise generation functions.

4.2.1 Creating Heightfields

To represent heightfields in code, a 2D array (matrix) of floating point values is used (`double**`).

4.2.2 Outputting Heightfields as Images

2D arrays can be used to create images fairly easily. The Portable Pixmap Format (PPM) image format can be used , which simply stores the value of each pixel using values in text to the file. This format was chosen because it would be relatively easy to code functions to read at write these files in C.

4.2.3 Noise Functions

The core noise generation code is a version of the reference Java implementation from (Perlin, 2002) translated into C (Rosetta Code, 2024).

4.3 Implementation of Stream Power Model

Note: there is varying terminology used when referring to graphs. The word 'vertex' has been used so far to refer to a 'point' or 'node' of the graph. The word 'edge' is used when referring to the connections between vertices. However, in the code, the word 'node' has been used extensively to refer to a vertex.

4.3.1 Stream Graph Initialisation

Using the current algorithm, all vertices with the same uplift initially won't be able to form streams, as they have the same elevation, leading to slow performance over the first iterations as there are many lakes. By protruding each vertex by a small random elevation, the creation of stream trees is forced, reducing number of lakes and reducing computation time.

Poisson Disk Sampling

To create the stream graph, the vertices are distributed using Poisson disk sampling. This was implemented using the algorithm introduced in (Bridson, 2007).

Delaunay Triangulation

For the Delaunay triangulation for the Stream Graph, a library was used to speed-up development. This is an algorithm which has been implemented well for C++ (`https://github.com/artem-ogre/CDT`, licensed under the MPL-2.0 license.).

Labelling Stream Outflows

The lake overflow algorithm relies of water being able to flow out of the terrain. This means some vertices on the border of the terrain must be marked as outflows. Most simply, this can be all border vertices, or some which are specified by a user. To find all vertices on the border, all edges are checked to see if two triangles share it. If not, the edge must be on the border.

Voronoi Area Calculation

To calculate the Voronoi areas for each vertex, Algorithm 1 was written, which works by summing areas of triangles formed by the stream vertices, the midpoints of the stream edges and the circumcentres of the triangles formed by the edges.

Algorithm 1 Calculation of Voronoi areas.

```
1: for t ∈ Triangles do
2:     c ← CircumcentreOfTriangle(t)
3:     for i ← 1 to 3 do
4:         v ← t[i]
5:         for j ← 1 to 3 do
6:             if j ≠ i then
7:                 w ← t[j]
8:                 m ← ½(v.pos + w.pos)
9:                 v.area ← v.area + AreaOfTriangle(v.pos, m, c)
10:            end if
11:        end for
12:    end for
13: end for
```

4.3.2 Construction of Stream Trees

Each stream vertex stores references to the vertex which the stream flows out to, labelled the 'downstream' vertex, and the set of vertices which flow into a vertex, labelled the 'upstream' vertices. Algorithm 2 constructs the stream trees by creating edges in T from each vertex to their lowest elevation neighbouring vertex in G.

Algorithm 2 Construct Stream Trees.

```
1: for all v ∈ V do
2:     minH ← v.height
3:     minV ← 0
4:     for all w ∈ v.neighbours do
5:         if w.height < minV then
6:             minH ← w.height
7:             minV ← w
8:         end if
9:     end for
10:    v.downstream ← minV
11:    minV.upstream ← minV.upstream ∪ {v.downstream}
12: end for
```

4.3.3 Lake Overflow

Creating Lake Graph

A set of potential passes is created which store edges between lakes which correspond with the lowest pairs of neighbouring vertices in G which belong to different lakes. This set is built up using Algorithm 3.

Algorithm 3 Calculating Passes.

$P \leftarrow \emptyset$
for all $e \in E$ **do**
  $v \leftarrow e.v1$
  $w \leftarrow e.v2$
  if $L(v) = L(w)$ **then**
    continue
  end if
  $isPass \leftarrow True$
  for all $p \in P$ **do**
    if $(L(v), L(w))$ **then**
      $isPass \leftarrow False$
      if $\text{MAX}(v.height, w.height) < p.passHeight$ **then**
        $isPass \leftarrow True$
        $P \leftarrow P \setminus \{p\}$
      end if
    end if
  end for
  if *isPass* **then**
    $L_E \leftarrow L_E \cup \{(L(v), L(w))\}$
    $P \leftarrow P \cup \{(L(v), L(w), passHeight = \text{MAX}(v.height, w.height))\}$
  end if
end for

Constructing Lake Trees

Once the candidate passes are identified, the candidates are considered to form an edge in L_T in order of lowest pass height. A priority queue is used to implement this. To construct the roots of the lake trees from boundary lakes, Algorithm 4 is used and then the form the full trees, Algorithm 5 is performed.

Algorithm 4 Lake river mouths.

$candidates \leftarrow PriorityQueue()$
for all $l \in L$ **do**
  if $l \in riverMouths$ **then**
    for all $k \in l.neighbours$ **do**
      if $k \notin riverMouths$ **then**
        for all $p \in P$ **do**
          if $p = (k, l) \vee p = (l, k)$ **then**
            $candidates \leftarrow candidates \cup \{(k, l)\}$
          end if
        end for
      end if
    end for
    $l.neighbours \leftarrow \emptyset$
  end if
end for

Algorithm 5 Lake trees.

```
validPasses ← ∅
while candidates ≠ ∅ do
    min ← POP(candidates)
    validPasses ← validPasses ∪ {min}
    for all l ∈ min.higherlake.neighbours do
        if l ∈ riverMouths then
            for all p ∈ validPasses do
                if p = (min.higherlake, l) ∨ p = (l, min.higherlake) then
                    candidates ← candidates ∪ {(min.higherlake, l)}
                end if
            end for
        end if
    end for
    NEIGBOURS(min_HIGH) ← ∅
end while
```

Modify Stream Graph

The stream graph is updated with new connections in the lake graph using Algorithm 6.

Algorithm 6 Stream Graph is modified by Lake Trees

```
for all p ∈ validPasses do
    lakeRoot ← p.higherLake.streamVertex
    receiver ← p.lowerPassVertex
    receiver.upstream ← receiver.upstream ∪ {lakeRoot}
    lakeRoot.downstream ← receiver
end for
```

4.3.4 Drainage Area Calculation

The drainage area is calculated using a recursive scheme where the leaf vertices of the stream trees are calculated first and then the drainage area for all vertices to the root are calculated from those results using Algorithm 7.

Algorithm 7 Calculate Drainage Area.

```
function DRAINAGEAREA(v)
    a ← v.area
    for all w ∈ V.upstream do
        a ← a + DRAINAGEAREA(w)
    end for
    return a
end function
```

The stream power model normally uses drainage area itself as a proxy for the the flow of water through a point in a stream, but the algorithm can be simply modified to introduce variable

rainfall by introducing a rainfall map where rainfall for different vertices is sampled from which scales the drainage area calculation, which despite the name, now calculates a volume of water using Algorithm 8. The results of this are discussed in the next chapter.

Algorithm 8 Modified Drainage Area for Rainfall.

```
1: function DrainageArea(v)
2:     a ← v.area ∗ v.rainfall
3:     for all w ∈ V.upstream do
4:         a ← a + DrainageArea(w)
5:     end for
6:     return a
7: end function
```

4.3.5 Stream Power Update

The stream power update is done in the opposite way to the drainage area calculation, using Equation 3.6 to compute the new elevation of vertices from root to leaves (Algorithm 9, as this is required for the implicit solver to work.

Algorithm 9 Modified Drainage Area for Rainfall.

```
1: function Update(v)
2:     Apply Stream Power Erosion
3:     for all w ∈ V.upstream do
4:         Update(w)
5:     end for
6:     return a
7: end function
```

4.3.6 Creation of terrain model from Stream Graph

The stream graph can be converted into a heightfield using Algorithm 10. The second output method is basic tessellation where a 3D mesh is created using the triangulation of the stream graph.

Algorithm 10 Create heightfield from stream graph.

```
1: function CreateHeightfield(σ, Ω)
2:     H, K are heightfield size Ω
3:     for all v ∈ V do
4:         for i ← 1 to |Ω| do
5:             for j ← 1 to |Ω| do
6:                 g ← e^{−||p_v − p_w||^2/2σ^2}
7:                 H_{i,j} ← H_{i,j} + h_v g
8:                 K_{i,j} ← K_{i,j} + g
9:             end for
10:        end for
11:    end for
12:    for i ← 1 to |Ω| do
13:        for j ← 1 to |Ω| do
14:            H_{i,j} ← H_{i,j} / K_{i,j}
15:        end for
16:    end for
17:    return a
18: end function
```

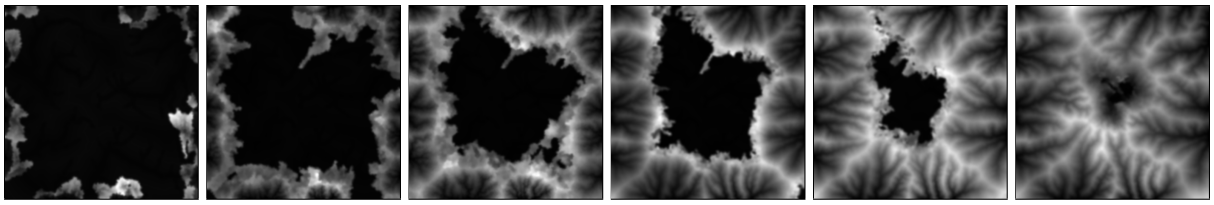
4.4 Complexity

The main algorithm has time complexity $O(n)$, but has the problem of local minima. The lake overflow algorithm is used to solve this. However this increases the time complexity to $O(n^2)$.

4.5 Testing

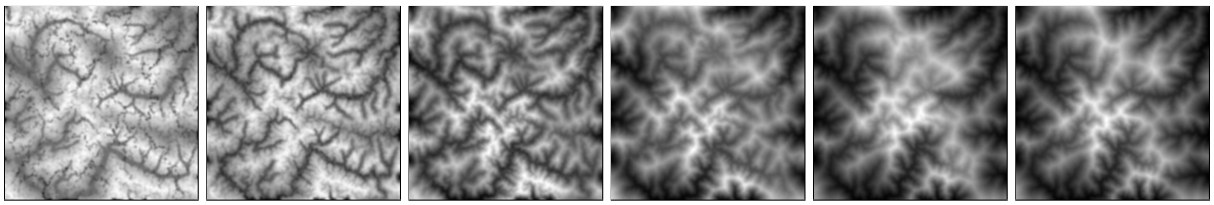
4.5.1 Issues

The development of the system generally went well, apart from one major issue with the lake algorithm. The system would eventually produce a terrain after enough iterations, so the issue wasn't initially obvious. However, looking at at snapshots from the simulation at different iterations (Figure 4.1), it was clear there was a problem with the lakes in the terrain not being calculated correctly. Investigating with a small number of vertices and manually drawing the lake graph from the memory addresses, the issue was found to be some lakes flowing to multiple outflows, corrupting the lake tree structure. This was fixed by first checking if a lake is already flowing to a lower lake before adding a pass. The updated algorithm is shown in Figure 11 and the snapshots of the correct generation in Figure 4.2.



(a) i=0 (b) i=5 (c) i=15 (d) i=50 (e) i=100 (f) i=200

Figure 4.1: Snapshots of incorrect terrain generation



(a) i=0 (b) i=5 (c) i=15 (d) i=50 (e) i=100 (f) i=200

Figure 4.2: Snapshots of correct terrain generation.

Algorithm 11 Fixed Lake trees.

```
1:  validPasses ← ∅
2:  while candidates ≠ ∅ do
3:      min ← POP(candidates)
4:      for all p ∈ validPasses do
5:          if p.higherlake = min.higherlake then
6:              goto line 2
7:          end if
8:      end for
9:      validPasses ← validPasses ∪ {min}
10:     for all l ∈ min.higherlake.neighbours do
11:         if l ∈ riverMouths then
12:             for all p ∈ validPasses do
13:                 if p = (min.higherlake, l) ∨ p = (l, min.higherlake) then
14:                     candidates ← candidates ∪ {(min.higherlake, l)}
15:                 end if
16:             end for
17:         end if
18:     end for
19:     NEIGBOURS(min_HIGH) ← ∅
20: end while
```

4.5.2 Memory-safety Testing

C/C++ is not a memory-safe language. Valgrind was used to check for memory leaks. The tests were successful, and the system had no memory leaks.

Chapter 5

Results

All examples of terrains were generated on a laptop computer with an Intel Core i7 CPU, clocked at 2.8 GHz with 8GB of RAM. Two methods for displaying terrains are used. Basic tessellation of the stream graph shows the shape of the terrain, and emphasises the stream network. High quality output images were generated by importing the resulting heightfields into Terragen™ (`https://planetside.co.uk`), a terrain rendering software, where realistic terrain shading was applied to render final images. Examples are shown in Figure 5.1.

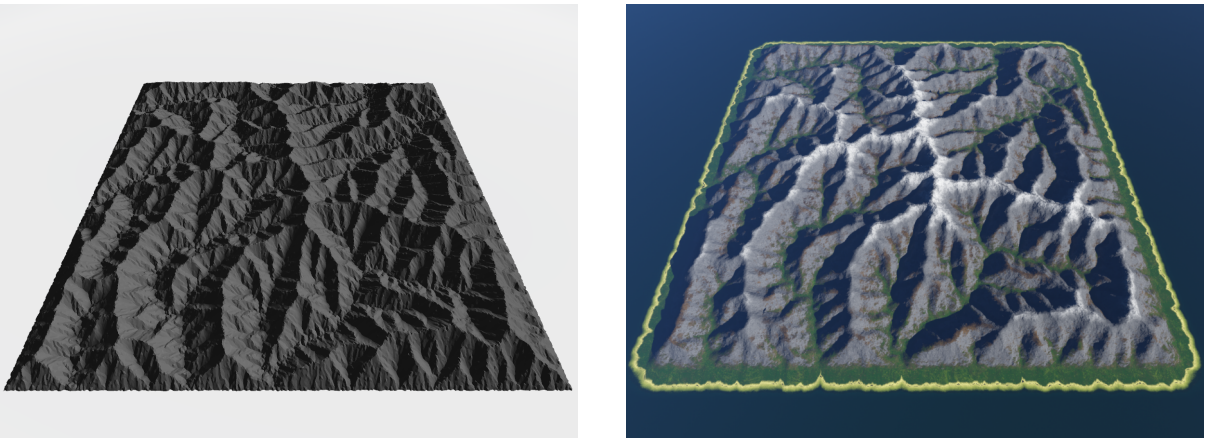


Figure 5.1: Left: Example of a mesh created from a tessellation of a stream graph, rendered in real-time using Microsoft 3D Viewer. Right: Heightfield rendered using Terragen™

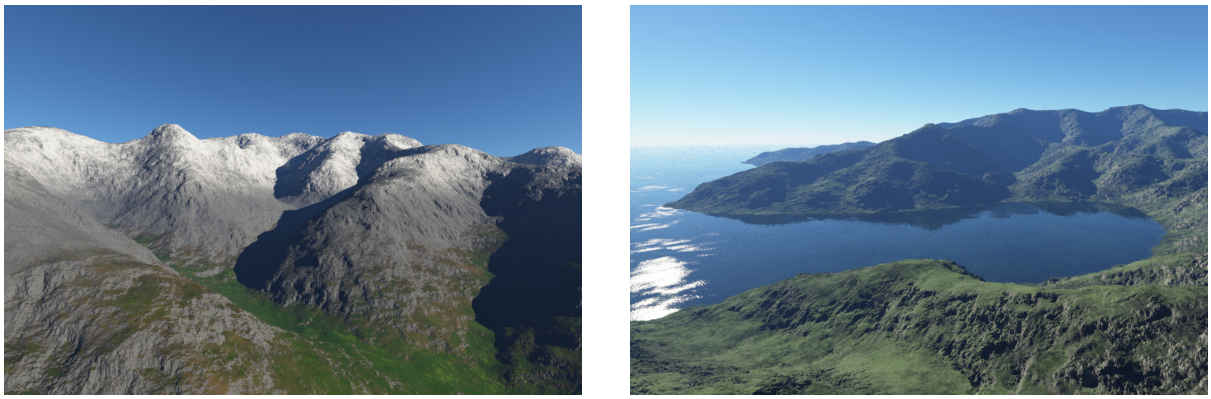
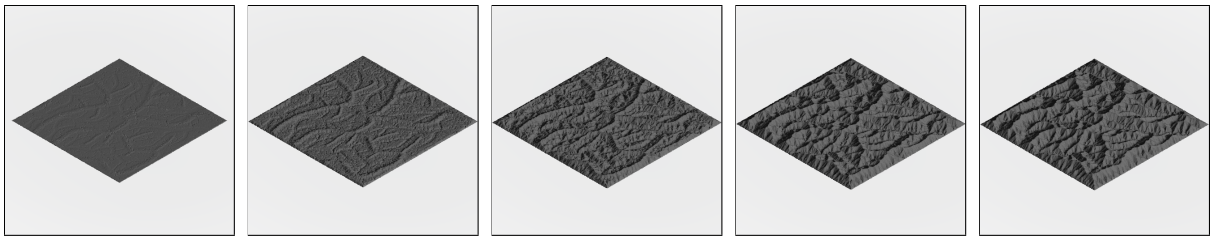


Figure 5.2: Close-up high-quality renderings.



(a) i=0 (b) i=5 (c) i=15 (d) i=50 (e) i=100

Figure 5.3: Terrain model snapshots taken after increasing iterations *i*.

The evolution of the terrain generation during the erosion simulation is shown in Figure 5.3.

5.1 Landform Generation

There is a range of landforms in the natural world, which are a result of many geological processes. The system only simulates the effect of fluvial erosion from streams, and thermal erosion from hill slope processes. Here, we shall look at which landforms have, and which have not, emerged from the simulation.

5.1.1 Landforms resulting from Fluvial Erosion

V-shaped Valleys

The process of fluvial erosion creates V-shaped valleys, which the system models very well. The dendritic (tree-like) structure of the streams are reproduced in the model of the valley. The hill-slope thermal erosion processes can also be seen in the synthesised terrain.

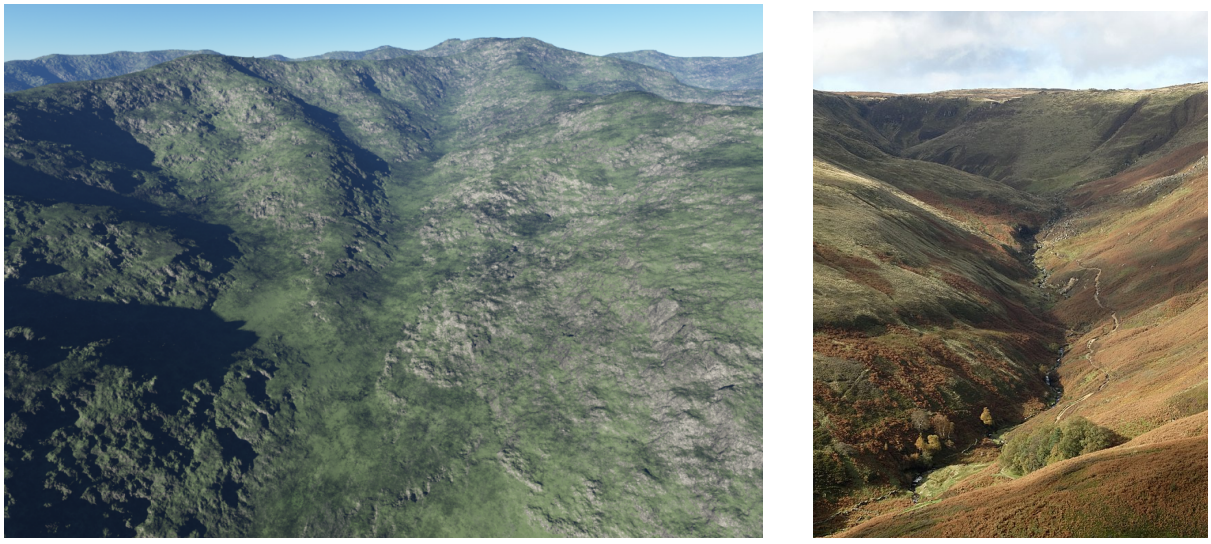


Figure 5.4: Left: V-shaped valley terrain produced by the system. Right: Example of a real V-shaped valley. A view of Grindsbrook Clough from the east cc-by-sa/2.0 - © Neil Theasby - geograph.org.uk/p/3713919.

Interlocking Spurs

The phenomena of interlocking spurs results from fluvial erosion as streams wind to avoid harder rock. These landforms emerge naturally from the simulation, as shown in Figure 5.5.

Figure 5.5: Left: Terrain produced by the system. Right: Example of a interlocking spurs. Towards the top of Grindsbrook Clough cc-by-sa/2.0 - © Andrew Hill - geograph.org.uk/p/3102696

Deltas

Deltas are areas of sediment which is deposited at river mouths. Braiding can occur where streams divide and converge again. This does not happen in the simulation, as streams can only flow in one direction.

Meanders and Oxbow Lakes

When rivers flow over flatter land, bends in the river develop due to erosion called meanders. When there is a lot of water flowing, the river cuts across this bend which cuts off the original meander, leaving an oxbow lake. This process is not simulated, as flat land is not modelled by the system.

5.1.2 Other Landforms

Some natural landforms cannot be formed by the simulation, as these require the simulation of different geological processes. For examples, landforms which are a product of glacial erosion, such as U-shaped valleys, hanging-valleys, arêtes and corries, will never form.

5.1.3 Realism

To evaluate the realism of the terrain models produced by the system, the results were compared against real world terrains.

Ideally, a elevation dataset and a matching uplift dataset could be used to evaluate the system by providing the real uplift data as input and comparing the result with the real elevation data. However, although elevation data is relatively easy to find online, simple uplift data like is used in this model is not available, as in reality uplift rates are always varying and are caused by very complex subsurface tectonic geology which is is difficult to model.

Instead, the visual realism of the model can be evaluated by comparing terrains produced by the system with elevation data of terrains which were mainly formed by fluvial erosion. Real height data for an area in the USA was taken from (United States Geological Survey, 2021). Figure 5.6 shows that the system successfully produces realistic stream structures which closely resemble the dendritic patterns in the real elevation data.

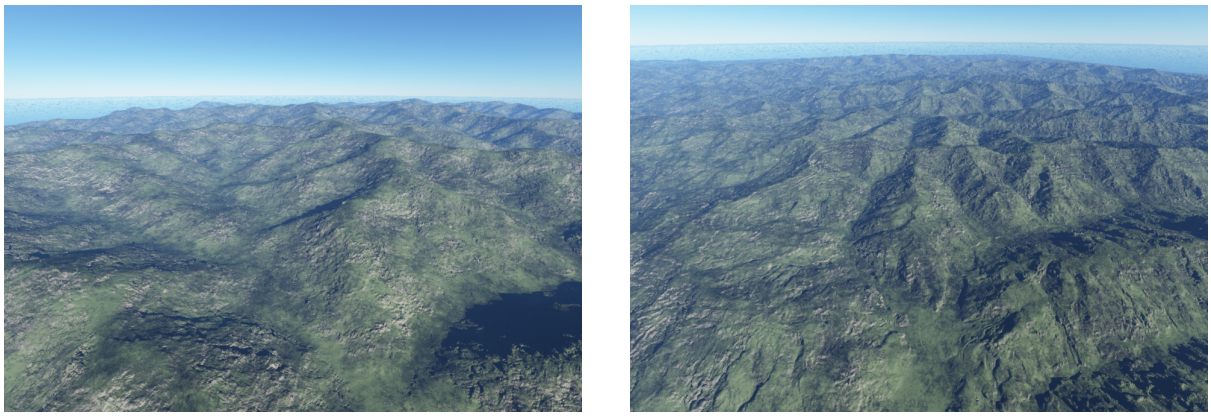


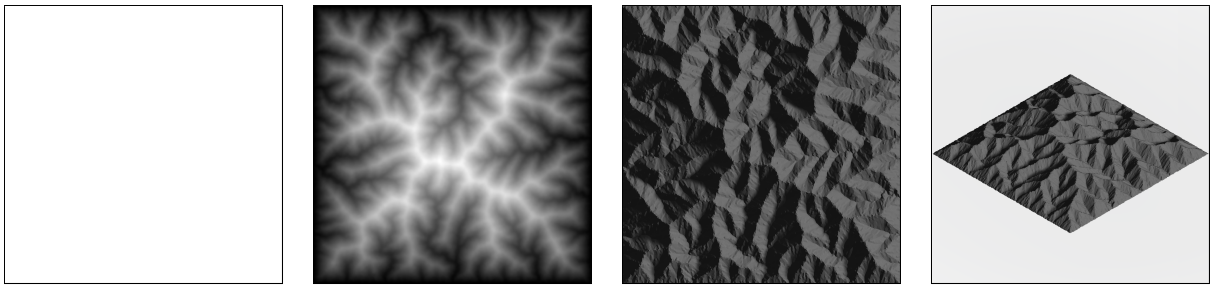
Figure 5.6: Left: Terrain model created by the system. Right: Real terrain data from the Blue Ridge Mountains, Tennessee, USA.

5.2 Authoring

Authoring is the process of designing a terrain. Although the model mandates a certain tree structure of the streams in a terrain, the user can strongly influence the generation processes of the terrain through authoring, producing a designed, yet realistic, final terrain.

5.2.1 Uplift

It is through changing the input uplift, that the most control over the terrain generation is had. The absolute uplift values have an effect on the terrains height which is discussed in Section 5.3. Here we look at how the shape and gradient of the input changes the terrain and how users can force the creation of prescribed streams and ridges. The figures below show the resulting terrains which generate from the some basic uplift fields.



(a) Uplift. (b) Heightfield. (c) Terrain. (d) Isometric view.

Figure 5.7: Constant uplift.

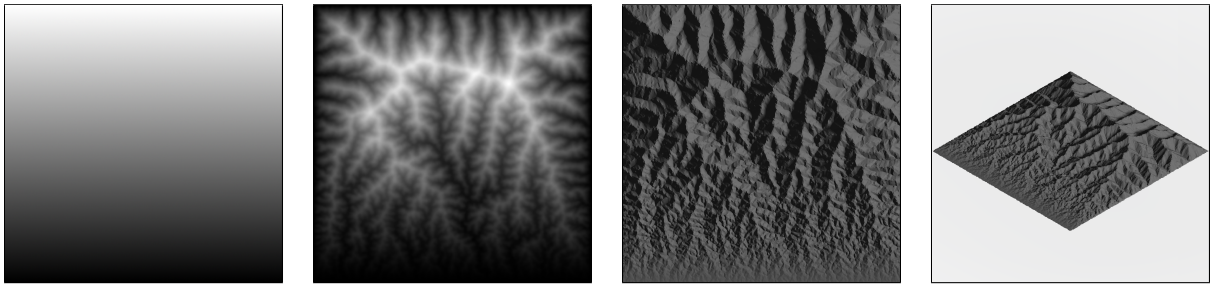


Figure 5.8: Gradient uplift.

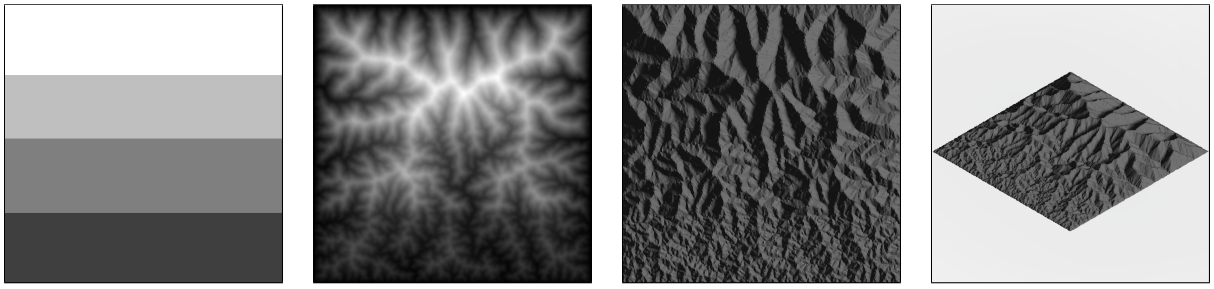


Figure 5.9: Stepped gradient uplift.

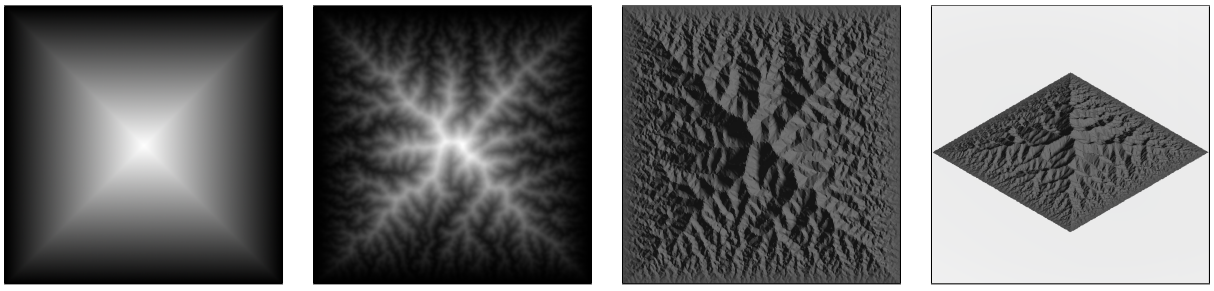


Figure 5.10: Diamond gradient uplift.

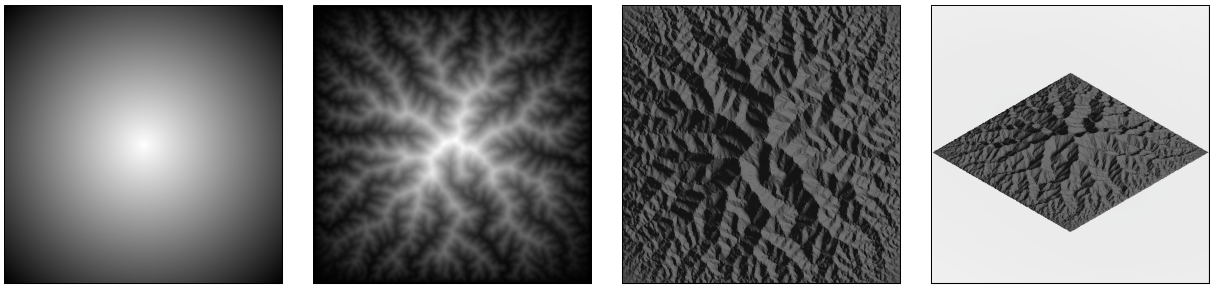
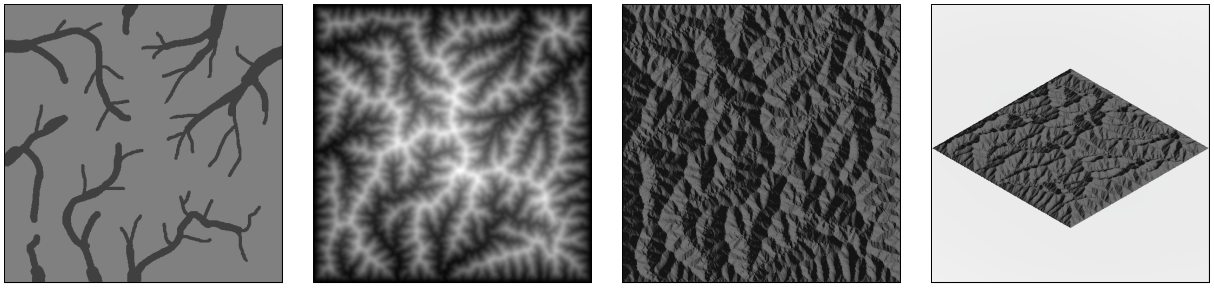


Figure 5.11: Radial gradient uplift.

Streams generally form in the direction of of the gradient with area of high uplift resulting in the tallest mountains. The stepped uplift gradient in Figure 5.9 gives less uniform valleys than in the smooth gradient in Figure 5.8. The results shown for the constant uplift may appear erroneous, as how can a constant uplift cause any streams to form, if the entire terrain is of equal height? The reason streams, and terrains, still form is due to the random protrusion in starting height, introduced to reduce the early lake count to increase performance.

Modelling Streams and Ridges

Uplift modelling can be used to author the terrain to include user-defined streams and mountains. Simple stream and ridge networks can be easily painted which gives very good control over the terrain formation, with the model ensuring geological consistency. Figure 5.12 shows an example of a user-painted stream network as an input to the system. The generated terrain is constrained to follow this, with the ridge network emerging from the simulation. Equally, the opposite can be done with mountains prescribed with a user-defined ridge network (Figure 5.13, in which the stream network is emergent. Figure 5.14 shows an example of using a combination of these techniques to author a terrain.



(a) Uplift. (b) Heightfield. (c) Terrain. (d) Isometric view.

Figure 5.12: Modelling streams in uplift domain.

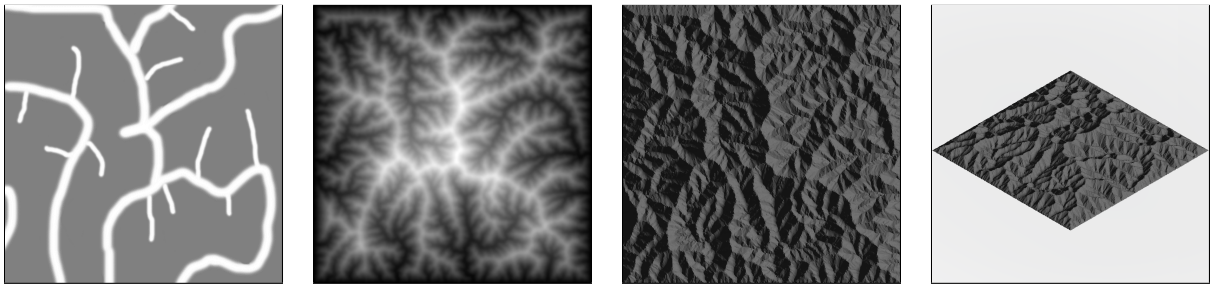


Figure 5.13: Modelling ridges in uplift domain.

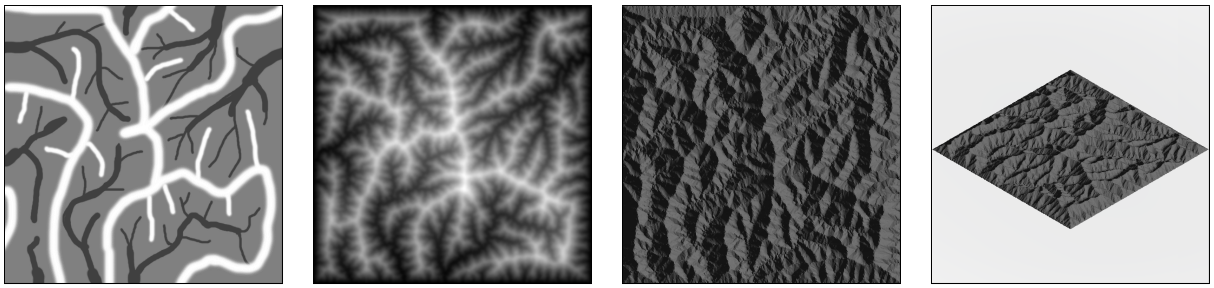
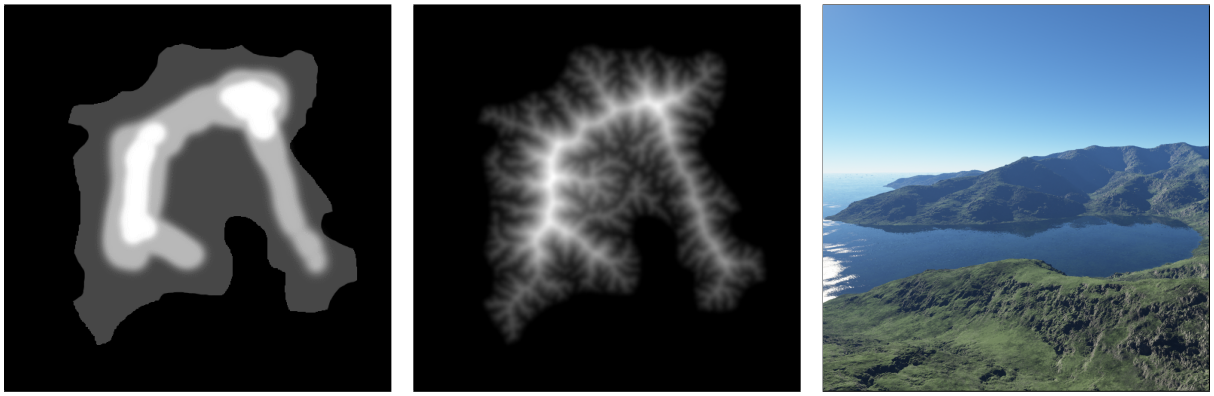


Figure 5.14: Modelling streams and ridges in uplift domain.

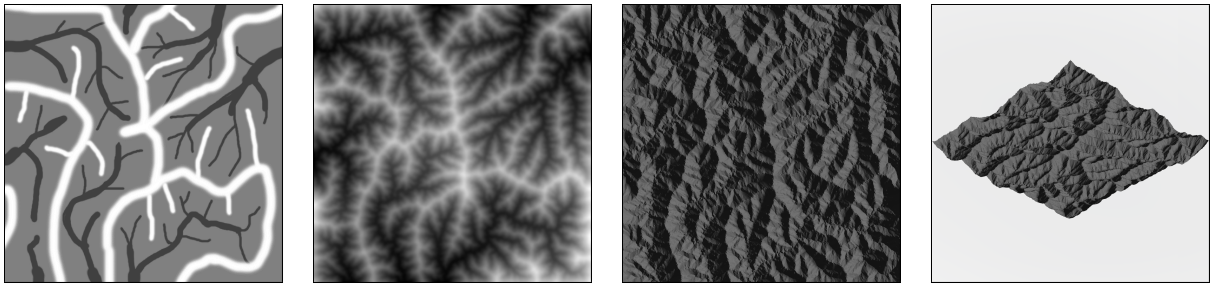
5.2.2 Outflows and Islands

In the default configuration, the entire boundary of the terrain is set as an outflow of the terrain, meaning water will always flow to the edge. This is good for generating terrains which are islands surrounded by oceans. The shape of which can be made non-square by setting 'sea' to have zero uplift. An example is shown in Figure However, to generate inland terrains, an alternative method can be used. Boundary vertices with an uplift below a certain threshold can be set as the only outflows which results in terrains like the example in Figure 5.16. This actually improves the time performance of the system, as there are less lake outflow vertices.



(a) Uplift. (b) Heightfield. (c) High-quality close-up.

Figure 5.15: Generating an island shaped terrain.

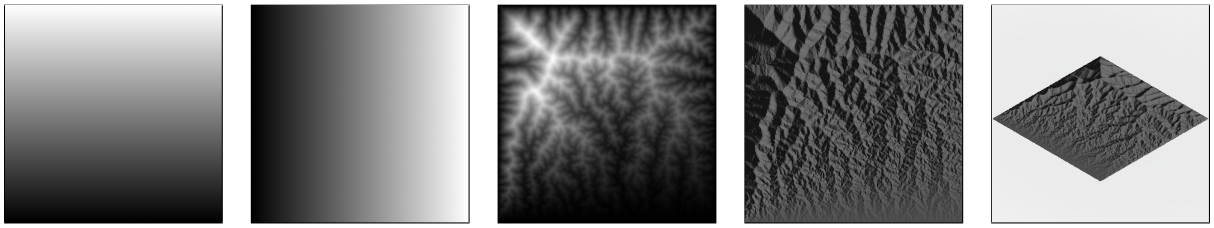


(a) Uplift. (b) Heightfield. (c) Terrain. (d) Isometric view.

Figure 5.16: Generation of a terrain with specified outflows.

5.2.3 Rainfall

The results of introducing variable rainfall to the model area shown in Figure 5.17, with a rainfall gradient across the terrain. Areas with higher rainfall produce lower elevation mountains, and smaller scale streams.



(a) Uplift. (b) Rainfall. (c) Heightfield. (d) Terrain. (e) Isometric view.

Figure 5.17: Rainfall

5.3 Parameter Analysis

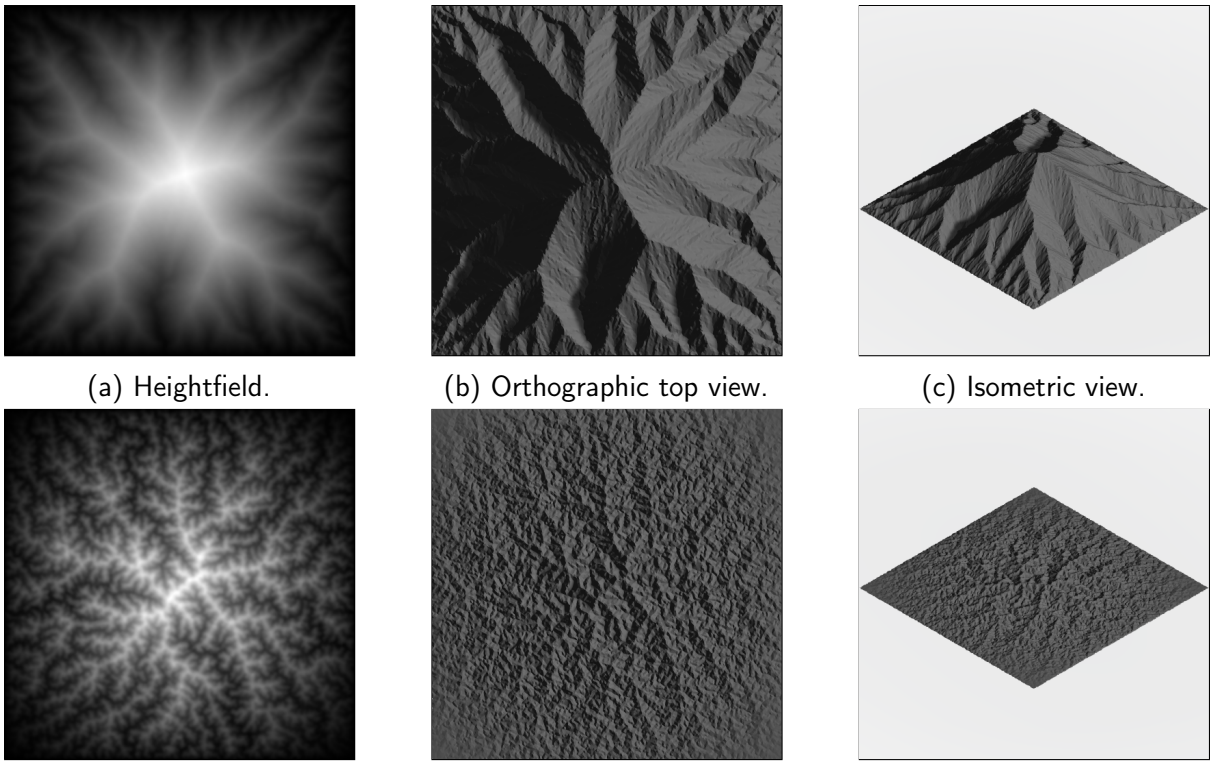
There are further ways in which the simulation can be controlled, through the adjustment of values of the erosion parameters used. Full results can be found in Appendix A.

5.3.1 Stream Power Law Parameters

The values to choose for the stream power parameters are not particularly obvious. Baseline values were initially set to values used in (Cordonnier et al., 2016) and then tests were carried out to find the results of varying these parameters.

Exponents m and n

The ratio m/n between between the drainage exponent m and the slope exponent n has a large effect on the shape of the streams generated in the model. To simplify the implicit solver, n is fixed at 1, and m can be changed to vary the ratio. Values around 0.5 were tested, which is the norm in geological studies (Whipple and Tucker, 1999).Examples of terrains generated with m set to 0.4 and 0.6 are shown in Figure 5.18. With smaller m values, the stream are on a much larger scale, with a low number of branches. Increasing m increases the number of streams and also reduces the overall vertical height of the entire terrain. As m increases the maximum height of the terrain approaches 0 and shown in the graph in Figure 5.19.



(a) Heightfield. (b) Orthographic top view. (c) Isometric view.

Figure 5.18: Top: m=0.4 with maximum height=9070m. Bottom: m=0.6 with maximum height=1098m.

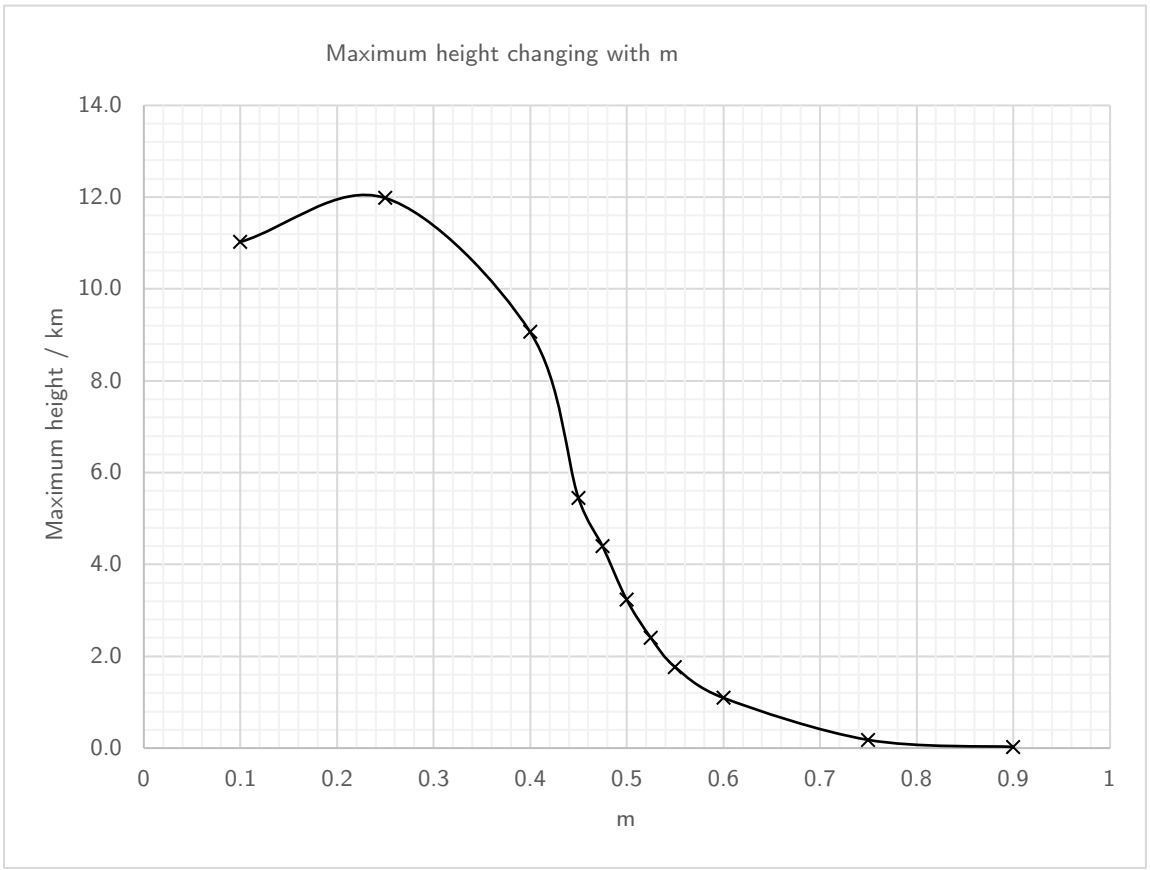


Figure 5.19: A graph showing how the maximum height of a terrain changes with m.

Uplift and erosion constant (k)

The uplift rate and the erosion constant (k) work against each other, with larger uplift values producing taller terrains, and larger erosion values, producing lower elevation terrains.

The results from the uplift experiment are shown in Figure 5.20. The relationship between the maximum uplift and the maximum height is not linear, and follows $h_{max} \propto {u_{max}}^{0.568}$. This result is different than the conclusion made in (Cordonnier et al., 2016), which uses a similar model, which found $h_{max} \propto u_{max}$.

The effect of changing the erosion constant, does not have a linear effect on the maximum height as shown in Figure 5.21. However when plotting the erosion constant against $1/h_{max}$, a linear relationship is revealed.

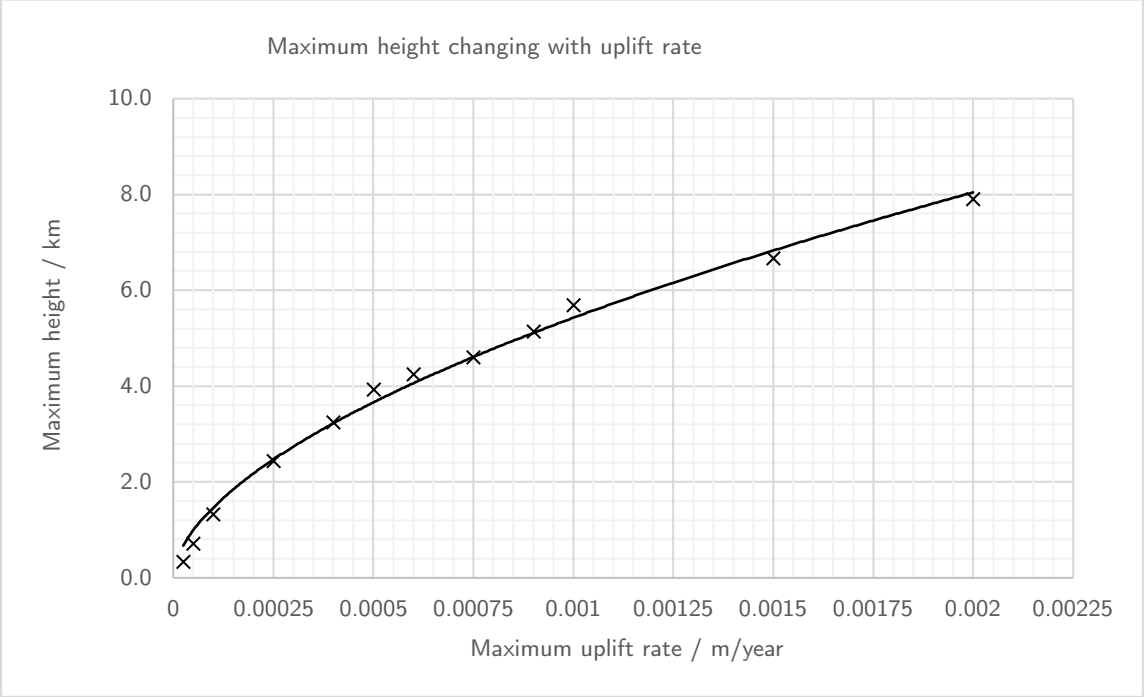


Figure 5.20: A graph showing how the maximum height of a terrain changes with uplift rate.

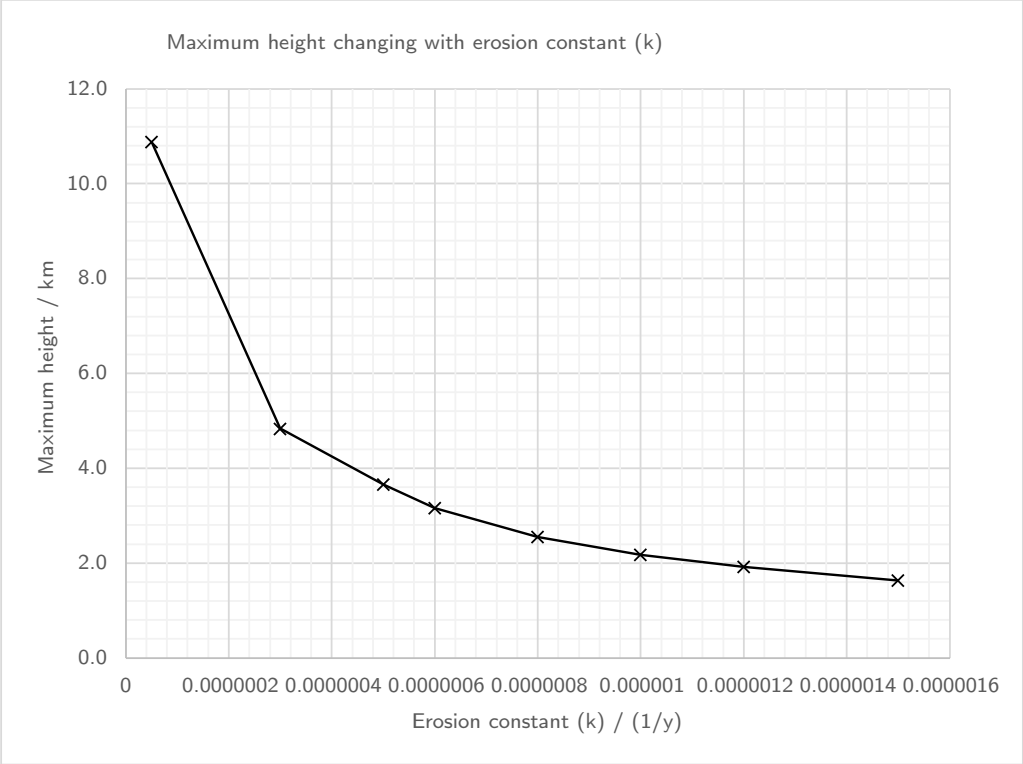


Figure 5.21: A graph showing how the maximum height of a terrain changes with the erosion constant k.

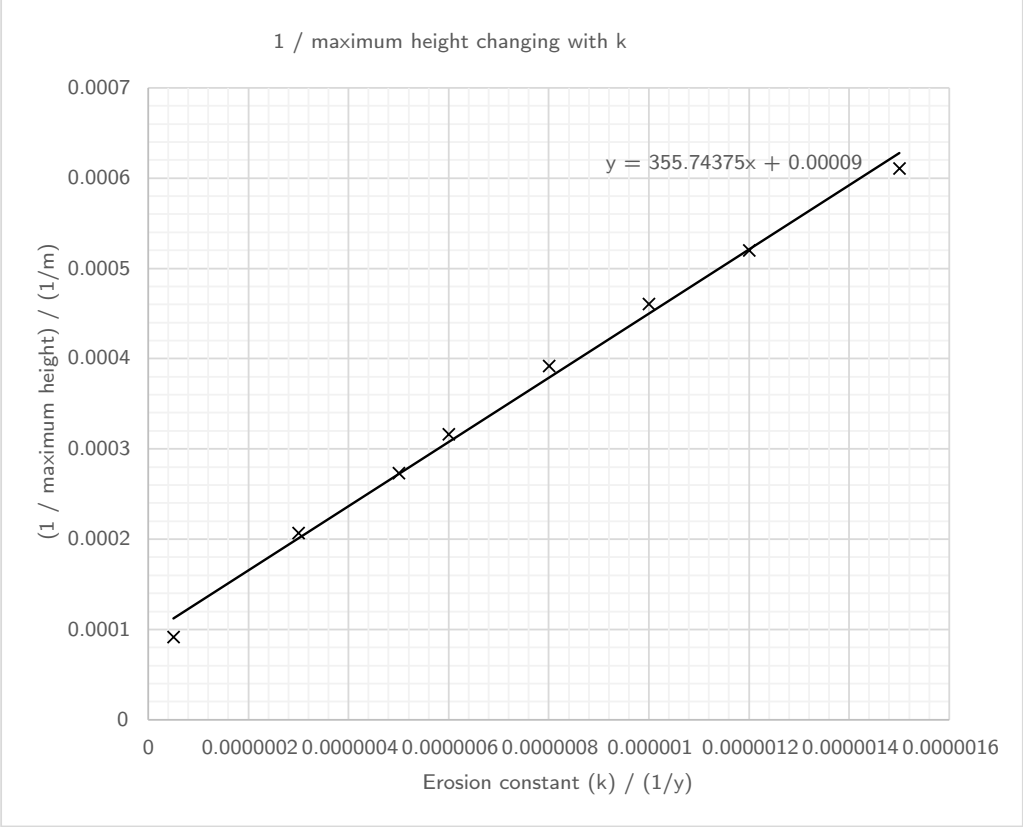
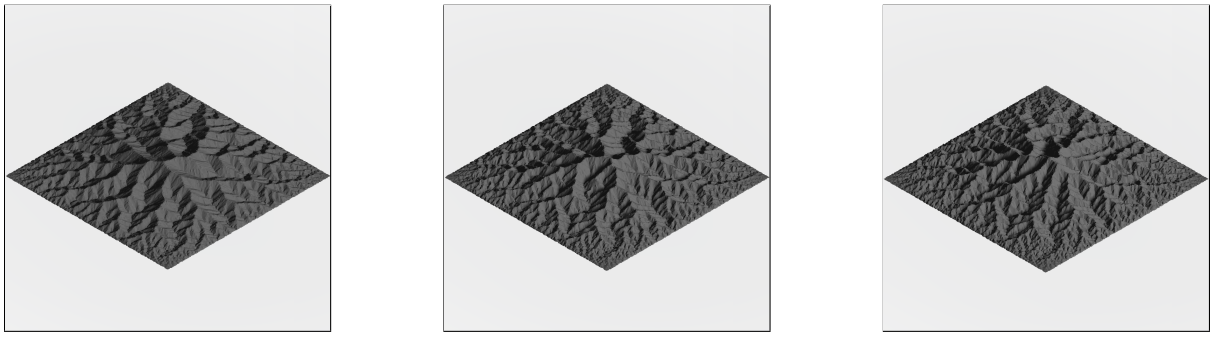


Figure 5.22: A graph to determine the relationship between the erosion constant k and the maximum height of a terrain

5.3.2 Thermal Erosion

The model of thermal erosion in the algorithm only affects the slope of the terrain, not allowing banks which are too steep. A talus angle α can be set, which is the maximum angle allowed. The effect of changing the talus angle is shown in Figure 5.23. The general stream structure remains the same, with only the the gradient of the slopes changing. An angle of 35° works well to smooth the slopes whilst still maintaining stream definition.

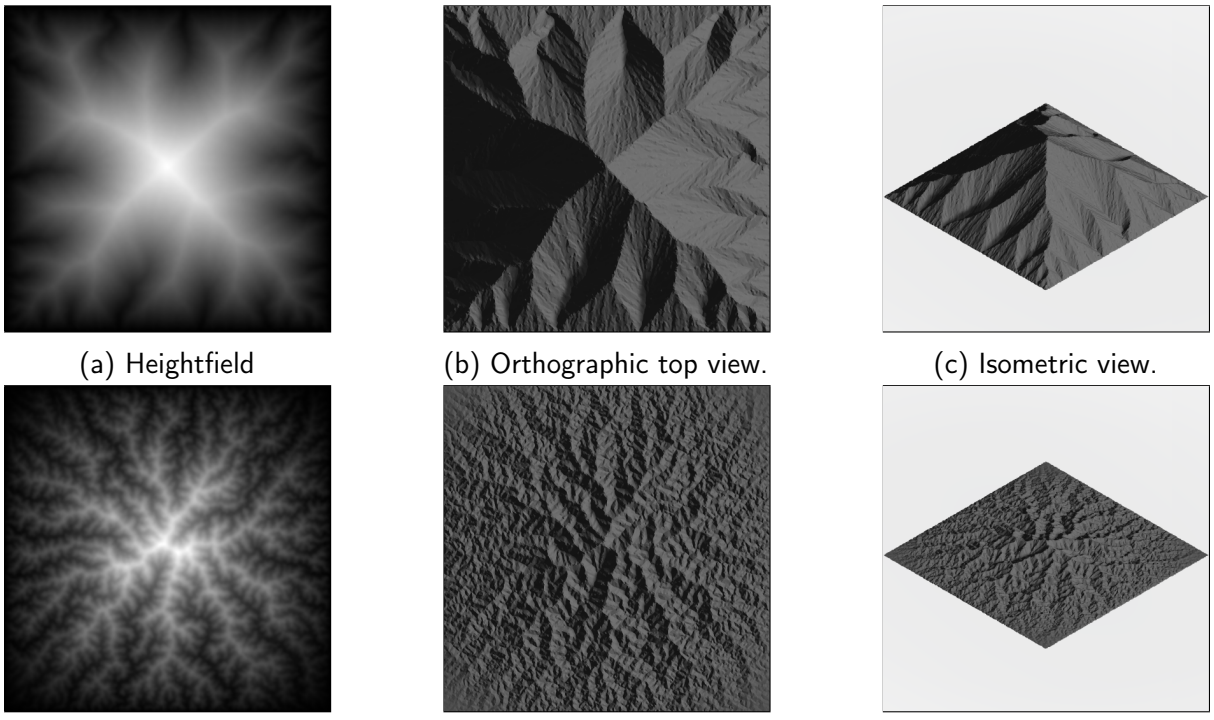


(a) α=25° (b) α=35° (c) α=45°

Figure 5.23: Terrains with varying thermal erosion talus angles.

5.3.3 Terrain Size

The physical size of the terrain produced can be changed to generate landforms at different scales. The results (Figure 5.24) appear very similar to the results from the experiments on m, with larger terrain sizes producing heightfields similar to larger values of m. However the actual heights are quite different, with realistic maximum heights to match the size of the terrain.



(a) Heightfield (b) Orthographic top view. (c) Isometric view.

Figure 5.24: Top: a terrain size 4096m x 4096m (maximum height=1205). Bottom: a terrain size 65536m x 65536m (maximum height=4269m).

5.4 Performance

To evaluate the performance of the system, experiments were run to determine the empirical time and memory complexities against different parameters. The criteria for for convergence of the model was also analysed.

5.4.1 Time Complexity

The performance in terms of simulation time can be measured against many variables, but the main factor in the time complexity of the system is the number of vertices used in the stream graph. The affect of this on the total run-time of the system was investigated, as well as how the duration of each iteration changes within a single simulation. The effect of the size of the terrain on time performance was also analysed.

Number of Vertices

Independent variable: number of vertices $|V|$ in stream graph G.

Dependent variable: time taken for terrain to converge.

Controlled variables: Terrain size: 32768m x 32768m, m= 0.5, k=0.00000056 / year, uplift=0.0004 m/year.

As can be seen in Figure 5.25, the time complexity of the implementation is quadratic $O(n^2)$.

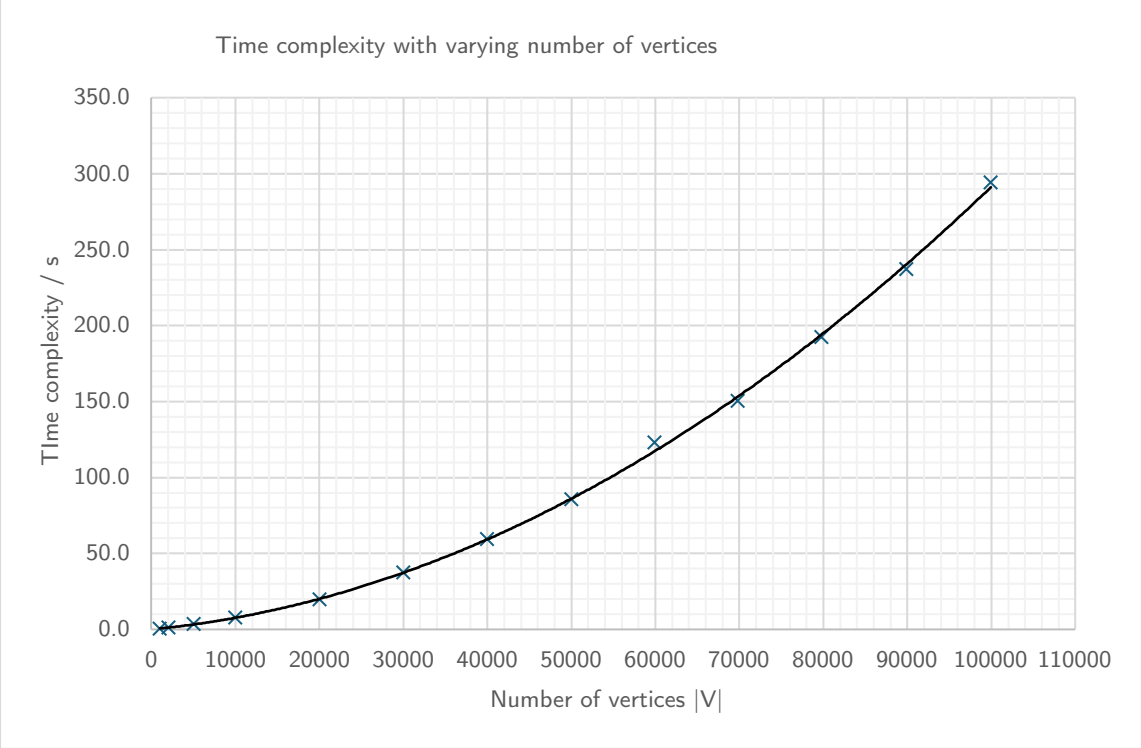


Figure 5.25: A graph showing the empirical time complexity of the system.

Iterations

During testing, it was observed that the time duration for each iteration during a given simulation, was not constant. To investigate this, many tests were run for different numbers of vertices and the duration of each iteration, recorded. The results are shown in Figure 5.26, but are easier to see in a log graph show in Figure 5.27.

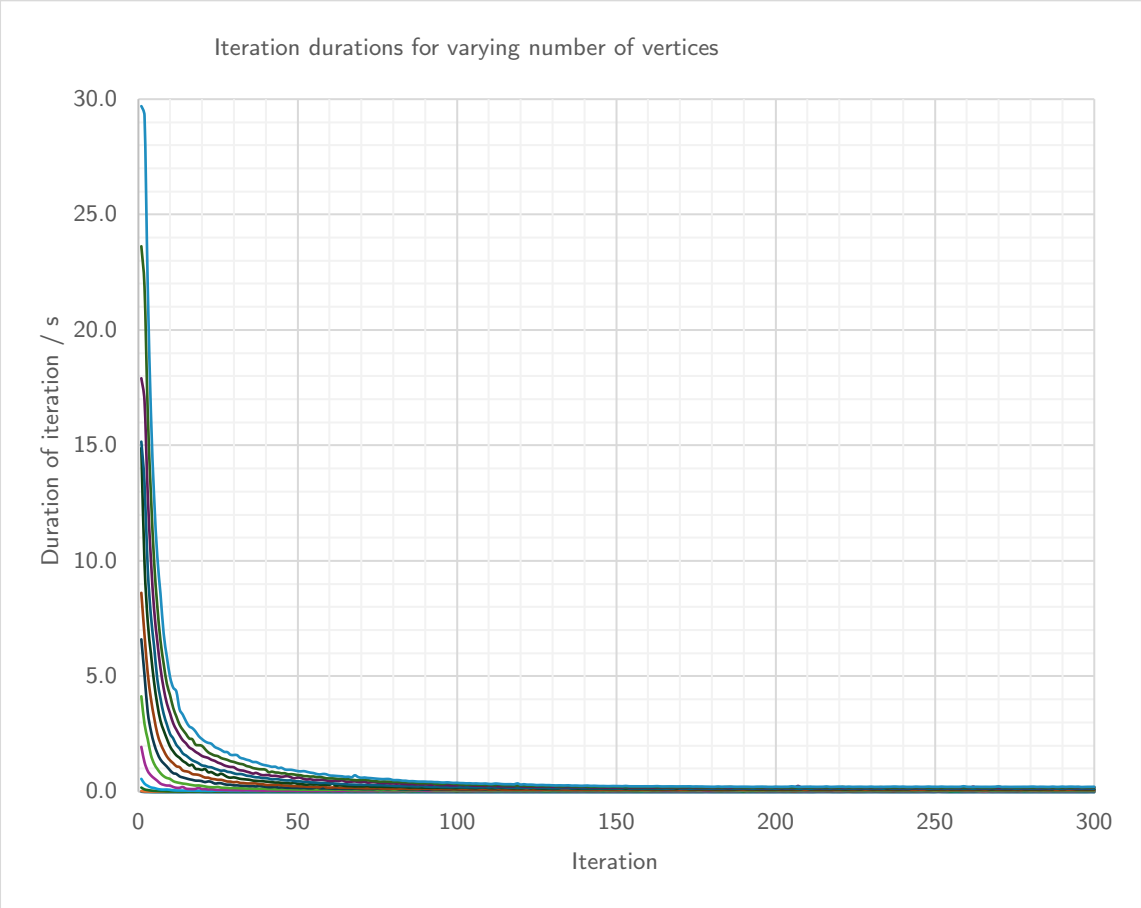


Figure 5.26: A graph of the durations for each iteration of the simulation.

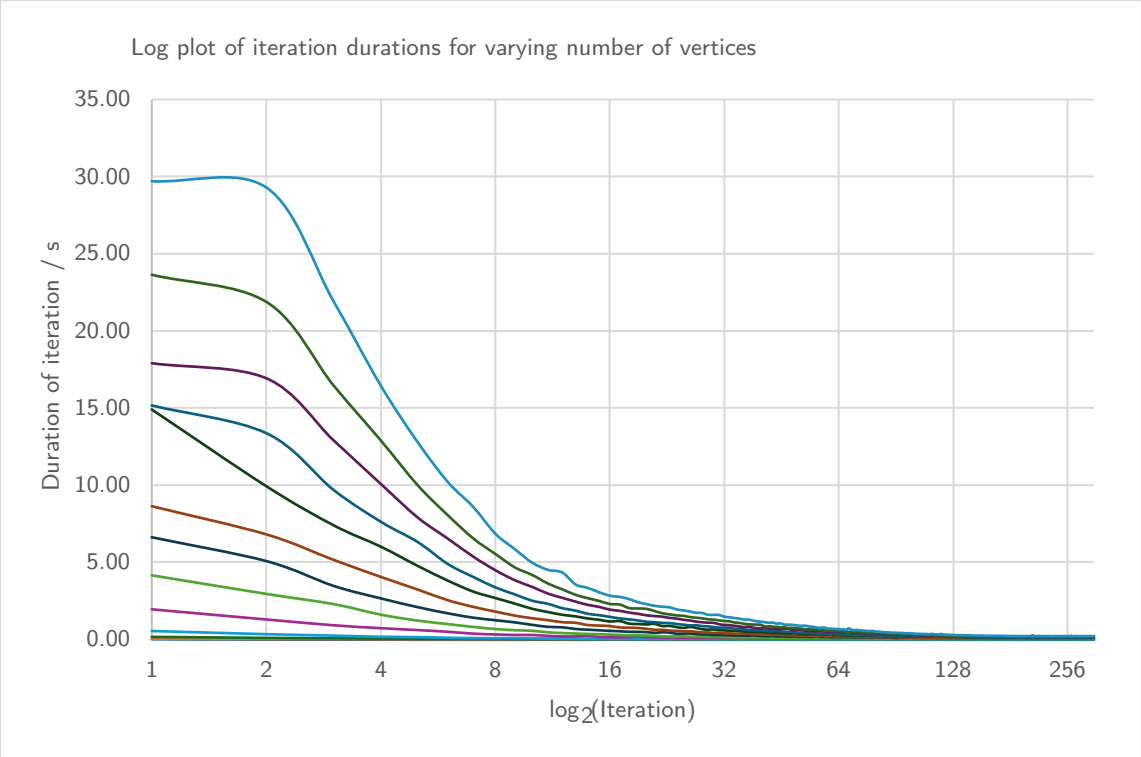


Figure 5.27: A graph of the durations for each (log) iteration of the simulation.

In these tests, in addition to measuring the duration of each iteration, the number of lakes was also recorded. An interesting observation can be made by looking at Figure 5.27 and Figure 5.28; the there is a clear relationship between the number of lakes and the duration of each iteration. This is expected, as the part of the algorithm which deals with local minima (lakes) is the most significant contribution to the overall time complexity. To deduce the exact relationship between the number of lakes and the duration of each iteration, another graph shown in Figure 5.29, can be drawn. This shows that the relationship, is quadratic. In other words, the empirical time complexity of the implementation of the lake algorithm, is $O(n^2)$.

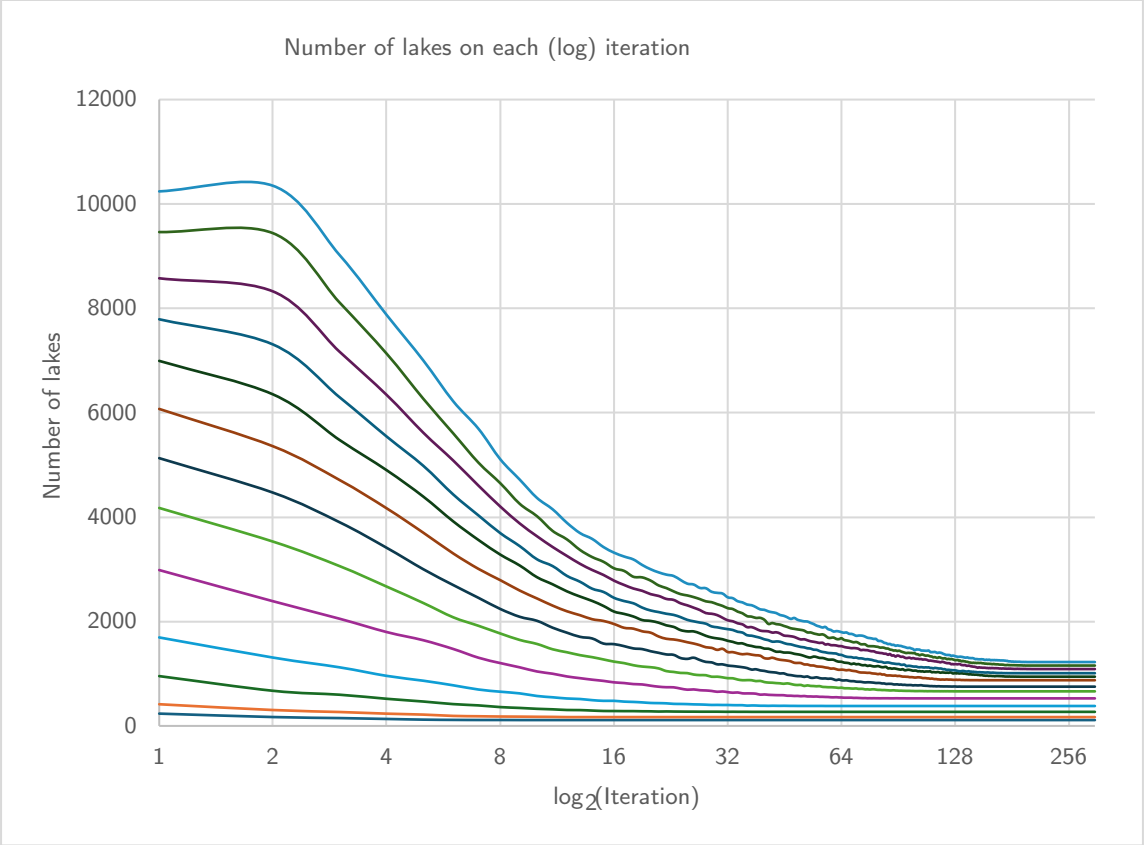


Figure 5.28: A graph of how the number of lakes changes for each iteration.

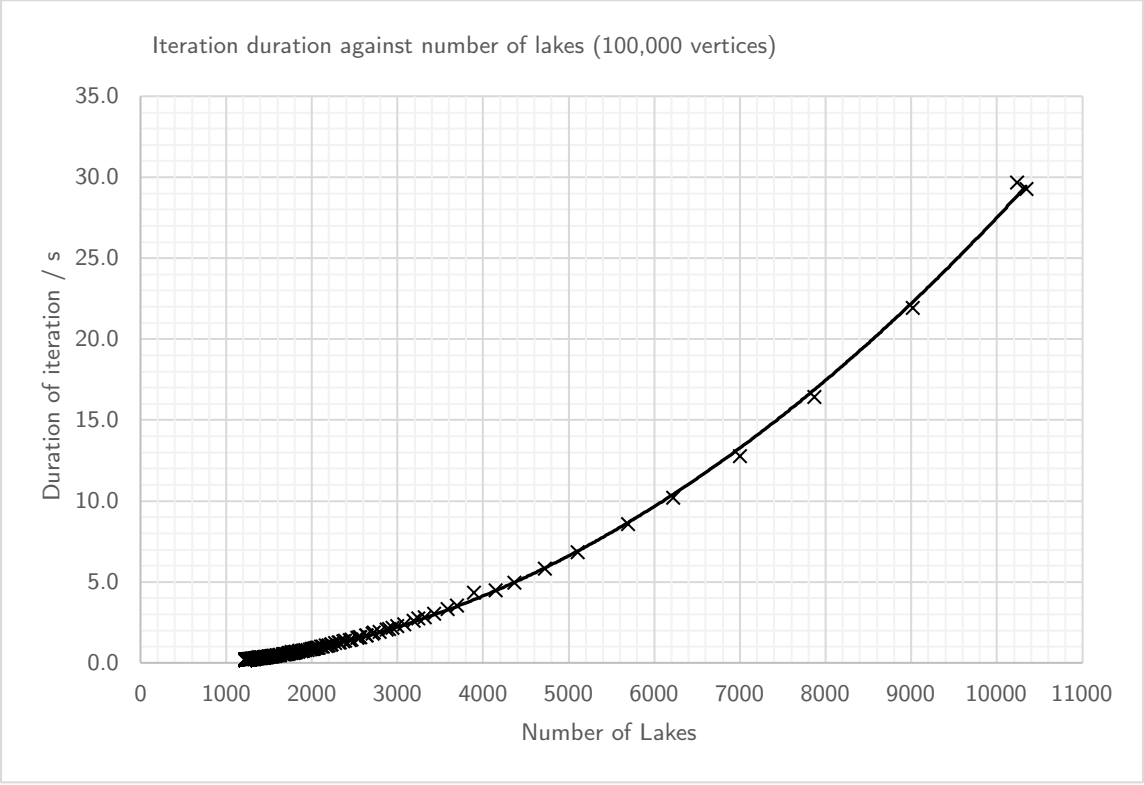


Figure 5.29: A graph showing the relationship between the number of lakes and the duration of an iteration.

Terrain Size

Independent variable: physical size of terrain model.

Dependent variable: time taken for terrain to converge.

The terrain size has an interesting effect on the time complexity of the system. The relationship is non-linear and is graphed in Figure 5.30. Small terrains are very slow to generate. For terrains larger than around 20km by 20km, the time complexity is much better and remains around a constant 40 seconds for the current parameters.

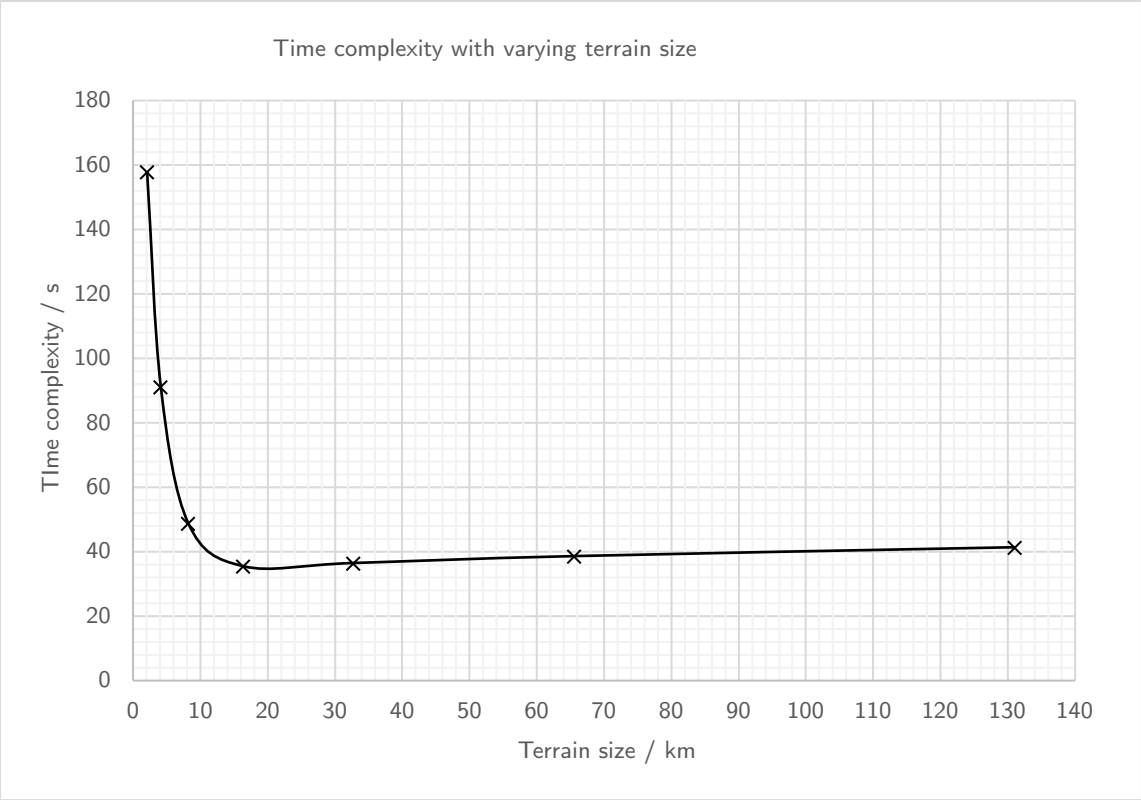


Figure 5.30: A graph showing how the time complexity of the system changed with terrain size.

5.4.2 Memory Complexity

The memory complexity of the system was measured using Valgrind. It reports on how many bytes of data was allocated on the heap.

Independent variable: Number of vertices $|V|$ in stream graph G.

Dependent variable: The memory usage of the system.

The results for the total memory allocation on the heap for different number of vertices are shown in Figure 5.31. A linear memory usage was expected. However, the results show that the memory increases linearly, but then suddenly 'steps' up and then continues linearly. To investigate this further, the number of allocs/frees were counted using Valgrind (Figure 5.32. This is linear in the number of vertices. The reason for this discrepancy may be to do with a C++ library (e.g. vector, queue) optimising its dynamic memory allocation.

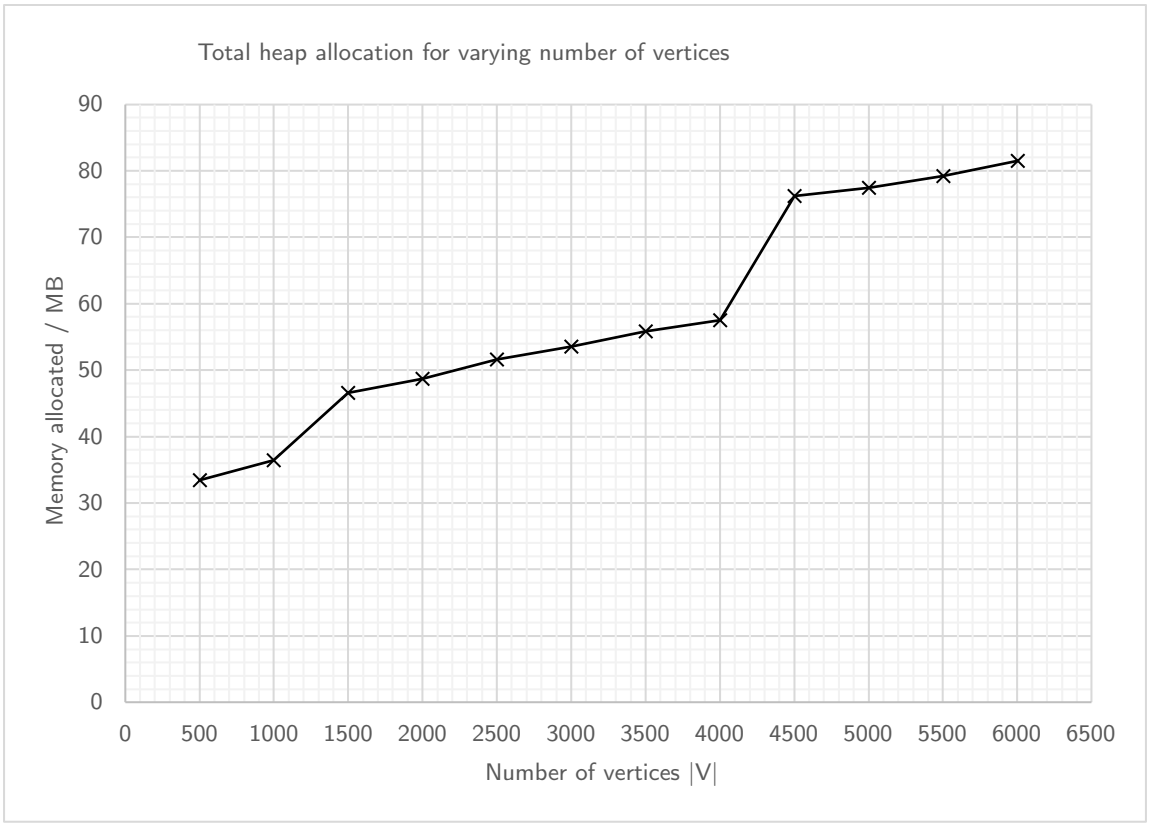


Figure 5.31: A graph showing how the memory complexity of the system changes with $|V|$.

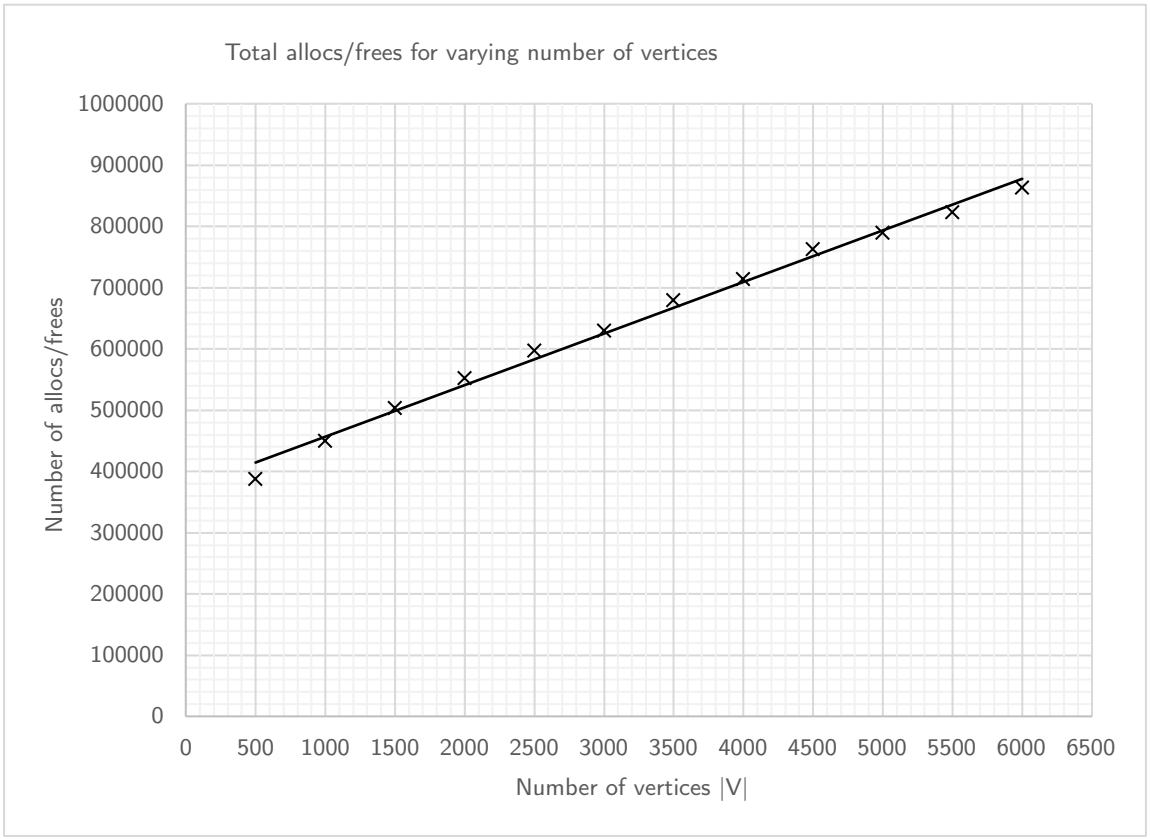


Figure 5.32: A graph showing the number of memory allocations for different $|V|$.

5.4.3 Convergence

With enough iterations of the simulation, the model will converge, with the heights of all the vertices in the stream graph approaching a final value, where the uplift is perfectly balanced by the erosion. The number of iterations needed is dependent on the size of the simulation time step δt and the number of vertices $|V|$. For all tests so far, the δt has been fixed at 250000 years.

Effect of $|V|$ on convergence

Figure 5.33 shows how the maximum heights of the terrain changes over each iteration for different $|V|$. The heights increase and then decrease before converging at a lower height. Here the maximum height converging is used as a proxy for the entire model converging. Lower number of vertices converge faster, as shown by some of the lines flattening off early. However, the lines which take longer to converge (larger $|V|$), converge at a higher elevation than the ones which converge faster. The height at which the lines converge, itself converges to a common value as $|V|$ is increased (around 3200m in this example.)

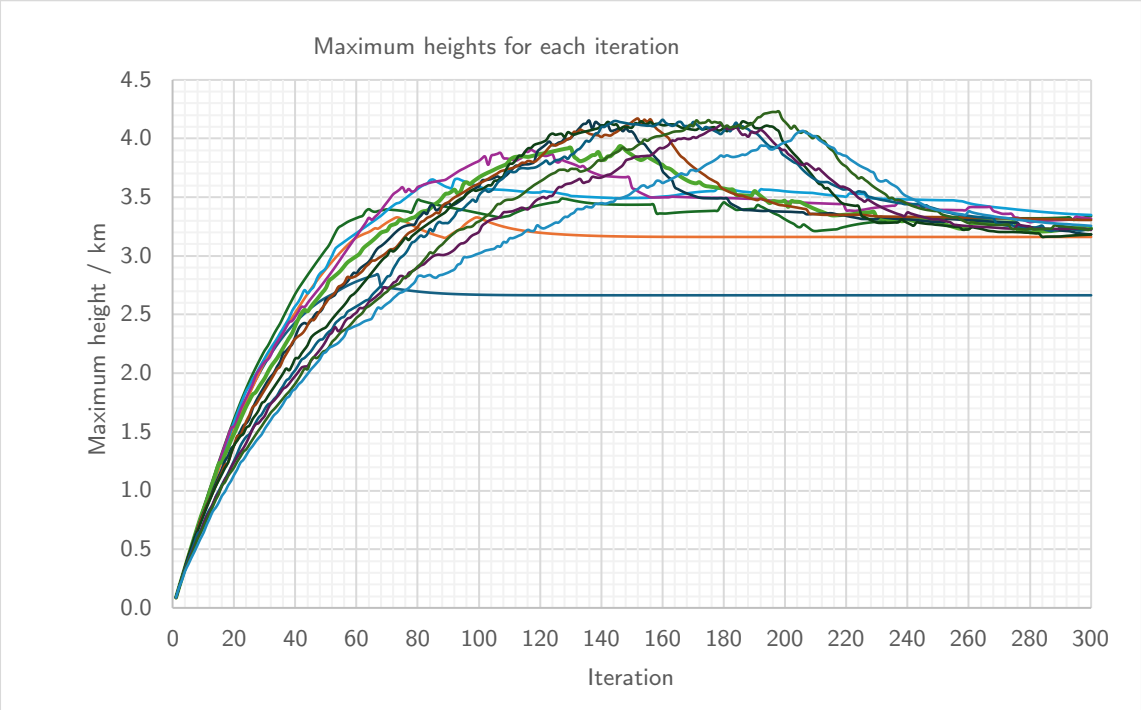


Figure 5.33: A graph of the maximum heights of the terrain at different iterations for various number of vertices.

Effect of δt on convergence

The convergence of the model was tested for different sized time steps δt, with results shown in Figure 5.34. Smaller time steps meant the model was slower to converge. When increasing δt, the model converges faster, but never faster than around 150 iterations, no matter how large δt is. Large δt values were expected to produce high unnatural cliffs (Cordonnier et al., 2016), but very large time steps were used here which resulted in no such errors.

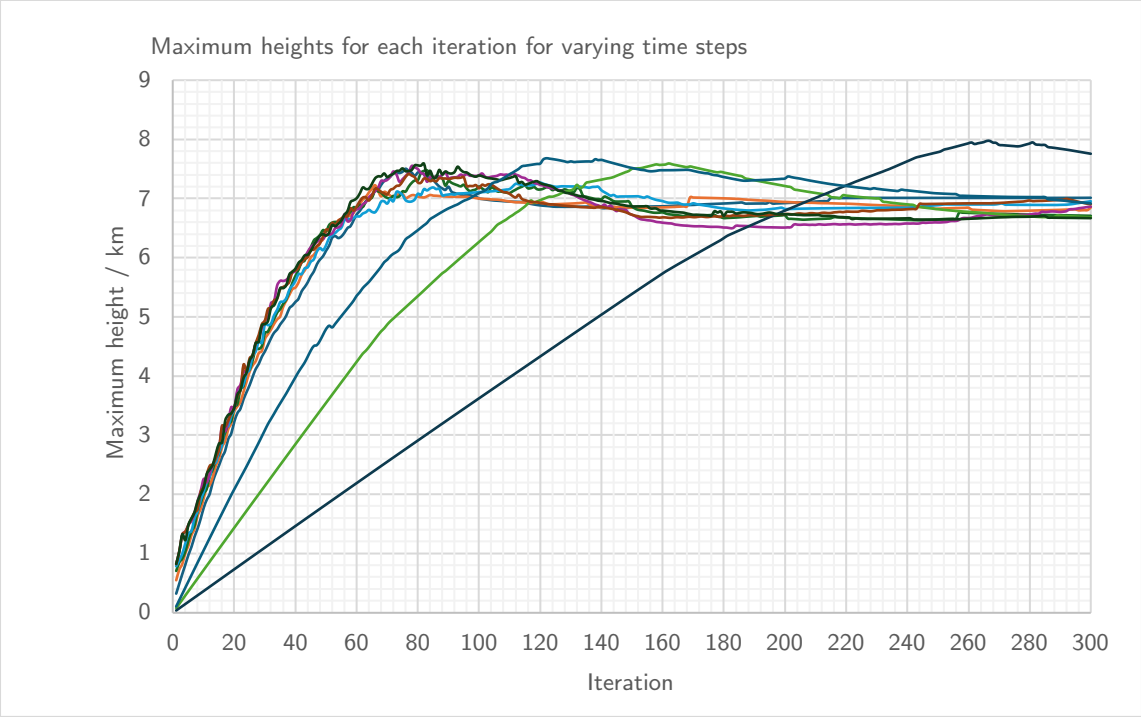


Figure 5.34: A graph of the maximum heights of the terrain at different iterations for various sized time steps.

Chapter 6

Conclusions

6.1 Achievements

Overall, the project was a success, creating a system which meets the project aim and objectives:

1. Algorithms for procedural and example-based terrain generation were researched well.
2. Algorithms for geomorphological simulation were researched thoroughly.
3. A state-of-the-art algorithm (Cordonnier et al., 2016) was used as the basis for the implemented system.
4. The system was evaluated comprehensively on the variety of landforms it can generate, the realism of the terrains, the system's performance and the system's degree of authoring.
5. The limitations of the system have only been touched upon so far, so is discussed further in the next section.

6.2 Limitations and Future Work

Despite good performance on large terrains compared to other simulation methods, the algorithm is slower than desired. It has $O(n^2)$ time complexity, which becomes slow with a large number of vertices, n. A faster implementation may be possible by creating a parallel version of the lake overflow algorithm.

A significant factor in fluvial erosion is sediment deposition, which is not currently part of the model. This could included by modelling it in a similar way to how water discharge is modelled by the stream power equation.

A large assumption in the algorithm presented here is that water will only flow in one route, to the lowest adjacent vertex. However, this means that multi-channel flow cannot happen in the model. Simulation of multichannel flow is something which could be combined with the stream power law to model new landforms such as deltas.

Another area of future work in this research area is in increasing the degree of authoring in the uplift domain. This has recently started to be explored in (Schott et al., 2023). The uplift could also simulated from the simulation of tectonic plates, such as in (Cordonnier et al., 2017).

Finally, there are many other natural processes which form terrains which are not modelled by the stream power law, such as glacial erosion forming U-shaped valleys (Figure 6.1) and coastal erosion processes. Appropriate geological models for these processes could used to generate a wider variety of terrains.

Figure 6.1: Example of a U-shaped valley Glen Avon in the Cairngorms. Photograph taken by the author.

Number of words until this point, excluding front matter: 10,568.

Bibliography

Anh, N.H., Sourin, A. and Aswani, P., 2007. Physically based hydraulic erosion simulation on graphics processing unit [Online]. *Proceedings of the 5th international conference on computer graphics and interactive techniques in australia and southeast asia*. New York, NY, USA: Association for Computing Machinery, GRAPHITE '07, p.257–264. Available from: https://doi.org/10.1145/1321261.1321308.

Beneš, B., 2007. Real-Time Erosion Using Shallow Water Simulation [Online]. In: J. Dingliana and F. Ganovelli, eds. *Workshop in virtual reality interactions and physical simulation "vriphys" (2007)*. The Eurographics Association. Available from: https://doi.org/10.2312/PE/vriphys/vriphys07/043-050.

Beneš, B. and Arriaga, X., 2005. Table mountains by virtual erosion [Online]. In: P. Poulin and E. Galin, eds. *Eurographics workshop on natural phenomena*. The Eurographics Association. Available from: https://doi.org/10.2312/NPH/NPH05/033-039.

Beneš, B. and Forsbach, R., 2001. Layered data representation for visual simulation of terrain erosion [Online]. *Proceedings spring conference on computer graphics*. pp.80–86. Available from: https://doi.org/10.1109/SCCG.2001.945341.

Beneš, B. and Forsbach, R., 2002. Visual simulation of hydraulic erosion. *Journal of wscg*, 10(1), pp.79–86.

Beneš, B., Těšínský, V., Hornyš, J. and Bhatia, S.K., 2006. Hydraulic erosion. *Computer animation and virtual worlds* [Online], 17(2), pp.99–108. Available from: https://doi.org/10.1002/cav.77.

Braun, J. and Willett, S.D., 2013. A very efficient o(n), implicit and parallel method to solve the stream power equation governing fluvial incision and landscape evolution. *Geomorphology* [Online], 180-181, pp.170–179. Available from: https://doi.org/https://doi.org/10.1016/j.geomorph.2012.10.008.

Bridson, R., 2007. Fast poisson disk sampling in arbitrary dimensions [Online]. *Acm siggraph 2007 sketches*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '07, p.22–es. Available from: https://doi.org/10.1145/1278780.1278807.

Chiba, N., Muraoka, K. and Fujita, K., 1998. An erosion model based on velocity fields for the visual simulation of mountain scenery. *The journal of visualization and computer animation*, 9(4), pp.185–194.

Cordonnier, G., Braun, J., Cani, M.P., Benes, B., Galin, E., Peytavie, A. and Guérin, E., 2016. Large scale terrain generation from tectonic uplift and fluvial erosion. *Computer*

graphics forum [Online], 35(2), pp.165–175. Available from: https://doi.org/10.1111/cgf.12820.

Cordonnier, G., Cani, M., Benes, B., Braun, J. and Galin, E., 2018. Sculpting mountains: Interactive terrain modeling based on subsurface geology. *Ieee transactions on visualization and computer graphics* [Online], 24(05), pp.1756–1769. Available from: https://doi.org/10.1109/TVCG.2017.2689022.

Cordonnier, G., Galin, E., Gain, J., Benes, B., Guérin, E., Peytavie, A. and Cani, M.P., 2017. Authoring landscapes by combining ecosystem and terrain erosion simulation. *Acm trans. graph.* [Online], 36(4). Available from: https://doi.org/10.1145/3072959.3073667.

Cordonnier, G., Jouvet, G., Peytavie, A., Braun, J., Cani, M.P., Benes, B., Galin, E., Guérin, E. and Gain, J., 2023. Forming terrains by glacial erosion. *Acm trans. graph.* [Online], 42(4). Available from: https://doi.org/10.1145/3592422.

Ebert, D.S., Musgrave, F.K., Peachey, D., Perlin, K. and Worley, S., 2003. *Texturing and modeling : a procedural approach*. 3rd ed. Amsterdam ; Boston: Academic Press.

Fournier, A., Fussell, D. and Carpenter, L., 1982. Computer rendering of stochastic models. *Commun. acm* [Online], 25(6), p.371–384. Available from: https://doi.org/10.1145/358523.358553.

Galin, E., Guérin, E., Peytavie, A., Cordonnier, G., Cani, M.P., Benes, B. and Gain, J., 2019. A review of digital terrain modeling. *Computer graphics forum* [Online], 38(2), pp.553–577. Available from: https://doi.org/10.1111/cgf.13657.

Génevaux, J.D., Galin, E., Guérin, E., Peytavie, A. and Benes, B., 2013. Terrain generation using procedural models based on hydrology. *Acm trans. graph.* [Online], 32(4). Available from: https://doi.org/10.1145/2461912.2461996.

Guérin, E., Digne, J., Galin, E., Peytavie, A., Wolf, C., Benes, B. and Martinez, B., 2017. Interactive example-based terrain authoring with conditional generative adversarial networks. *Acm transactions on graphics* [Online], 36. Available from: https://doi.org/10.1145/3130800.3130804.

Ito, T., Fujimoto, T., Muraoka, K. and Chiba, N., 2003. Modeling rocky scenery taking into account joints [Online]. *Proceedings computer graphics international 2003*. pp.244–247. Available from: https://doi.org/10.1109/CGI.2003.1214475.

Kelley, A.D., Malin, M.C. and Nielson, G.M., 1988. Terrain simulation using a model of stream erosion [Online]. *Proceedings of the 15th annual conference on computer graphics and interactive techniques*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '88, p.263–268. Available from: https://doi.org/10.1145/54852.378519.

Krištof, P., Beneš, B., Křivánek, J. and Šťava, O., 2009. Hydraulic erosion using smoothed particle hydrodynamics. *Computer graphics forum* [Online], 28(2), pp.219–228. Available from: https://doi.org/10.1111/j.1467-8659.2009.01361.x.

Lague, D., 2014. The stream power river incision model: evidence, theory and beyond. *Earth surface processes and landforms* [Online], 39(1), pp.38–61. Available from: https://doi.org/https://doi.org/10.1002/esp.3462.

Lorensen, W.E. and Cline, H.E., 1987. Marching cubes: A high resolution 3d surface

construction algorithm [Online]. *Proceedings of the 14th annual conference on computer graphics and interactive techniques*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '87, p.163–169. Available from: https://doi.org/10.1145/37401.37422.

Mandelbrot, B.B., 1982. *The fractal geometry of nature*. Updated and augmented. ed. Oxford: Freeman.

Mandelbrot, B.B. and Ness, J.W.V., 1968. Fractional brownian motions, fractional noises and applications. *Siam review*, 10(4), pp.422–437.

Mei, X., Decaudin, P. and Hu, B.G., 2007. Fast hydraulic erosion simulation and visualization on gpu [Online]. *15th pacific conference on computer graphics and applications (pg'07)*. pp.47–56. Available from: https://doi.org/10.1109/PG.2007.15.

Michel, E., Emilien, A. and Cani, M.P., 2015. Generation of Folded Terrains from Simple Vector Maps [Online]. In: B. Bickel and T. Ritschel, eds. *Eg 2015 - short papers*. The Eurographics Association. Available from: https://doi.org/10.2312/egsh.20151019.

Miller, G.S.P., 1986. The definition and rendering of terrain maps [Online]. *Proceedings of the 13th annual conference on computer graphics and interactive techniques*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '86, p.39–48. Available from: https://doi.org/10.1145/15922.15890.

Musgrave, F.K., Kolb, C.E. and Mace, R.S., 1989. The synthesis and rendering of eroded fractal terrains [Online]. *Proceedings of the 16th annual conference on computer graphics and interactive techniques*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '89, p.41–50. Available from: https://doi.org/10.1145/74333.74337.

Nagashima, K., 1998. Computer generation of eroded valley and mountain terrains. *The visual computer*, 9(13), pp.456–464.

Neidhold, B., Wacker, M. and Deussen, O., 2005. Interactive physically based Fluid and Erosion Simulation [Online]. In: P. Poulin and E. Galin, eds. *Eurographics workshop on natural phenomena*. The Eurographics Association. Available from: https://doi.org/10.2312/NPH/NPH05/025-032.

O'Brien, J. and Hodgins, J., 1995. Dynamic simulation of splashing fluids. *Proceedings computer animation'95*. Ithaca: IEEE Comput. Soc. Press, pp.198–205.

Onoue, K. and Nishita, T., 2000. A method for modeling and rendering dunes with wind-ripples [Online]. *Proceedings the eighth pacific conference on computer graphics and applications*. pp.427–428. Available from: https://doi.org/10.1109/PCCGA.2000.883978.

Perlin, K., 1985. An image synthesizer [Online]. *Proceedings of the 12th annual conference on computer graphics and interactive techniques*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '85, p.287–296. Available from: https://doi.org/10.1145/325334.325247.

Perlin, K., 2002. Improving noise. *Acm trans. graph.* [Online], 21(3), p.681–682. Available from: https://doi.org/10.1145/566654.566636.

Peytavie, A., Dupont, T., Guérin, E., Cortial, Y., Benes, B., Gain, J. and Galin, E., 2019.

Procedural riverscapes. *Computer graphics forum* [Online], 38(7), pp.35–46. Available from: https://doi.org/10.1111/cgf.13814.

Peytavie, A., Galin, E., Grosjean, J. and Merillou, S., 2009. Arches: a framework for modeling complex terrains. *Computer graphics forum* [Online], 28(2), pp.457–467. Available from: https://doi.org/10.1111/j.1467-8659.2009.01385.x.

Rosetta Code, 2024. *Perlin noise - rosetta code.* [Online]. Available from: https://rosettacode.org/wiki/Perlin_noise [Accessed 2024-01-18].

Rosgen, D.L., 1994. A classification of natural rivers. *Catena* [Online], 22(3), pp.169–199. Available from: https://doi.org/https://doi.org/10.1016/0341-8162(94)90001-9.

Roudier, P., Peroche, B. and Perrin, M., 1993. Landscapes synthesis achieved through erosion and deposition process simulation. *Computer graphics forum* [Online], 12(3), pp.375–383. Available from: https://doi.org/10.1111/1467-8659.1230375.

Schott, H., Paris, A., Fournier, L., Guérin, E. and Galin, E., 2023. Large-scale terrain authoring through interactive erosion simulation. *Acm trans. graph.* [Online], 42(5). Available from: https://doi.org/10.1145/3592787.

Stachniak, S. and Stuerzlinger, W., 2005. An algorithm for automated fractal terrain deformation. *Computer graphics and artificial intelligence*, 1, pp.64–76.

Šťava, O., Beneš, B., Brisbin, M. and Křivánek, J., 2008. Interactive terrain modeling using hydraulic erosion. *Proceedings of the 2008 acm siggraph/eurographics symposium on computer animation*. pp.201–210.

Sutherland, B. and Keyser, J., 2006. Particle-based enhancement of terrain data [Online]. *Acm siggraph 2006 research posters*. New York, NY, USA: Association for Computing Machinery, SIGGRAPH '06, p.96–es. Available from: https://doi.org/10.1145/1179622.1179732.

United States Geological Survey, 2021. *United states geological survey (2021). united states geological survey 3d elevation program 1/3 arc-second digital elevation model.* [Online]. Available from: https://doi.org/10.5069/G98K778D [Accessed 2024-05-02].

Voss, R.F., 1991. Random fractal forgeries. *Fundamental algorithms for computer graphics: Nato advanced study institute*. Springer, pp.805–835.

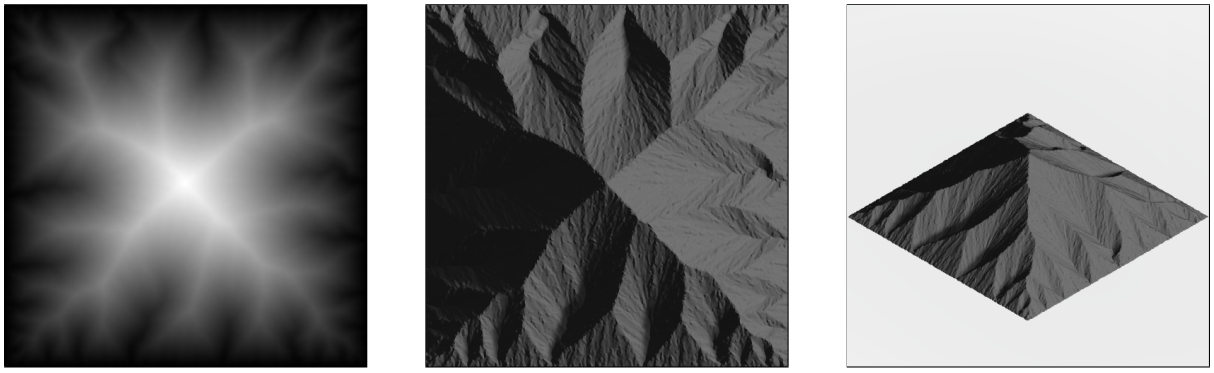
Whipple, K.X. and Tucker, G.E., 1999. Dynamics of the stream-power river incision model: Implications for height limits of mountain ranges, landscape response timescales, and research needs. *Journal of geophysical research: Solid earth* [Online], 104(B8), pp.17661–17674. Available from: https://doi.org/10.1029/1999JB900120.

Zhou, H., Sun, J., Turk, G. and Rehg, J.M., 2007. Terrain synthesis from digital elevation models. *Ieee transactions on visualization and computer graphics* [Online], 13(4), pp.834–848. Available from: https://doi.org/10.1109/TVCG.2007.1027.

Appendix A

Parameter Analysis Results

A.1 Terrain Size



(a) Heightfield (b) Orthographic top view. (c) Isometric view.

Figure A.1: 4096m x 4096m terrain size.

(a) Heightfield (b) Orthographic top view. (c) Isometric view.

Figure A.2: 8192m x 8192m terrain size.

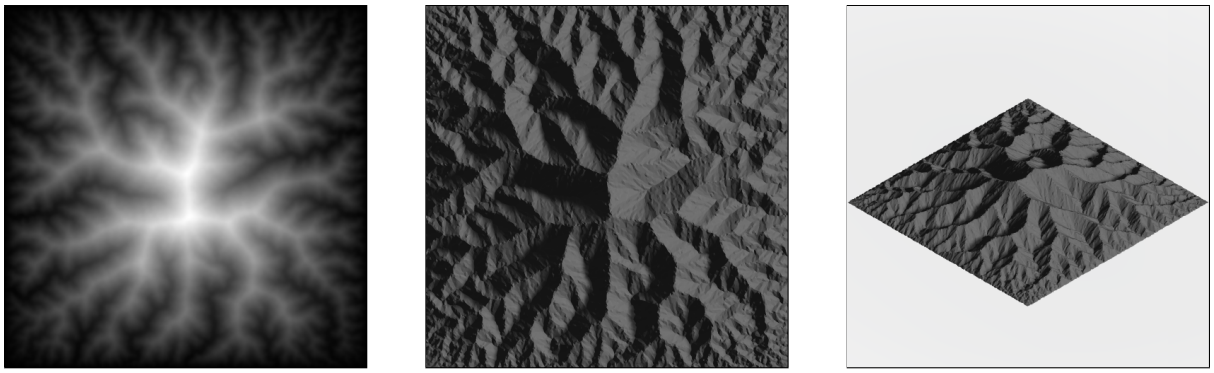


Figure A.3: 16384m x 16384m terrain size.

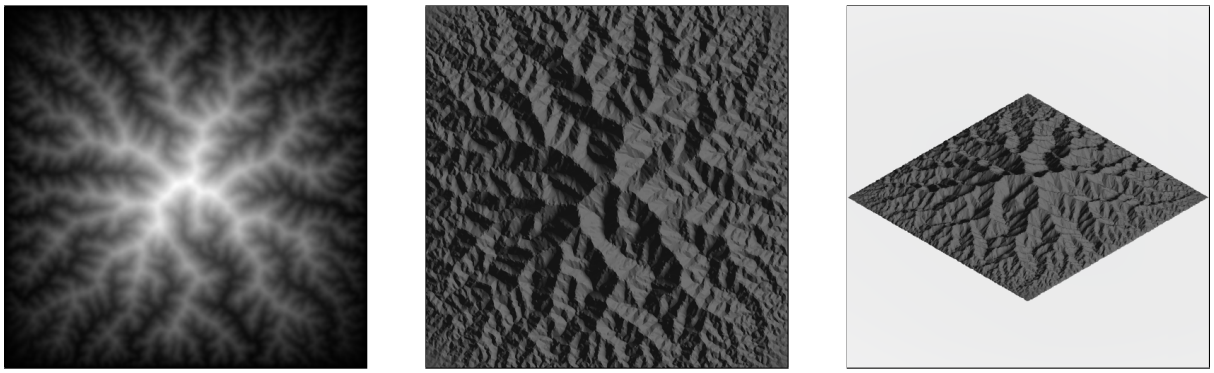


Figure A.4: 32768m x 32768m terrain size.

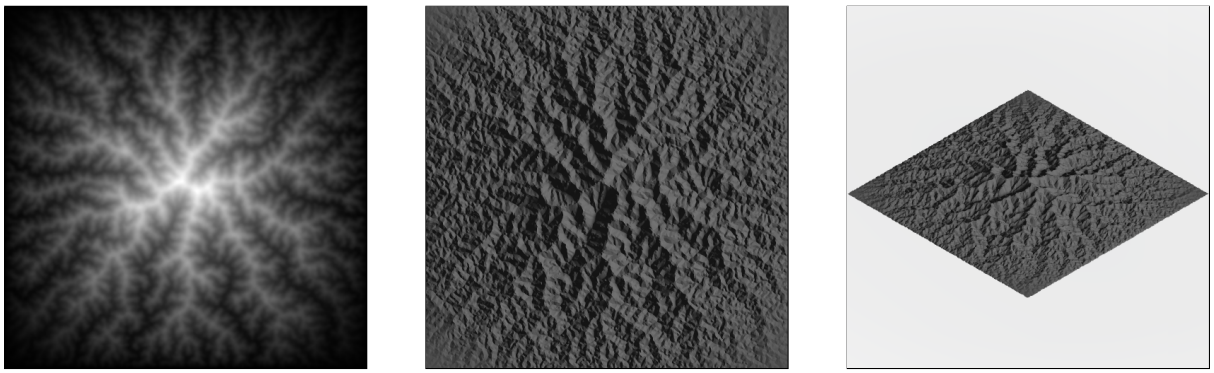


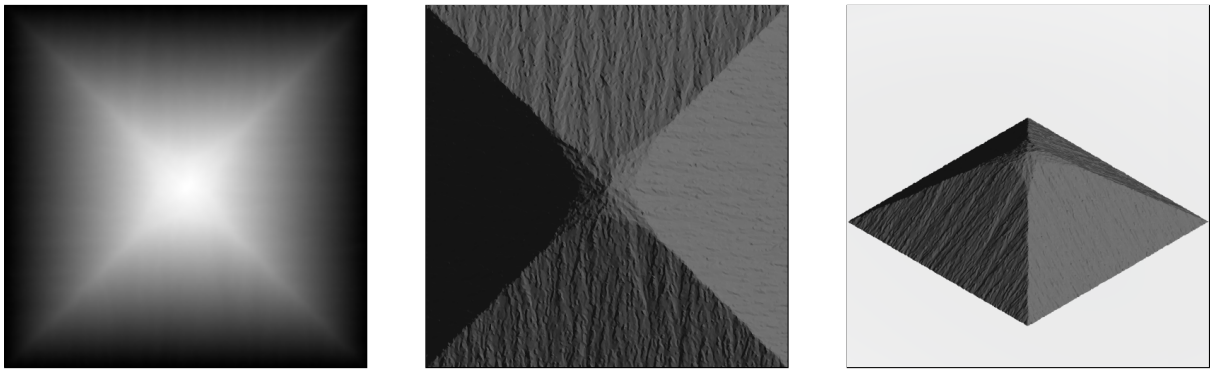
Figure A.5: 65536m x 65536m terrain size.

A.2 Stream Power Parameters

A.2.1 m

m	Maximum height / m
0.1	11025.7
0.25	11986.6
0.4	9070.26
0.45	5456.45
0.475	4402.73
0.5	3235.46
0.525	2407.12
0.55	1771.22
0.6	1097.88
0.75	181.337
0.9	29.6928

Table A.1: Table of values for analysis of m parameter.



(a) Heightfield (b) Orthographic top view. (c) Isometric view.

Figure A.6: m=0.1, maximum height=11025m

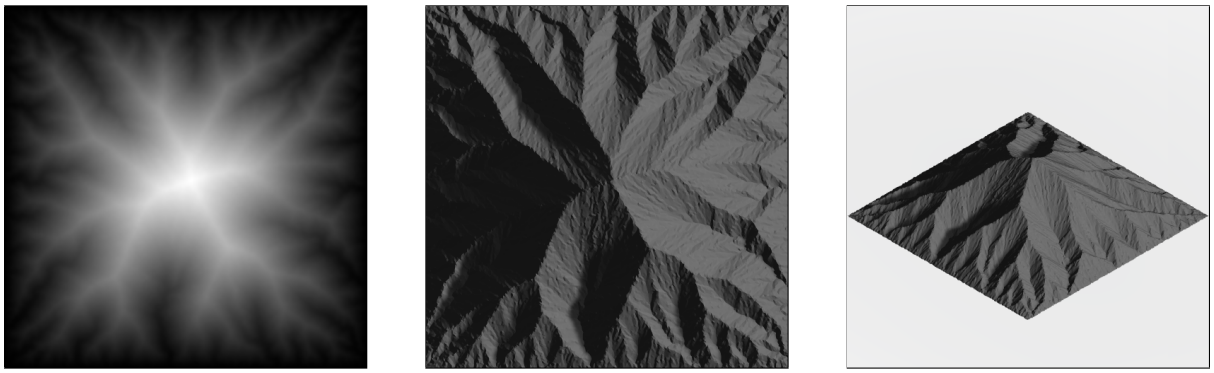


Figure A.7: m=0.4, maximum height=9070m

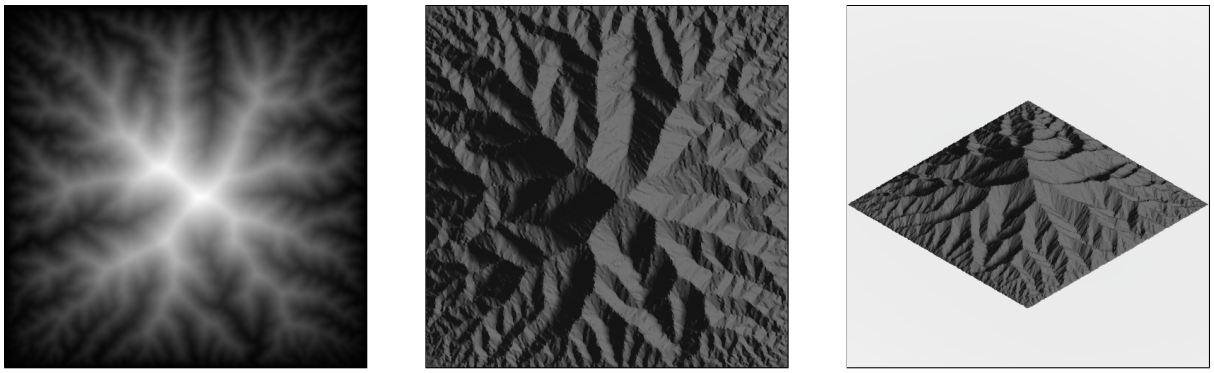


Figure A.8: m=0.45, maximum height=5456m

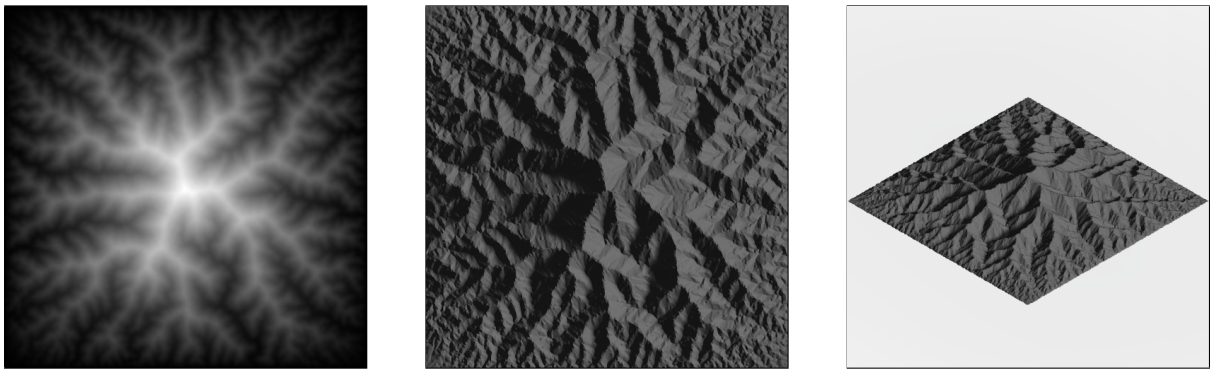


Figure A.9: m=0.475, maximum height=4402m

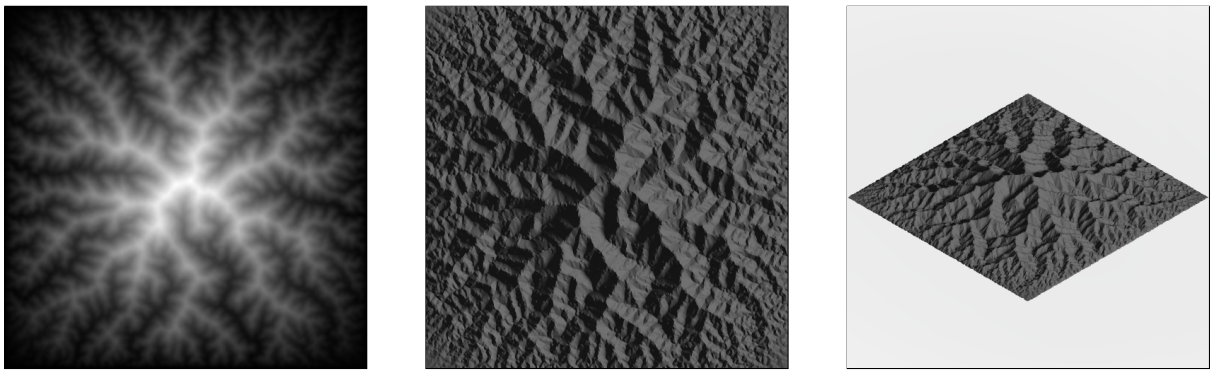


Figure A.10: m=0.5, maximum height=3235m

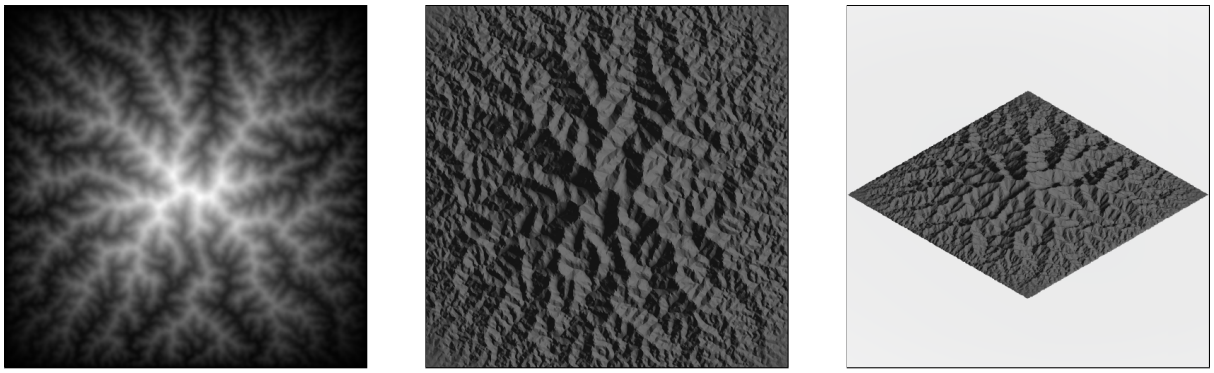


Figure A.11: m=0.525, maximum height=2407m

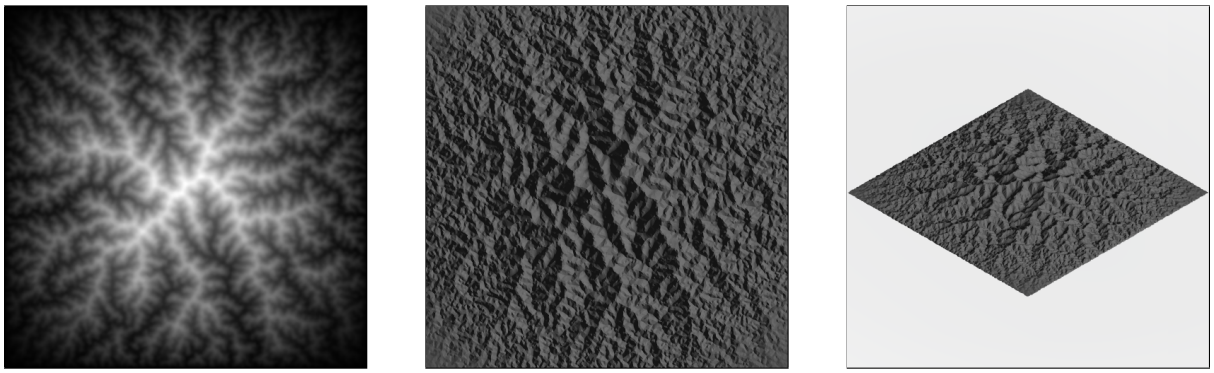


Figure A.12: m=0.55, maximum height=1771m

Figure A.13: m=0.6, maximum height=1098m

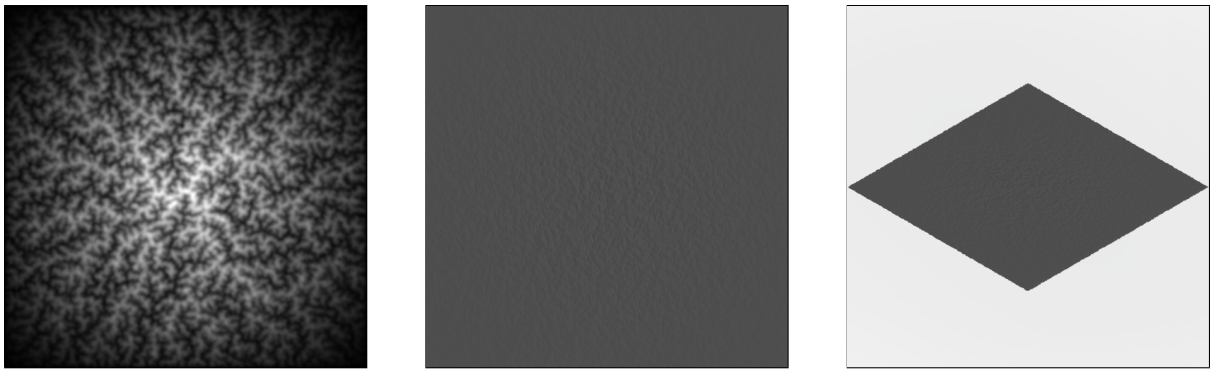


Figure A.14: m=0.9, maximum height=30m

A.2.2 Uplift Rate

Uplift / (m/y)	Maximum height / m
0.000025	329.733
0.00005	700.752
0.0001	1318.4
0.00025	2430.02
0.0004	3235.46
0.0005	3917.03
0.0006	4234.99
0.00075	4595.85
0.0009	5128.45
0.001	5678.49
0.0015	6664.74
0.002	7899.32

Table A.2: Table of values for analysis of uplift rate.

A.2.3 Erosion Constant (k)

k / (1/y)	Maximum height / m	1 / (Maximum height)
0.00000005	10881.5	0.000091899
0.0000003	4834.19	0.00020686
0.0000005	3661.59	0.000273105
0.0000006	3162.46	0.00031621
0.0000008	2549.92	0.000392169
0.000001	2169.5	0.00046094
0.0000012	1922.36	0.000520194
0.0000015	1637.59	0.000610653

Table A.3: Table of values for analysis of erosion constant (k) parameter.

(a) Heightfield (b) Orthographic top view. (c) Isometric view.

Figure A.15: k=0.00000005 per year, maximum height=10882m

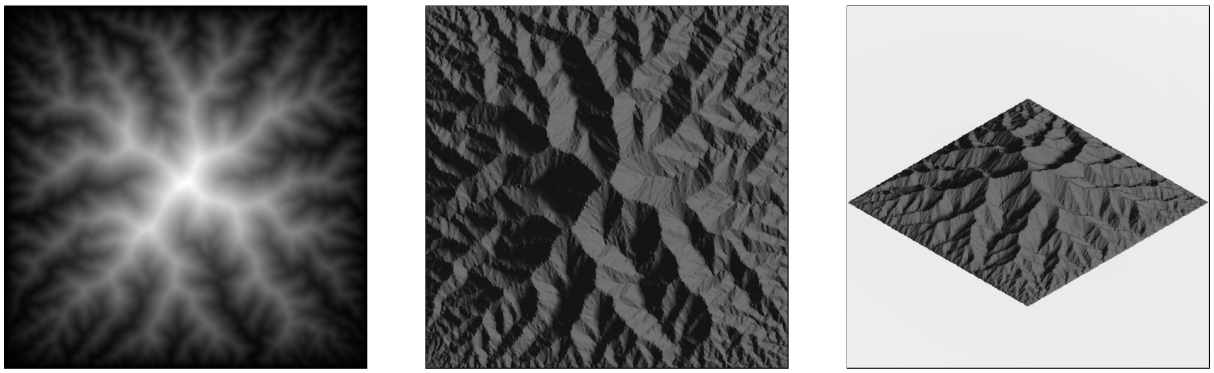


Figure A.16: k=0.0000003 per year, maximum height=4834m

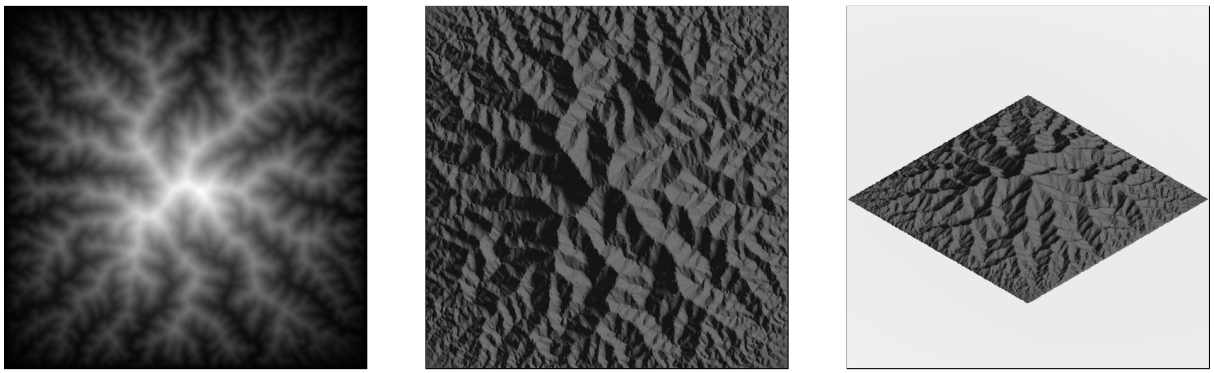


Figure A.17: k=0.0000005 per year, maximum height=3662m

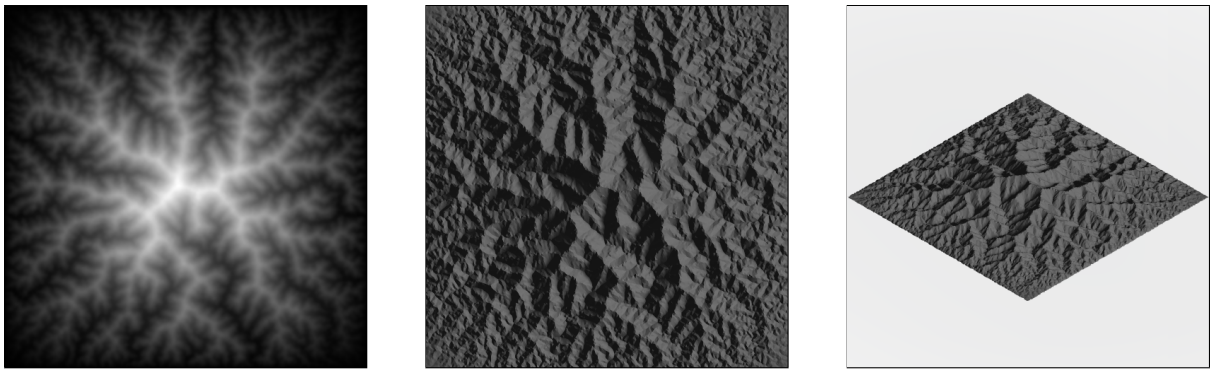


Figure A.18: k=0.0000006 per year, maximum height=3162m

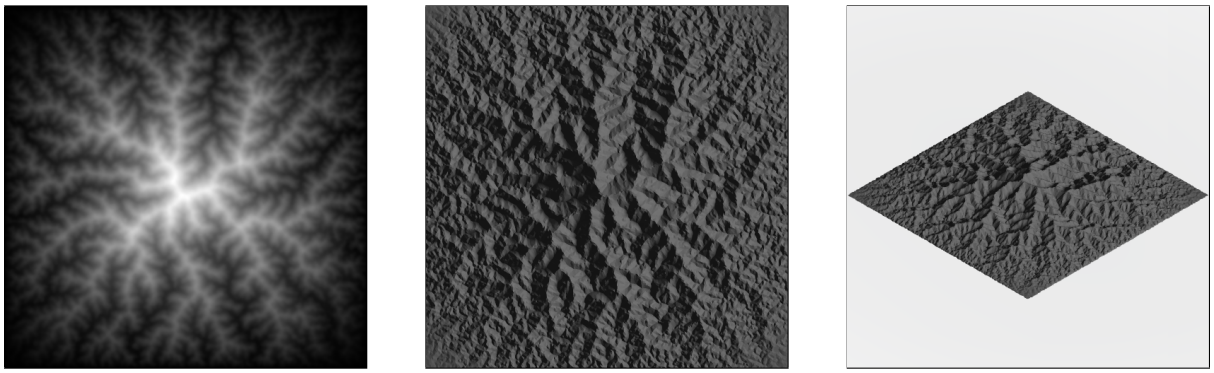


Figure A.19: k=0.0000008 per year, maximum height=2550m

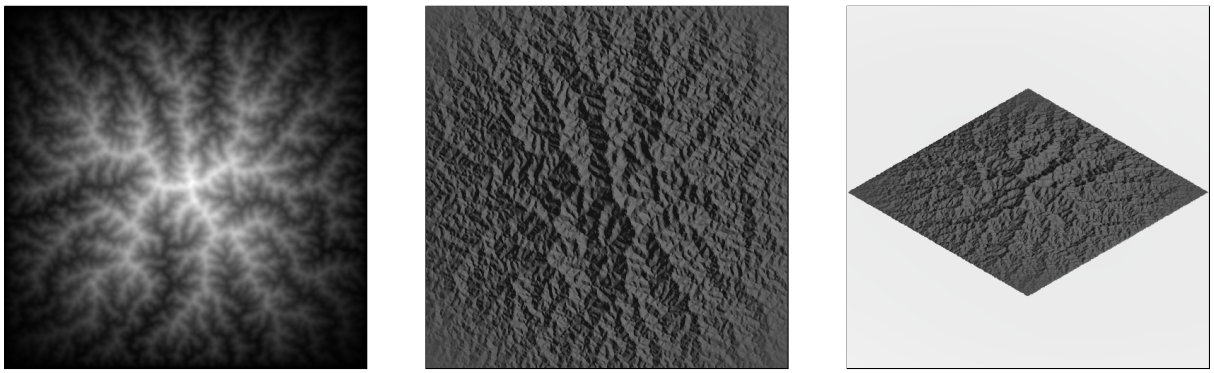
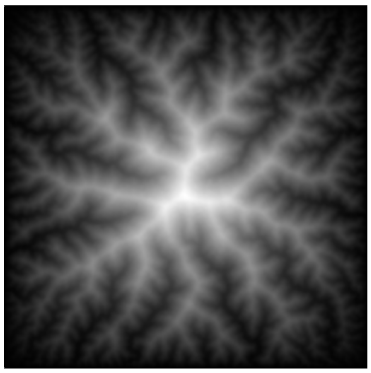


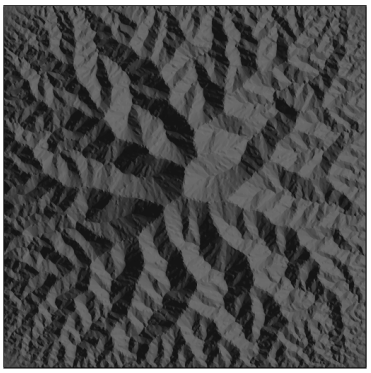
Figure A.20: k=0.0000015 per year, maximum height=1638m

A.3 Thermal Erosion

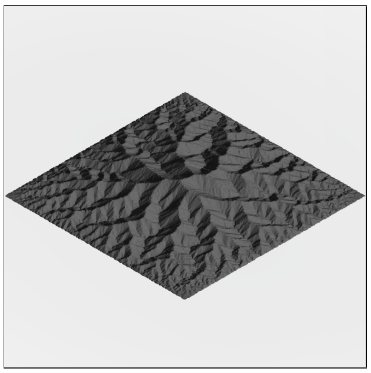
A.3.1 Talus Angle



(a) Heightfield

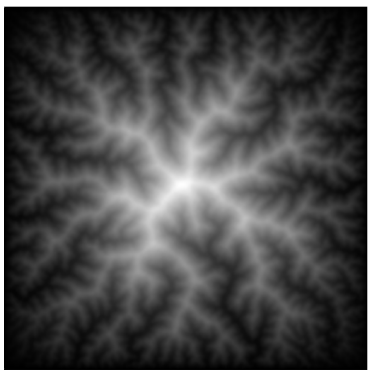


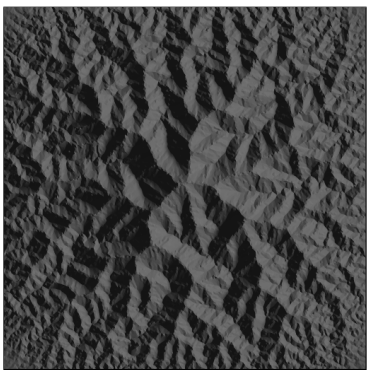
(b) Orthographic top view.



(c) Isometric view.

Figure A.21: α=25 degrees





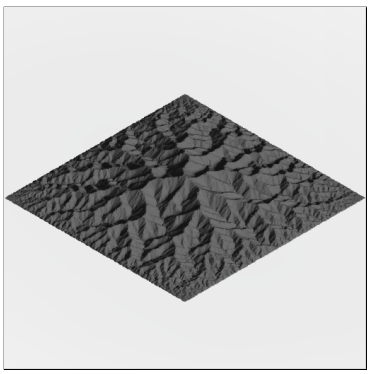
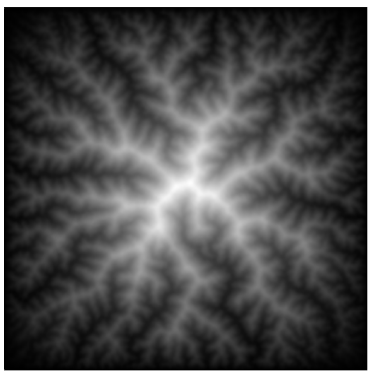
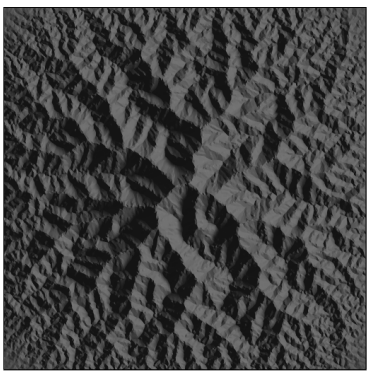


Figure A.22: α=30 degrees





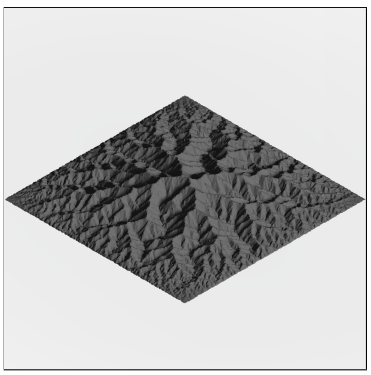


Figure A.23: α=35 degrees

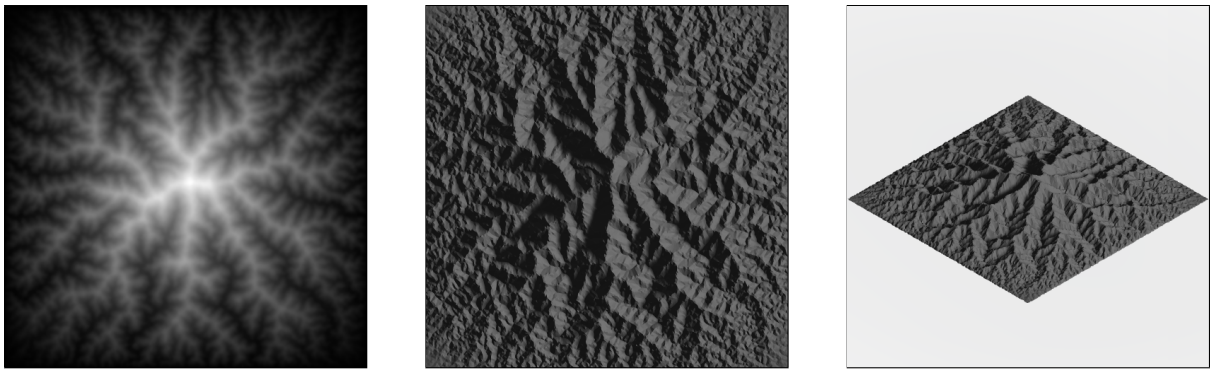


Figure A.24: α=40 degrees

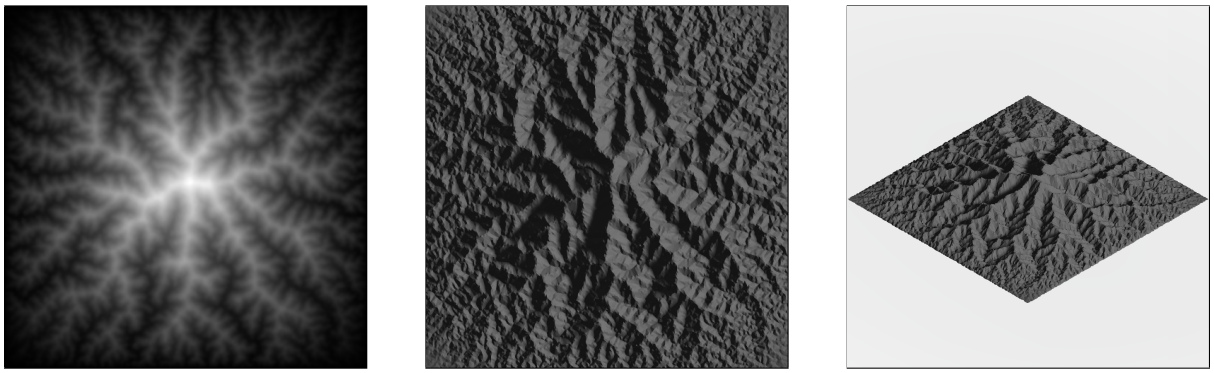


Figure A.25: α=45 degrees

Appendix B

Performance Results

B.1 Time Complexity Results

Vertex count	Time / s
1018	0.4431
2032	1.1508
5040	3.3412
10035	7.7410
20031	19.4900
30014	37.4151
40014	59.5623
49986	85.6297
59934	122.8690
69849	150.2990
79808	192.5560
89870	237.0460
99927	294.3030

Table B.1: Table of values used for analysis of time complexity with respect to number of vertices.

Terrain size / m	Time / s
2048	157.836
4096	91.0184
8192	48.6948
16384	35.4402
32768	36.4467
65536	38.5874
131072	41.338

Table B.2: Table of values used for analysis of time complexity with respect to size of terrain.

B.2 Memory Complexity Results

Vertex count	Memory allocated / Bytes	Number of allocs/frees
500	33462903	387667
1000	36425015	450258
1500	46554519	503986
2000	48662059	552120
2500	51636259	597143
3000	53573035	629768
3500	55868207	680015
4000	57514099	714355
4500	76192635	762669
5000	77449311	789694
5500	79175155	823576
6000	81481551	863441

Table B.3: Table of values used for analysis of memory complexity with respect to number of vertices.

Appendix C

Code Listings

C.1 Core Functions Code

C.1.1 vector.h

```
#ifndef VECTOR_H_
#define VECTOR_H_

#include <math.h>

class Vector{
public:
  double x;
  double y;
  double z;

  Vector(double a, double b, double c){
    x = a;
    y = b;
    z = c;
  }

    Vector(double a, double b){
    x = a;
    y = b;
    z = 0.0;
  }

  Vector(){
    x = 0.0;
    y = 0.0;
    z = 0.0;
  }

  double lengthSquared(){
    return x * x + y * y + z * z ;
  }

  double length(){
    return sqrt(lengthSquared());
  }

  Vector operator+(const Vector v){
    Vector result;
    result.x = this->x + v.x;
    result.y = this->y + v.y;
    result.z = this->z + v.z;
    return result;
  }

  Vector operator-(){
    Vector result;
    result.x = -this->x;
    result.y = -this->y;
    result.z = -this->z;
    return result;
  }

  Vector operator*(double scalar){
    Vector result;
    result.x = scalar * this->x;
    result.y = scalar * this->y;
    result.z = scalar * this->z;
    return result;
  }

  friend Vector operator-(Vector a, Vector b){
```

```
    Vector result;
    result.x = a.x - b.x;
    result.y = a.y - b.y;
    result.z = a.z - b.z;
    return result;
  }

  friend Vector operator*(double scalar, Vector v){
    Vector result;
    result.x = scalar * v.x;
    result.y = scalar * v.y;
    result.z = scalar * v.z;
    return result;
  }
};

#endif
```

C.1.2 noise.h

```
#ifndef NOISE_H_
#define NOISE_H_

#include "vector.h"

void loadNoisePermutation(char* fileName);
double noise3D(double x, double y, double z);
double noise2D(double x, double y);

double perlinNoise(Vector p, int octaves, double lacunarity, double persistence, double scale);
double billowNoise(Vector p, int octaves, double lacunarity, double persistence, double scale);
double ridgeNoise(Vector p, int octaves, double lacunarity, double persistence, double scale);

double warpedNoise(Vector warp, double warpScale, Vector p, int octaves, double lacunarity, double persistence, double
    ↪  scale);

double randRange(double min, double max);

#endif
```

C.1.3 noise.cpp

```
// This code is adapted from the original Java Code by Ken Perlin (2002)
// Which can be found here: https://cs.nyu.edu/~perlin/noise/

// The translated c code is from https://rosettacode.org/wiki/Perlin_noise

#include<stdlib.h>
#include<stdio.h>
#include<math.h>

#include "noise.h"

int p[512];

double fade(double t) { return t * t * t * (t * (t * 6 - 15) + 10); }
double lerp(double t, double a, double b) { return a + t * (b - a); }
double grad(int hash, double x, double y, double z) {
      int h = hash & 15;
      double u = h<8 ? x : y,
             v = h<4 ? y : h==12||h==14 ? x : z;
      return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
   }

double noise3D(double x, double y, double z) {
      int X = (int)floor(x) & 255,
          Y = (int)floor(y) & 255,
          Z = (int)floor(z) & 255;
      x -= floor(x);
      y -= floor(y);
      z -= floor(z);
      double u = fade(x),
             v = fade(y),
             w = fade(z);
      int A = p[X  ]+Y, AA = p[A]+Z, AB = p[A+1]+Z,
          B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z;

      return lerp(w, lerp(v, lerp(u, grad(p[AA  ], x  , y  , z   ),
                                     grad(p[BA  ], x-1, y  , z   )),
                             lerp(u, grad(p[AB  ], x  , y-1, z   ),
                                     grad(p[BB  ], x-1, y-1, z   ))),
                     lerp(v, lerp(u, grad(p[AA+1], x  , y  , z-1 ),
                                     grad(p[BA+1], x-1, y  , z-1 )),
                             lerp(u, grad(p[AB+1], x  , y-1, z-1 ),
                                     grad(p[BB+1], x-1, y-1, z-1 ))));
}

double noise2D(double x, double y){
  return noise3D(x, y, 0.0);
}

void loadNoisePermutation(char* fileName){
  FILE* fp = fopen(fileName,"r");
  int permutation[256],i;
```

```

  for(i=0;i<256;i++)
    fscanf(fp,"%d",&permutation[i]);

  fclose(fp);

  for (int i=0; i < 256 ; i++) p[256+i] = p[i] = permutation[i];
}

// Combines layers octaves of perlin noise
double perlinNoise(Vector p, int octaves, double lacunarity, double persistence, double scale){
  double sum = 0.0;
  double frequency = scale;
  double amplitude = 1.0;

  for(int i = 0; i < octaves; i++) {
    double n = noise2D(p.x * frequency, p.y * frequency);
    sum += n * amplitude;
    frequency *= lacunarity;
    amplitude *= persistence;
  }

  return sum;
}

// Billow noise is the absolute value of perlin noise
double billowNoise(Vector p, int octaves, double lacunarity, double gain, double scale){
  return fabs(perlinNoise(p, octaves, lacunarity, gain, scale));
}

// Ride noise and billow noise sum's to 1
double ridgeNoise(Vector p, int octaves, double lacunarity, double gain, double scale){
  return 1.0f - billowNoise(p, octaves, lacunarity, gain, scale);
}

// Warped noise changes the position the noise is sampled from to create a warping effect
double warpedNoise(Vector warp, double warpScale, Vector p, int octaves, double lacunarity, double persistence, double
    ↪  scale){
    Vector offset = Vector(perlinNoise(p, octaves, lacunarity, persistence, scale),
            perlinNoise(p + warp, octaves, lacunarity, persistence, scale));

    return perlinNoise(p + offset * warpScale, octaves, lacunarity, persistence, scale);
}

double randRange(double min, double max){
    return min + (max - min) * ((double)rand() / RAND_MAX);
}
```

C.1.4 heightfield.h

```
#ifndef HEIGHTFIELD_H_
#define HEIGHTFIELD_H_

double **createHeightfield(const int size);
void freeHeightfield(double **a, const int size);
void outputHeightfieldAsImage(double **a, const int size, const double maxHeight, char *filename);
double **importImageAsHeightfield(char *filename, int *size, const double maxHeight);

#endif
```

C.1.5 heightfield.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <sstream>

// Allocates memory for a heightfield
double **createHeightfield(const int size){
    double **a = (double**)malloc(size * sizeof(double*));
    for (int i = 0; i < size; i++){
        a[i] = (double*)calloc(size, sizeof(double));
    }

    return a;
}

// Deallocates memory for a heightfield
void freeHeightfield(double **a, const int size){
    for (int i = 0; i < size; i++){
        free(a[i]);
    }
    free(a);
}

// Outputs a heightfield as a ppm image file to a specified file path
void outputHeightfieldAsImage(double **a, const int size, const double maxHeight, char *filename){
    // Adapted from https://rosettacode.org/wiki/Bitmap/Write_a_PPM_file
    FILE *imageFile = fopen(filename, "wb");
    fprintf(imageFile, "P6\n%d %d\n255\n", size, size);

    for (int z = 0; z < size; z++){
        for (int x = 0; x < size; x++){
```

```
            unsigned char color[3];

            // Normalise colour
            unsigned char value = (char)(a[z][x] * 255.0f / maxHeight);

            // Write colour to r,g and b channels.
            color[0] = value;
            color[1] = value;
            color[2] = value;
            fwrite(color, 1, 3, imageFile);
        }
    }

    fclose(imageFile);
}

// Imports a heightfield as a ppm image file
double **importImageAsHeightfield(char *filename, int *terrainSize, const double maxHeight){
    std::ifstream imageFile(filename);

    std::string lineString;

    // Check for standard header of file
    std::getline(imageFile, lineString);
    if (lineString[0] != 'P'){
        return 0;
    }

    // Ignore comments
    std::getline(imageFile, lineString);
    while (lineString[0] == '#') {
        std::getline(imageFile, lineString);
    }

    // Get the dimensions of the iamge
    std::istringstream dimensions(lineString);
    int width, height;
    dimensions >> width;
    dimensions >> height;
    if (width != height){
        return 0;
    }

    int size = width;
    double **heightfield = createHeightfield(size);

    // Get maximum colour value (normally 255)
    std::getline(imageFile, lineString);
    std::istringstream maxValueString(lineString);
    double maxImageValue;
    maxValueString >> maxImageValue;

    // Read pixels from file (as RGB, but only looking at Red channel, as R=G=B for greyscale images)
    char imageChar;
    for (int i = 0; i < size * size; i++) {
        std::getline(imageFile, lineString);
        std::istringstream valueString(lineString);
        double value;
        valueString >> value;

        // Convert colour value to height value
        double height = value * maxHeight / maxImageValue;
        heightfield[i / size][i % size] = height;
        std::getline(imageFile, lineString);
        std::getline(imageFile, lineString);
    }

    imageFile.close();

    *terrainSize = size;
    return heightfield;
}
```

C.1.6 mesh.h

```
#ifndef MESH_H_
#define MESH_H_

#include "vector.h"

typedef struct{
    unsigned int v0, v1, v2;
} Triangle;

typedef struct{
    int vertexCount;
    Vector *vertices;

    int faceCount;
    Triangle *faces;
} Mesh;

Triangle makeTriangle(unsigned int a, unsigned int b, unsigned int c);
void addVertexToMesh(Mesh *mesh, double x, double y, double z);
void addFaceToMesh(Mesh *mesh, unsigned int a, unsigned int b, unsigned int c);
Mesh *createMeshFromHeightfield(double **heightfield, const int size);
void exportMeshAsObj(Mesh *mesh, const char *filename);
```

```
23 void freeMesh(Mesh *mesh);
24
25 #endif
```

C.1.7 mesh.cpp

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include "mesh.h"
5
6  // Create a triangle from three vertex indicies
7  Triangle makeTriangle(unsigned int a, unsigned int b, unsigned int c){
8      Triangle t;
9      t.v0 = a;
10     t.v1 = b;
11     t.v2 = c;
12     return t;
13 }
14
15 // Adds a vertex to a mesh
16 void addVertexToMesh(Mesh *mesh, double x, double y, double z){
17     mesh->vertices[mesh->vertexCount] = Vector(x, y, z);
18     (mesh->vertexCount)++;
19 }
20
21 // Adds a face to a mesh
22 void addFaceToMesh(Mesh *mesh, unsigned int a, unsigned int b, unsigned int c){
23     mesh->faces[mesh->faceCount] = makeTriangle(a, b, c);
24     (mesh->faceCount)++;
25 }
26
27 // Create a mesh from a heightfield
28 Mesh *createMeshFromHeightfield(double **heightfield, const int size){
29     Mesh *mesh = (Mesh*)malloc(sizeof(Mesh));
30     mesh->vertexCount = 0;
31     mesh->faceCount = 0;
32
33     int vertexCount = size * size + 2 * size + 2 * (size - 2);
34     mesh->vertices = (Vector*)malloc(vertexCount * sizeof(Vector));
35
36     int faceCount = 2 * (size - 1) * (size - 1) + 4 * 2 * (size - 1) + 2;
37     mesh->faces = (Triangle*)malloc(faceCount * sizeof(Triangle));
38
39     // Create verticies for every cell in the heightfield
40     for (int z = 0; z < size; ++z){
41         for (int x = 0; x < size; x++) {
42             double elevation = heightfield[z][x];
43             addVertexToMesh(mesh, x, elevation, z);
44         }
45     }
46
47     // Create pairs of triangles between each of the verticies to create a tesselation
48     for (int z = 0; z < size - 1; z++) {
49         for (int x = 0; x < size - 1; x++) {
50             int i = size * z + x;
51             addFaceToMesh(mesh, i, i + 1, i + 1 + size);
52             addFaceToMesh(mesh, i, i + 1 + size, i + size);
53         }
54     }
55
56     // Add cut-out sides to the mesh
57     int i = mesh->vertexCount;
58     int baseTopLeft = size * size;
59     int baseTopRight = baseTopLeft + size - 1;
60     int baseBottomLeft = baseTopLeft + size + 2 * (size - 2);
61     int baseBottomRight = baseBottomLeft + size - 1;
62
63     for (int x = 0; x < size; x++) {
64         addVertexToMesh(mesh, x, 0.0f, 0.0f);
65
66         if (x < size - 1){
67             addFaceToMesh(mesh, i, x, x + 1);
68             addFaceToMesh(mesh, i, x + 1, i + 1);
69         }
70         i++;
71     }
72     addFaceToMesh(mesh, baseTopLeft, 0, size);
73     addFaceToMesh(mesh, baseTopLeft, size, i);
74
75     for (int z = 1; z < size - 1; z++) {
76         addVertexToMesh(mesh, 0, 0.0f, z);
77         addFaceToMesh(mesh, i, size * z, size * (z + 1));
78
79         if (z < size - 2){
80             addFaceToMesh(mesh, i, size * (z + 1), i + 1);
81         }
82         else{
83             addFaceToMesh(mesh, i, size * (z + 1), baseBottomLeft);
84         }
85         i++;
86     }
87     addFaceToMesh(mesh, baseTopRight, size - 1, 2 * size - 1);
88     addFaceToMesh(mesh, baseTopRight, 2 * size - 1, i);
89
90     for (int z = 1; z < size - 1; z++) {
91         addVertexToMesh(mesh, size - 1, 0.0f, z);
```

```
        addFaceToMesh(mesh, i, size * z + size - 1, size * (z + 1) + size - 1);

        if (z < size - 2){
            addFaceToMesh(mesh, i, size * (z + 1) + size - 1, i + 1);
        }
        else{
            addFaceToMesh(mesh, i, size * (z + 1) + size - 1, baseBottomRight);
        }
        i++;
    }
    for (int x = 0; x < size; x++) {
        addVertexToMesh(mesh, x, 0.0f, size - 1);

        if (x < size - 1){
            addFaceToMesh(mesh, i, size * (size - 1) + x, size * (size - 1) + x + 1);
            addFaceToMesh(mesh, i, size * (size - 1) + x + 1, i + 1);
        }
        i++;
    }

    // Create base of mesh with two triangles
    addFaceToMesh(mesh, baseTopLeft, baseTopRight, baseBottomRight);
    addFaceToMesh(mesh, baseTopLeft, baseBottomRight, baseBottomLeft);

    return mesh;
}

// Export mesh as OBJ (Wavefront) file at a specfified file path
void exportMeshAsObj(Mesh *mesh, const char *filename){
    Vector *vertices = mesh->vertices;
    Triangle *faces = mesh->faces;

    FILE* objFile = fopen(filename, "w");
    if (!objFile) {
        fprintf(stderr, "Error: Unable to open file for writing: %s\n", filename);
        return;
    }

    // Write vertices
    for (int i = 0; i < mesh->vertexCount; i++) {
        fprintf(objFile, "v %f %f %f\n", vertices[i].x, vertices[i].y, vertices[i].z);
    }

    // Write faces
    for (int i = 0; i < mesh->faceCount; i++) {
        fprintf(objFile, "f %d %d %d\n", faces[i].v0 + 1, faces[i].v1 + 1, faces[i].v2 + 1);
    }

    fclose(objFile);
}

// Frees the memory for a mesh
void freeMesh(Mesh *mesh){
    free(mesh->vertices);
    free(mesh->faces);
    free(mesh);
}
```

C.2 Stream Power Model Code

C.2.1 poisson_disk_sampling.h

```
#ifndef POISSON_DISK_SAMPLING_H_
#define POISSON_DISK_SAMPLING_H_

#include <vector>
#include "../core/vector.h"

std::vector<Vector> poissonDiskSampling(float radius, Vector domainSize, int samplesBeforeRejection = 30);

#endif
```

C.2.2 poisson_disk_sampling.cpp

```
// Using algorithm introduced by Robert Bridson (Fast Poisson Disk Sampling in Arbitrary Dimensions)
// Implementation based on https://sighack.com/post/poisson-disk-sampling-bridsons-algorithm

#include <stdlib.h>

#include "poisson_disk_sampling.h"

float randomRange(float min, float max){
    return min + (max - min) * ((float)rand() / RAND_MAX);
}

bool isValidPoint(Vector point, float radius, Vector domainSize, std::vector<Vector> points, int** grid){
    float cellSize = radius / sqrt(2.0f);

    // Check in domain bounds
    if (point.x >=0 && point.x < domainSize.x && point.y >= 0 && point.y < domainSize.y){
        // Check neighbouring cells for points within radius
```

```
        int cellX = (int)(point.x / cellSize);
        int cellY = (int)(point.y / cellSize);
        int startX = std::max(0, cellX - 1);
        int endX = std::min(cellX + 1, (int)(domainSize.x / cellSize));
        int startY = std::max(0, cellY - 1);
        int endY = std::min(cellY + 1, (int)(domainSize.y / cellSize));

        for (int i = startX; i <= endX; i++){
            for (int j = startY; j <= endY; j++){
                int index = grid[j][i] -1;
                if (index != -1){
                    float distanceSquared = (point - points[index]).lengthSquared();
                    if (distanceSquared < pow(radius, 2)) {
                        return false;
                    }
                }
            }
        }
        return true;
    }

    return false;
}

std::vector<Vector> poissonDiskSampling(float radius, Vector domainSize, int samplesBeforeRejection){
    float cellSize = radius / sqrt(2.0f);

    int **grid = (int**)malloc((int)std::ceil(domainSize.y / cellSize) * sizeof(int*));
    for (int i = 0; i < domainSize.y / cellSize; i++){
        grid[i] = (int*)calloc((int)std::ceil(domainSize.x / cellSize), sizeof(int));
    }

    std::vector<Vector> points;
    std::vector<Vector> activePoints;

    activePoints.push_back(0.5f * domainSize);

    // While active list not empty
    while (activePoints.size() > 0){
        int index = (int)randomRange(0, activePoints.size());
        Vector centre = activePoints[index];

        bool pointValid = false;
    for (int i = 0; i < samplesBeforeRejection; i++){
            // Spherical sampling
            float angle = randomRange(0, 2.0f * M_PI);
            Vector direction = Vector(cos(angle), sin(angle));
            Vector samplePoint = randomRange(radius, 2.0f * radius) * direction + centre;

            // Check if point is valid
            if (isValidPoint(samplePoint, radius, domainSize, points, grid)){
                points.push_back(samplePoint);
                activePoints.push_back(samplePoint);
                grid[(int)(samplePoint.y / cellSize)][(int)(samplePoint.x / cellSize)] = points.size();
                pointValid = true;
                break;
            }
        }

        // If no valid point found after max samples, instead remove original point from active list
        if (!pointValid){
            activePoints.erase(activePoints.begin() + index);
        }
    }

    for (int i = 0; i < (int)std::ceil(domainSize.y / cellSize); i++){
        free(grid[i]);
    }
    free(grid);

    return points;
}
```

C.2.3 node.h

```
#ifndef NODE_H_
#define NODE_H_

#include <vector>

class Node{
public:
    std::vector<Node*> neighbours;

    Node(){

  }

    bool addEdge(Node *node);
    void removeEdge(Node *node);
    bool isNeighbour(Node *a);
};

#endif
```

C.2.4 node.cpp

```cpp
#include "node.h"

// Add edge to the graph, true if added, false if already present
bool Node::addEdge(Node *node){
    for (int i = 0; i < neighbours.size(); i++){
        if (neighbours[i] == node){
            return false;
        }
    }
    neighbours.push_back(node);
    return true;
}

// Remove edge form the graph
void Node::removeEdge(Node *node){
    for (int i = 0; i < neighbours.size(); i++){
        if (neighbours[i] == node){
            neighbours.erase(neighbours.begin() + i);
            break;
        }
    }
}

// Check is another node is a neighbour
bool Node::isNeighbour(Node *a){
    for (int i = 0; i < neighbours.size(); i++){
        if (neighbours[i] == a){
            return true;
        }
    }
    return false;
}
```

C.2.5 stream_node.h

```cpp
#ifndef STREAM_NODE_H_
#define STREAM_NODE_H_

#include "graph/node.h"
#include "../core/vector.h"
#include "lake_node.h"

class StreamNode : public Node{
public:
    double m;
    double n;
    double erosionConstant;
    double convergenceThreshold;
    double talusAngle;

    Vector position;
    double height;
    double uplift;

    double rainfall;
    double voronoiArea = 0;
    double drainageArea;

    bool boundaryNode = false;
    std::vector<int> edgeShareCount;

    StreamNode *downstreamNode = 0;
    std::vector<StreamNode*> upstreamNodes;

    LakeNode *lakeNode;

    StreamNode(){

  }

    StreamNode(double x, double y, double height, double uplift, double m, double n, double k, double
     ↪ convergenceThreshold, double talusAngle, double rainfall){
        this->position = Vector(x, y);
        this->height = height;
        this->uplift = uplift;
        this->m = m;
        this->n = n;
        this->erosionConstant = k;
        this->convergenceThreshold = convergenceThreshold;
        this->talusAngle = talusAngle;
        this->rainfall = rainfall;
    }

    bool addEdge(StreamNode *node);
    double calculateDrainageArea();
    void addToLake(LakeNode *lakeNode);
    bool update(double dt);
};

#endif
```

C.2.6 stream_node.cpp

```cpp
#include "stream_node.h"

// Add an edge to the stream graph
bool StreamNode::addEdge(StreamNode *node){
    for (int i = 0; i < neighbours.size(); i++){
        if (neighbours[i] == node){
            edgeShareCount[i]++;
            return false;
        }
    }
    neighbours.push_back(node);
    edgeShareCount.push_back(1);
    return true;
}

// Calculate drainage area of stream node recursively
double StreamNode::calculateDrainageArea(){
    double area = voronoiArea * rainfall;
    for (StreamNode *upstreamNode : upstreamNodes){
        area += upstreamNode->calculateDrainageArea();
    }

    drainageArea = area;
    return area;
}

// Assign a stream node to a lake
void StreamNode::addToLake(LakeNode *lakeNode){
    for (StreamNode *upstreamNode : upstreamNodes){
        upstreamNode->addToLake(lakeNode);
    }

    this->lakeNode = lakeNode;
}

// Perform the stream power equation update on a node
bool StreamNode::update(double dt){
    double newHeight = height;
    if (downstreamNode != 0){
        // Apply the
        double horizontalDistance = (position - downstreamNode->position).length();
        newHeight = (height + dt * (uplift + erosionConstant * pow(drainageArea, m) * downstreamNode->height /
    ↪ horizontalDistance)) /
                        (1 + erosionConstant * pow(drainageArea, m) * dt / horizontalDistance);

        // Apply correction based on thermal erosion
        double maxSlope = tan(talusAngle);
        double slope = (newHeight - downstreamNode->height) / horizontalDistance;
        if (slope > maxSlope){
            newHeight = downstreamNode->height + horizontalDistance * maxSlope;
        }
        else if (slope < (-1.0f * maxSlope)){
            newHeight = downstreamNode->height - horizontalDistance * maxSlope;
        }
    }

    // Update the children of the node in its stream tree
    bool childrenConverged = true;
    for (int i = 0; i < upstreamNodes.size(); i++){
        bool converged = upstreamNodes[i]->update(dt);
        childrenConverged = converged && childrenConverged;
    }

    // Check for convergence of this node, and its children
    bool converged;
    if (height == 0){
        converged = childrenConverged;
    }
    else{
        double hightDifference = fabs(newHeight - height) / height;
        converged = childrenConverged && (hightDifference < convergenceThreshold);
    }

    height = newHeight;

    return converged;
}
```

C.2.7 lake_node.h

```cpp
#ifndef LAKE_NODE_H_
#define LAKE_NODE_H_

#include <vector>

#include "stream_node.h"
#include "lake_edge.h"

class StreamNode;

class LakeEdge;

class LakeNode : public Node{
public:
    StreamNode *streamNode;
    bool isRiverMouth = false;
```

```
    std::vector<LakeEdge*> passes;

    LakeNode()
  {

  }

    LakeNode(StreamNode *streamNode,bool isRiverMouth){
        this->streamNode = streamNode;
        this->isRiverMouth = isRiverMouth;
  }
};

#endif
```

C.2.8 lake_edge.h

```
#ifndef LAKE_EDGE_H_
#define LAKE_EDGE_H_

#include <vector>

#include "stream_node.h"
#include "lake_node.h"

class StreamNode;

class LakeNode;

enum PassDirection {oneToTwo, twoToOne, none};

class LakeEdge{
public:
    double passHeight;
    StreamNode *passNode1;
    StreamNode *passNode2;

    LakeNode *lake1;
    LakeNode *lake2;

    PassDirection direction;

    LakeEdge()
  {

  }

    LakeEdge(StreamNode *n1, StreamNode *n2, double height);

    LakeNode *lowerLake();
    LakeNode *higherLake();
    StreamNode *lowerPass();
    StreamNode *higherPass();
};

class PassCompare{
    public:
        bool operator()(LakeEdge *a, LakeEdge *b){
            return (a->passHeight > b->passHeight);
        }
};

#endif
```

C.2.9 lake_edge.cpp

```
#include "lake_edge.h"

LakeEdge::LakeEdge(StreamNode *n1, StreamNode *n2, double height){
    passNode1 = n1;
    passNode2 = n2;
    passHeight = height;
    lake1 = passNode1->lakeNode;
    lake2 = passNode2->lakeNode;
}

// Gets lake edges lower elevation neighbour
LakeNode* LakeEdge::lowerLake(){
    if (direction == oneToTwo){
        return lake2;
    }
    else if (direction == twoToOne){
        return lake1;
    }
    else{
        return (LakeNode*)0;
    }
}

// Gets lake edges lower elevation neighbour
LakeNode* LakeEdge::higherLake(){
    if (direction == oneToTwo){
```

```
        return lake1;
    }
    else if (direction == twoToOne){
        return lake2;
    }
    else{
        return (LakeNode*)0;
    }
}

// Gets lake edges lower elevation stream node of the pass
StreamNode* LakeEdge::lowerPass(){
    if (direction == oneToTwo){
        return passNode2;
    }
    else if (direction == twoToOne){
        return passNode1;
    }
    else{
        return (StreamNode*)0;
    }
}

// Gets lake edges higher elevation stream node of the pass
StreamNode* LakeEdge::higherPass(){
    if (direction == oneToTwo){
        return passNode1;
    }
    else if (direction == twoToOne){
        return passNode2;
    }
    else{
        return (StreamNode*)0;
    }
}
```

C.2.10 stream_graph.h

```
#ifndef STREAM_GRAPH_H_
#define STREAM_GRAPH_H_

#include <vector>
#include <tuple>

#include "stream_node.h"
#include "../core/mesh.h"

Vector circumcentreOfTriangle(Vector a, Vector b, Vector c);
double areaOfTriangle(Vector a, Vector b, Vector c);

class StreamGraph{
public:
    int terrainSize;

    double **upliftField;
    int upliftFieldSize;
    float maximumUplift;

    bool isIsland = true;

    bool variableRainfall;
    double **rainfallField;
    int rainfallFieldSize;

    double timeStep;

    std::vector<StreamNode> nodes;
    std::vector<Triangle> triangles;
    std::vector<std::tuple<int, int>> edges;

    std::vector<LakeNode*> lakeGraph;

    StreamGraph(){

    }

    StreamGraph(int terrainSize, double timeStep, double **upliftField, int upliftFieldSize, float maxUplift, bool
     ↪ variableRainfall, double **rainfallField, int rainfallFieldSize){
        this->terrainSize = terrainSize;
        this->timeStep = timeStep;
        this->upliftField = upliftField;
        this->upliftFieldSize = upliftFieldSize;
        this->maximumUplift = maxUplift;
        this->variableRainfall = variableRainfall;
        this->rainfallField = rainfallField;
        this->rainfallFieldSize = rainfallFieldSize;
    }

    void initialise(int nodeCount, double m, double n, double k, double convergenceThreshold, double talusAngle);
    double getUplift(Vector p);
    void voronoiTessellation();
    void createStreamTrees();
    bool update();
    void calculatePasses();

    Mesh *createMesh();
```

```
    double **createHightfield(double precision, double sigma, double *maxHeight);

    double getRainfall(Vector p);

    double getMaxHeight();
};

#endif
```

C.2.11 stream_graph.cpp

```
#include <iostream>
#include <queue>
#include <math.h>

#include "stream_graph.h"

#include "../core/vector.h"
#include "../core/noise.h"
#include "../core/heightfield.h"

#include "poisson_disk_sampling.h"
#include "delaunay/CDT.h"

Vector circumcentreOfTriangle(Vector a, Vector b, Vector c){
    double t = a.lengthSquared() - b.lengthSquared();
    double u = a.lengthSquared() - c.lengthSquared();
    double J = ((a.x - b.x) * (a.y - c.y) - ((a.x - c.x) * (a.y - b.y))) * 2.0;

    double x = (-(a.y - b.y) * u + (a.y - c.y) * t) / J;
    double y = ((a.x - b.x) * u - (a.x - c.x) * t) / J;
    return Vector(x, y);
}

double StreamGraph::getUplift(Vector p){
    Vector upliftIndex = ((double)upliftFieldSize / (double)terrainSize) * p;
    double uplift = upliftField[(int)upliftIndex.x][(int)(upliftIndex.y)];
    return uplift;
}

double StreamGraph::getRainfall(Vector p){
    if (variableRainfall){
        Vector rainfallIndex = ((double)rainfallFieldSize / (double)terrainSize) * p;
        double rainfall = rainfallField[(int)rainfallIndex.x][(int)(rainfallIndex.y)];
        return rainfall;
    }

    return 1.0;
}

// Initialise the stream graph
void StreamGraph::initialise(int nodeCount, double m, double n, double k, double convergenceThreshold, double
    ↪ talusAngle){
    //Sample points for stream nodes using poisson disk samping
    double radius = (double)terrainSize / sqrt((double)nodeCount * 1.62);
    std::vector<Vector> points = poissonDiskSampling(radius, Vector(terrainSize, terrainSize));

    // Initialise each node
    for (Vector v : points){
        double uplift = getUplift(v);
        double height = 0.01 * fabs(perlinNoise(Vector(v.x / (double)terrainSize, v.y / (double)terrainSize), 5, 2.0,
    ↪ 0.5, 40.0));
        double rainfall = getRainfall(v);
        nodes.push_back(StreamNode(v.x, v.y, height, uplift, m, n, k, convergenceThreshold, (M_PI / 180) * talusAngle,
    ↪  rainfall));
    }

    // Perform Delaunay Triangulation
    CDT::Triangulation<double> cdt;
    cdt.insertVertices(
        points.begin(),
        points.end(),
        [](const Vector& p){ return p.x; },
        [](const Vector& p){ return p.y; }
    );

    cdt.eraseSuperTriangle();

    // Create stream graph from triangulation
    for (auto tri : cdt.triangles){
        Vector centre = circumcentreOfTriangle(nodes[tri.vertices[0]].position, nodes[tri.vertices[1]].position, nodes
    ↪ [tri.vertices[2]].position);
        if (centre.x <= 0.0 || centre.y <= 0.0 || centre.x > (double)terrainSize || centre.y > (double)terrainSize){
            //Exclude if centre outside of terrain boundary
            continue;
        }

        // Add nodes to graph, and add edges, if not already present

        nodes[tri.vertices[1]].addEdge(&nodes[tri.vertices[0]]);
        if (nodes[tri.vertices[0]].addEdge(&nodes[tri.vertices[1]])){
            edges.push_back(std::make_tuple(tri.vertices[0], tri.vertices[1]));
        }

        nodes[tri.vertices[2]].addEdge(&nodes[tri.vertices[0]]);
        if (nodes[tri.vertices[0]].addEdge(&nodes[tri.vertices[2]])){
```

```
            edges.push_back(std::make_tuple(tri.vertices[2], tri.vertices[0]));
        }

        nodes[tri.vertices[2]].addEdge(&nodes[tri.vertices[1]]);
        if (nodes[tri.vertices[1]].addEdge(&nodes[tri.vertices[2]])){
            edges.push_back(std::make_tuple(tri.vertices[1], tri.vertices[2]));
        }

        // Create triangles
        Triangle face = {(unsigned int)tri.vertices[0], (unsigned int)tri.vertices[1], (unsigned int)tri.vertices[2]};
        triangles.push_back(face);
    }


    // Define which nodes are outflows from the terrain (the boundary nodes)
    for (int i = 0; i < nodes.size(); i++){
        if (nodes[i].boundaryNode){
            continue;
        }
        for (int j = 0; j < nodes[i].neighbours.size(); j++){
            if (nodes[i].edgeShareCount[j] < 2  && (isIsland || (nodes[i].uplift < 0.33 * maximumUplift))){
                nodes[i].boundaryNode = true;
                nodes[i].height = 0.0f;
                ((StreamNode*)nodes[i].neighbours[j])->boundaryNode = true;
                ((StreamNode*)nodes[i].neighbours[j])->height = 0.0;
                continue;
            }
        }
    }

    // Calculate voronoi areas for each node
    voronoiTessellation();
}

// Calculates the area of the triangle of three points.
double areaOfTriangle(Vector a, Vector b, Vector c){
    return fabs(a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y)) / 2;
}

void StreamGraph::voronoiTessellation(){
    for (Triangle tri : triangles){
        // Calculate circumcentre of triangle
        Vector circumcentre = circumcentreOfTriangle(nodes[tri.v0].position, nodes[tri.v1].position, nodes[tri.v2].
    ↪ position);
        unsigned int vertexIndicies[3] = {tri.v0, tri.v1, tri.v2};
        // For each node the triangle
        for (int i = 0; i < 3; i++){
            StreamNode *node = &nodes[vertexIndicies[i]];
            // For all neighbours of node
            for (int j = 0; j < 3; j++){
                if (j != i){
                    StreamNode neighbour = nodes[vertexIndicies[j]];
                    // Calculate midpoint of edge between node and neighbour
                    Vector midpoint = 0.5 * (node->position + neighbour.position);
                    // Calculate area of trianlge formed by node position, neighbour position, and circumcentre of
    ↪ triangle
                    double area = areaOfTriangle(node->position, midpoint, circumcentre);
                    node->voronoiArea += area;
                }
            }
        }
    }
}

// Calculate stream trees based on heights of stream nodes
void StreamGraph::createStreamTrees(){
    // Reset upstream and downstream stream tree connections for all nodes in stream graph
    for (int i = 0; i < nodes.size(); i++){
        StreamNode *node = &(nodes[i]);
        node->downstreamNode = 0;
        node->upstreamNodes.clear();
        node->lakeNode = 0;
    }

    // Create new stream trees
    for (int i = 0; i < nodes.size(); i++){
        StreamNode *node = &(nodes[i]);
        StreamNode *lowest = 0;
        double lowestHeight = node->height;

        // Find lowest neighbour to become downstream node
        for (Node *n : node->neighbours){
            StreamNode *neighbour = (StreamNode*)n;
            if (neighbour->height < lowestHeight){
                lowestHeight = neighbour->height;
                lowest = neighbour;
            }
        }

        // Create connection with lowest node
        if (lowest != 0){
            node->downstreamNode = lowest;
            lowest->upstreamNodes.push_back(node);
        }
    }
}

// Update stream graph
bool StreamGraph::update(){
```

```
    // Create stream trees
    createStreamTrees();

    // For each root, assign lakes
    for (int i = 0; i < nodes.size(); i++){
        if (nodes[i].downstreamNode == 0){
            // Root
            // Assign to lake
            LakeNode *lakeNode = new LakeNode(&nodes[i], nodes[i].boundaryNode);
            nodes[i].addToLake(lakeNode);
            lakeGraph.push_back(lakeNode);
        }
    }

    // Calculate passes
    calculatePasses();

    // For each root, calculate drainage areas.
    for (int i = 0; i < nodes.size(); i++){
        if (nodes[i].boundaryNode){
            // Root
            // Calculate drainage area
            nodes[i].calculateDrainageArea();
        }
    }

    // For each tree from root to leaf
    // Update node
    bool allConverged = true;
    for (int i = 0; i < nodes.size(); i++){
        if (nodes[i].boundaryNode){
            bool converged = (&nodes[i])->update(timeStep);
            allConverged = converged && allConverged;
        }
    }

    //Free lakes
    for (int i = 0; i < lakeGraph.size(); i++){
        delete lakeGraph[i];
    }

    lakeGraph.clear();

    return allConverged;
}

void StreamGraph::calculatePasses(){
    std::vector<LakeEdge> passes;
    std::vector<LakeEdge> validPasses;

    // Find lake connections which could form passes
    for (int i = 0; i < edges.size(); i++){
        StreamNode *node1 = &(nodes[std::get<0>(edges[i])]);
        StreamNode *node2 = &(nodes[std::get<1>(edges[i])]);

        if (node1->lakeNode == node2->lakeNode){
            // Both nodes lead to same lake
            continue;
        }

        // Check if potential pass is the lowest between the nodes, and update if it is.
        bool isPass = true;
        for (int j = 0; j < passes.size(); j++){
            LakeEdge p = passes[j];
            if ((p.lake1 == node1->lakeNode && p.lake2 == node2->lakeNode) || (p.lake1 == node2->lakeNode && p.lake2
 ↪ == node1->lakeNode)){
                isPass = false;
                double height = std::max(node1->height, node2->height);
                if (height < p.passHeight){
                    isPass = true;
                    passes.erase(passes.begin() + j);
                }
                break;
            }
        }

        if (isPass){
            // Add connection in lake graph
            node1->lakeNode->addEdge(node2->lakeNode);
            node2->lakeNode->addEdge(node1->lakeNode);
            double passHeight = std::max(node1->height, node2->height);
            passes.push_back(LakeEdge(node1, node2, passHeight));
        }
    }

    // Compute lake trees
    std::priority_queue<LakeEdge*, std::vector<LakeEdge*>, PassCompare> candidates;

    // Remove connections aways from river mouth (no flow upstream)
    for (int i = 0; i < lakeGraph.size(); i++){
        if (lakeGraph[i]->isRiverMouth){
            for (int j = 0; j < lakeGraph[i]->neighbours.size(); j++){
                if (!((LakeNode*)(lakeGraph[i]->neighbours[j]))->isRiverMouth){
                    for (int p = 0; p < passes.size(); p++){
                        // Set direciton of lake flow for lake trees
                        if (passes[p].lake1 == lakeGraph[i] && passes[p].lake2 == lakeGraph[i]->neighbours[j]){
                            passes[p].direction = twoToOne;
                            candidates.push(&passes[p]);

```

```
                        }
                        else if (passes[p].lake2 == lakeGraph[i] && passes[p].lake1 == lakeGraph[i]->neighbours[j]){
                            passes[p].direction = oneToTwo;
                            candidates.push(&passes[p]);
                        }
                    }
                }
            }
            lakeGraph[i]->neighbours.clear();
        }
    }

    // Build valid lake tree
    buildLakeTree:
    while (!candidates.empty()){
        // // Find candidate pass with minimum height
        LakeEdge *minHeightPass = candidates.top();
        candidates.pop();

        LakeNode *lowerLake = minHeightPass->lowerLake();
        LakeNode *upperLake = minHeightPass->higherLake();

        for (int i = 0; i < validPasses.size(); i++){
            if (validPasses[i].higherLake() == upperLake){
                // Cannot have lake flowing in two differnt directions
                goto buildLakeTree;
            }
        }

        validPasses.push_back(*minHeightPass);

        // Add neighbours of upper lake as candidates for next pass (if not river mouth)
        for (int j = 0; j < upperLake->neighbours.size(); j++){
            if (!((LakeNode*)(upperLake->neighbours[j]))->isRiverMouth){
                for (int p = 0; p < passes.size(); p++){
                    // Set direciton of lake flow for lake tree
                    if (passes[p].lake1 == upperLake && passes[p].lake2 == upperLake->neighbours[j]){
                        passes[p].direction = twoToOne;
                        candidates.push(&passes[p]);

                    }
                    else if (passes[p].lake2 == upperLake && passes[p].lake1 == upperLake->neighbours[j]){
                        passes[p].direction = oneToTwo;
                        candidates.push(&passes[p]);
                    }
                }
            }
        }

        upperLake->neighbours.clear();
    }

    // Update stream graph with new connections formed by the lake trees
    for (int p = 0; p < validPasses.size(); p++){
        StreamNode *upperLakeRoot = validPasses[p].higherLake()->streamNode;
        StreamNode *receiverNode = validPasses[p].lowerPass();

        receiverNode->upstreamNodes.push_back(upperLakeRoot);
        upperLakeRoot->downstreamNode = receiverNode;
    }
}

// Create mesh of stream graph
Mesh* StreamGraph::createMesh(){
    Mesh *mesh = (Mesh*)malloc(sizeof(Mesh));

    mesh->vertexCount = nodes.size();
    mesh->vertices = (Vector*)malloc(mesh->vertexCount * sizeof(Vector));
    for (int i = 0; i < mesh->vertexCount; i++){
        mesh->vertices[i] = Vector(nodes[i].position.y, nodes[i].height, nodes[i].position.x);
    }

    mesh->faceCount = triangles.size();
    mesh->faces = (Triangle*)malloc(mesh->faceCount * sizeof(Triangle));
    for (int i = 0; i < mesh->faceCount; i++){
        mesh->faces[i] = triangles[i];
    }

    return mesh;
}

// Creates a heightfield from the stream graph, using gaussian filters
double** StreamGraph::createHightfield(double precision, double sigma, double *maxHeight){
    int arraySize = (int)(terrainSize / precision);
    double **heightfield = createHeightfield(arraySize);
    double **kernelSum = createHeightfield(arraySize);

    int range = (int)(4.0 * sigma);

    // Sum the gaussian kernels * height on the heightfield
    for (int n = 0; n < nodes.size(); n++){
        int nodeX = (int)(nodes[n].position.x / precision);
        int nodeZ = (int)(nodes[n].position.y / precision);

        for (int i = std::max(nodeX - range, 0); i < std::min(nodeX + range, arraySize); i++) {
            for (int j = std::max(nodeZ - range, 0); j < std::min(nodeZ + range, arraySize); j++) {
                double distanceSquared = pow(i - nodeX, 2) + pow(j - nodeZ, 2);
                double gaussian = exp(-distanceSquared / (2 * pow(sigma, 2)));
                heightfield[i][j] += (nodes[n].height / precision) * gaussian;
```

```
                kernelSum[i][j] += gaussian;
            }
        }
    }

    // Normalise by the kernel sum
    double max = 0.0f;
    for (int i = 0; i < arraySize; i++) {
        for (int j = 0; j < arraySize; j++) {
            heightfield[i][j] /= kernelSum[i][j];
            if (heightfield[i][j] > max){
                max = heightfield[i][j];
            }
        }
    }

    freeHeightfield(kernelSum, arraySize);
    *maxHeight = max;

    return heightfield;
}

double StreamGraph::getMaxHeight(){
    double maxHeight = 0.0;
    for (int i = 0; i < nodes.size(); i++){
        if (nodes[i].height > maxHeight){
            maxHeight = nodes[i].height;
        }
    }

    return maxHeight;
}
```

C.3 Main Program

C.3.1 terrain.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include <chrono>
#include <cstring>
#include <iostream>
#include <fstream>

#include "core/heightfield.h"
#include "core/mesh.h"
#include "core/noise.h"

#include "streamPower/stream_graph.h"

// Saves a frame of the heightfield
void generateImageFrame(StreamGraph *sg, int iteration, char* heightfieldFilename, int terrainSize, double resolution,
    ↪  double standardDev){
    double maxHeight;
    double **heightfield = sg->createHightfield(resolution, standardDev, &maxHeight);
    std::string filenameString = heightfieldFilename;
    filenameString.append(std::to_string(iteration));
    filenameString.append(".ppm");
    char filename[filenameString.length() + 1];
    strcpy(filename, filenameString.c_str());
    outputHeightfieldAsImage(heightfield, terrainSize / resolution, maxHeight, filename);
    freeHeightfield(heightfield, terrainSize / resolution);
}


bool hasArg(int argc, char *argv[], char* argName){
    for (int i = 0; i < argc; i++){
        if (strcmp(argv[i], argName) == 0){
            return true;
        }
    }
    return false;
}

char *getArg(int argc, char *argv[], char* argName){
    for (int i = 0; i < argc; i++){
        if ((strcmp(argv[i], argName) == 0) && (i + 1 < argc)){
            return argv[i+1];
        }
    }
    return 0;
}

int main(int argc, char *argv[]){
    // Parse arguments
    if (argc > 46) {
        std::cout << "Too many arguments!";
        return 1;
    }

    // Perlin noise
```

```
57      char *perlinNoiseDataFilename = getArg(argc, argv, (char*)"-perlinNoiseDataFilename");
58
59      // Parameters
60
61      // Size of the terrain
62      int terrainSize = atoi(getArg(argc, argv, (char*)"-terrainSize"));
63      // Approximate number of points to sample in the terrain
64      int nodeCount = atoi(getArg(argc, argv, (char*)"-nodeCount"));
65
66      // Stream power equation
67      // Drainage area constant
68      double m = atof(getArg(argc, argv, (char*)"-m"));
69      // Slope constant
70      double n = atof(getArg(argc, argv, (char*)"-n"));
71      // Erosion constant
72      double k = atof(getArg(argc, argv, (char*)"-k"));
73
74      // Simulation iteration
75      // Simulation time step
76      double timeStep = atof(getArg(argc, argv, (char*)"-timeStep"));
77      //Maximum number of time steps to run before stopping
78      int maxTimeSteps = atoi(getArg(argc, argv, (char*)"-maxTimeSteps"));
79      // Value to determine if simulation can stop before max time steps
80      double convergenceThreshold = atof(getArg(argc, argv, (char*)"-convergenceThreshold"));
81
82      // Uplift
83      // Maximum value of the uplift to control height of mountains
84      double maximumUplift = atof(getArg(argc, argv, (char*)"-maximumUplift"));
85      // Filename of ppm image file which contains the uplift field
86      char *upliftFieldFilename = getArg(argc, argv, (char*)"-upliftFieldFilename");
87
88      // Thermal erosion
89      // Maximum thermal erosion talus angle
90      double talusAngle = atof(getArg(argc, argv, (char*)"-talusAngle"));
91
92      // Rainfall
93      bool variableRainfall = hasArg(argc, argv, (char*)"-variableRainfall");
94      // Maximum value of the rainfall
95      double maximumRainfall = atof(getArg(argc, argv, (char*)"-maximumRainfall"));
96      // Filename of ppm image file which contains the rainfall field
97      char *rainfallFieldFilename = getArg(argc, argv, (char*)"-rainfallFieldFilename");
98
99      // Ouput mesh
100     // Should tesselated mesh of terrain be generated
101     bool generateMesh = hasArg(argc, argv, (char*)"-generateMesh");
102     // Filename to write mesh to
103     char *meshFilename = getArg(argc, argv, (char*)"-meshFilename");
104
105     // Output heightfield
106     // Should heightfield be generated
107     bool generateHeightfield = hasArg(argc, argv, (char*)"-generateHeightfield");
108     // Filename to write heightfield ppm image file to
109     char* heightfieldFilename = getArg(argc, argv, (char*)"-heightfieldFilename");
110     // Size of heightfield
111     int heightfieldSize = atoi(getArg(argc, argv, (char*)"-heightfieldSize"));
112     // Standard deviation of gaussian filter for heightfield generation
113     double heightfieldStandardDeviation = atof(getArg(argc, argv, (char*)"-heightfieldStandardDeviation"));
114     // Should heightfield mesh be generated
115     bool generateHeightfieldMesh = hasArg(argc, argv, (char*)"-generateHeightfieldMesh");
116     // Filename to write heightfield mesh file to
117     char* heightfieldMeshFilename = getArg(argc, argv, (char*)"-heightfieldMeshFilename");
118     //Should heightfield image sequence be generated
119     bool generateImageSequence = hasArg(argc, argv, (char*)"-generateImageSequence");
120     // Interval of frames of image sequence
121     int imageSequenceInterval = atoi(getArg(argc, argv, (char*)"-imageSequenceInterval"));
122
123     // Main program
124     srand(time(NULL));
125     loadNoisePermutation(perlinNoiseDataFilename);
126
127     double heightfieldResolution = (float)terrainSize / (float)heightfieldSize;
128
129     std::ofstream resultFile;
130     resultFile.open("results.txt");
131     resultFile << "Arguments:\n";
132     for (int i = 1; i < argc; i++){
133         if (argv[i][0] == '-'){
134             resultFile << "\n";
135         }
136         else{
137             resultFile << " ";
138         }
139         resultFile << argv[i];
140     }
141
142     // Import uplift
143     int upliftFieldSize;
144     double **upliftField = importImageAsHeightfield(upliftFieldFilename, &upliftFieldSize, maximumUplift);
145
146     // Import rainfall
147     int rainfallFieldSize;
148     double **rainfallField = importImageAsHeightfield(rainfallFieldFilename, &rainfallFieldSize, maximumRainfall);
149
150     // Initialise
151     StreamGraph sg = StreamGraph(terrainSize, timeStep, upliftField, upliftFieldSize, maximumUplift, variableRainfall,
        ↪  rainfallField, rainfallFieldSize);
152     sg.initialise(nodeCount, m, n, k, convergenceThreshold, talusAngle);
153
154     std::cout << "Initialised stream graph\n";
```

```
155     std::cout.flush();
156     resultFile << "\n\nNode count: " << sg.nodes.size();
157 
158     std::ofstream timesFile;
159     timesFile.open("times.csv");
160     timesFile << "Iteration, LakeCount, Duration, MaxHeight\n";
161 
162     // Main simulation loop
163     std::cout << "Running Simulation";
164     auto start = std::chrono::high_resolution_clock::now();
165     for (int i = 0; i < maxTimeSteps; i++){
166         // Update terrain
167         bool converged = sg.update();
168 
169         std::cout << ".";
170         std::cout.flush();
171 
172         if (converged){
173             resultFile << "\nModel converged in " << i << " time steps";
174             break;
175         }
176 
177         if (generateImageSequence && i % imageSequenceInterval == 0){
178             generateImageFrame(&sg, i, heightfieldFilename, terrainSize, heightfieldResolution,
     ↪ heightfieldStandardDeviation);
179         }
180     }
181     auto stop = std::chrono::high_resolution_clock::now();
182     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
183 
184     // Write info to file:
185     resultFile << "\nTime elapsed: " << (double)duration.count() / 1000000 << "s\n";
186     resultFile << "Maximum height: " << sg.getMaxHeight() << "m\n";
187     resultFile.close();
188 
189     freeHeightfield(upliftField, upliftFieldSize);
190 
191     if (rainfallField != 0){
192         freeHeightfield(rainfallField, rainfallFieldSize);
193     }
194 
195     if (generateMesh){
196         //Create mesh from graph and export mesh as OBJ file
197         Mesh *mesh = sg.createMesh();
198         exportMeshAsObj(mesh, meshFilename);
199         freeMesh(mesh);
200     }
201 
202     if (generateHeightfield){
203         // Generate heightfield and output as image
204         double maxHeight;
205         double **heightfield = sg.createHightfield(heightfieldResolution, heightfieldStandardDeviation, &maxHeight);
206         outputHeightfieldAsImage(heightfield, terrainSize / heightfieldResolution, maxHeight, heightfieldFilename);
207 
208         if (generateHeightfieldMesh){
209             // Generate mesh from heightfield
210             Mesh *mesh = createMeshFromHeightfield(heightfield, terrainSize / heightfieldResolution);
211             exportMeshAsObj(mesh, heightfieldMeshFilename);
212             freeMesh(mesh);
213         }
214 
215         freeHeightfield(heightfield, terrainSize / heightfieldResolution);
216     }
217 
218     return 0;
219 }
```