FH Technikum Wien

MASTER THESIS

Thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Engineering

at the University of Applied Sciences Technikum Wien
Master Game Engineering & Simulation

Real-Time Erosion Simulation & Visualization with OpenCL

by
Michael Muck, BSc
3452 Atzenbrugg, Leopold Figl Straße 8

Supervisors:
Dipl.-Ing. Andreas Monitzer, Bakk. techn. MSc
Dipl.-Ing. Dr. Markus Schordan

Vienna, March 5, 2012

FACHHOCHSCHULE
TECHNIKUM WIEN

Declaration

I confirm that this paper is entirely my own work. All sources and quotations have been fully acknowledged in the appropriate places with adequate footnotes and citations. Quotations have been properly acknowledged and marked with appropriate punctuation. The works consulted are listed in the bibliography. This thesis has not been submitted to another examination panel in the same or a similar form, and has not been published.

Place, Date

Signature

Kurzfassung

Um die Erosion eines Terrains simulieren zu können gibt es die verschiedensten Methoden. Allen gemeinsam ist die Komplexität, welche oftmals eine Echtzeitsimulation umöglich macht. Trotz der immer schneller werdenden Hardware ist es daher von großer Wichtigkeit einen geeigneten Algorithmus zu finden, der auch unter den verschiedensten Hardwarekonfigurationen effizient funktioniert und glaubwürdige Ergebnisse produziert.

Um dieses Ziel zu erreichen, werden in dieser Arbeit zunächst einige bekannte Algorithmen zur Erosionssimulation analysiert und danach ein weiterer vorhandener Algorithmus, welcher auf Geschwindigkeitsfeldern von Wasser, die durch ein optimiertes Shallow-Water Modell generiert werden, untersucht. In weiterer Folge wird die Implementierung und Erweiterung dieses Algorithmus mit Hilfe des OpenCL Frameworks vorgestellt und auch näher auf die Umsetzung und die dabei erreichten Ergebnisse eingegangen.

OpenCL (Open Computing Language) ist eine neue Programmierplattform (erste Spezifikation von 8.12.2008 - erste Implementierungen ab Aug. 2009) für CPUs, GPUs aber auch DSPs, welche die Hardware abstrahiert um Berechnungen („Kernels“) auf verschiedene Geräte im System beliebig verteilen zu können. Im Gegensatz zu CUDA, welches nur auf nVidia Grafikkarten lauffähig ist, ist OpenCL eine offene Plattform, welche im Moment von allen großen Herstellern aktiv unterstützt wird.

Schlagwörter: Hydraulische Erosion, Verwitterung, Simulation, OpenCL, Implementierung, Echtzeit, Shallow-Water, Pipe Model

Abstract

There are many methods for simulating erosion of a terrain. All of them have in common a high complexity, which often makes it impossible to run the simulation in real-time. Despite the increasing speed of the hardware it is very important to use a suitable algorithm, which also runs efficiently under the most common hardware configurations and produces credible results.

To achieve this goal, the known algorithms for erosion simulation are reviewed at first. Then another existent algorithm, which relies on the velocity field of the running water which is generated by an optimized Shallow-Water model, is examined. Furthermore an improved implementation of this algorithm with the OpenCL Framework is presented and the realisation and its results are discussed in detail.

OpenCL (Open Computing Language) is a new programming platform (first specification released on 8th december 2008 – first implementations where shown in august 2009) for CPUs, GPUs as well as DSPs, which aims to abstract the actual hardware and distribute the calculations (so called "Kernels") to various devices in the system. In contrast to CUDA being for nVidia devices only, OpenCL is also an open platform which is supported by all major hardware vendors.

Keywords: Hydraulic Erosion, Weathering, Simulation, OpenCL, Implementation, Real-Time, Shallow-Water, Pipe Model

Acknowledgements

First of all i would like to thank Dipl-Ing. Andreas Monitzer, Bakk. techn. Msc. my first supervisor for his many advices as a professional as well as Dipl.-Ing. Dr. Markus Schordan my second supervisor.

Second i want to thank Bernadette Thalhammer for supporting me whenever i needed her help and for tolerating the nights awake spent on this thesis.

Special thanks go to Patrick Thalhammer for his instant proofreading, Stefan Sperlhofer for advices regarding the visualization and Emanuel Plochberger for the rich correspondence regarding OpenCL.

At last i would also like to thank my parents, who supported me during my studies whenever and wherever possible.

Contents

1 Introduction

Although erosion simulation or terrain modelling in general is a well explored field in computer science, it still presents a reasonable challenge to engage. While there have been proposed many good algorithms in soil science, all of them have in common a high complexity, which often makes it difficult or even impossible to run a simulation in real-time. Since there is an always emerging demand for realistic simulation in games too, the pressure is even higher to find a good balance between realism and a soft real-time simulation. Despite the increasing speed of hardware it is therefore very important to use a suitable algorithm which also runs efficiently under the most common hardware configurations and produces credible results.

To achieve this goal it is essential to know where to start, what is possible and where are the boundaries of a given algorithm. This master thesis is designated to provide such information on a detailed level by analysing, extending and implementing a gpu-accelerated erosion variant which is based on an optimized Shallow-Water model and was proposed in a series of papers recently. To ensure the compatibility with as many devices as possible and to exploit the parallel nature of the problem, OpenCL was chosen as the implementation platform.

Initially the chapter *Related Work* (2) gives an overview of already researched approaches in this sector. Furthermore the closely related recent works will be discussed in more detail.

In the next chapter *Erosion Model* (3) the model which is going to be implemented will be described. The erosion model takes several approaches from "Related Work" and extends them where necessary.

In the following chapter *A Brief Introduction to OpenCL* (4) the OpenCL Platform, its Memory Model and Execution Environment as well as the OpenCL C language is introduced, culminating in a short basic example on how to start an OpenCL application from scratch.

The chapter *Implementation of a Test Framework*(5) gives an overview of the host application at first. Subsequently the focus is on the OpenCL implementation of the given erosion model as introduced in chapter (3). Also pitfalls which are uncovered during the development process will be discussed here as well.

In *Results* (6) the achieved effects of the erosion model variants are then compared to each other as well as to natural erosion and weathering phenomena. Also performance measurements and comments on the specific settings are presented in this chapter.

Chapter *Conclusion* (7) closes the master thesis with a final summary of the learnings that have been gathered and gives an outlook what a future work on erosion simulation could improve even further.

2 Related Work

2.1 Overview

Many erosion algorithms have been examined in the past. Starting from fractal rules (Harmon et al. [1] and Valette et al. [2]) to produce eroded terrains like in *Terrain Simulation Using a Model of Stream Erosion* (Kelly et al. [3]) where drainage networks generated by geological data together with fractal terrain are used and in *A Fractal Model of Mountains with Rivers* (Prusinkiewicz et al. [4]) where rivers where integrated through midpoint displacement into mountain models, to more physically based models which modify the relief of premodeled/generated terrains. While the earlier papers basically focus on the algorithms, the majority of the more recent ones also target the domain of (soft) real time simulation.

Musgrave et al. [5] suggest to first create a terrain (fractal based) and then simulate a simple diffusion-based hydraulic erosion process. A thermal weathering simulation which deposits material based on the local surface gradient is also discussed.

Roudier et al. [6] extended the erosion process by applying local geological parameters like fluvial erosion, gravity creep and chemical dissolution and different materials.

In *Computer Generation of Eroded Valley and Mountain Terrains* [7] Nagashima also presented a model in which valleys and canyons are generated by a synthesis of a generated fractal terrain and a predefined river network. The river banks then get eroded by a combination of hydraulic and thermal erosion.

Beneš et al. present in their paper *Visual Simulation of Hydraulic Erosion* [8] another diffusion-based erosion simulation variant in which the visual appearance and not the physical accuracy plays a major role. Beneš et al. divide the simulation process in several independent steps (Water Update & Evaporation, Erosion & Deposition, Water & Sediment Transport) to simplify the repeated execution of arbitrary steps. The deposition is based on the evaporation of water, leaving behind the sediment.

All methods described previously use either a simple diffusion based water model or a derivative. Since this solution is only practicable for slow moving fluids and not very accurate, Chiba et al. [9] propose a different solution in which a velocity field is calculated from the motion of water particles. The name "velocity field" is rather used as an umbrella term for the different simulation data arrays (e.g. water quantity, velocity vector and collision energy arrays). In the next step the erosion and transportation process calculates new output values from the updated data. The algorithm works very well for the simulation of ridges and valleys and also takes the collision between water and the ground surface into account. This particle-based approach was improved further by Sutherland et al. in *Particle-based enhancement of terrain data* [10].

Beneš et al. also improved their algorithm in *Hydraulic Erosion* [11] by using Navier-Stokes equations for velocity and pressure simulations on a three-dimensional regular grid. Although yielding impressive visual results, their solution was far away from a real-time simulation.

In *Interactive Physically Based Fluid and Erosion Simulation* [34] Neidhold et al. propose a new approach with a strong focus on real-time simulation. To accomplish this, Neidhold uses a layered terrain instead of a full three-dimensional representation. The water simulation is based on Stam's Semi-Lagrangian *Stable Fluids* [12], but reduced to a two-dimensional version, which is necessary to save computation time. After each water update step, a diffusion step is used to avoid oscillation and smooth the velocity field. The erosion algorithm is mass conserving and interactively customizeable. On the downside the algorithm is still of limited use for real time applications - the paper reports 4 frames per seconds (FPS) on a 256x256 grid. Optimization through parallelization was not an option due to the algorithm's dependent data processing.

In recent years more and more people are using graphics processing hardware to cover general computing problems [13]. There are many examples especially in the fluid simulation domain which make use of these concepts very successfully.

In *Fast Fluid Dynamics Simulation on the GPU* [14] Harris proposes a fluid simulation based on the *Stable Fluids* method [12] which runs a 2D Navier Stokes Equation (NSE) solver entirely on the GPU. At the same time Wu et al. also presented a similar approach in their paper [15].

In another paper from the same year Liu et al. [16] went a step ahead and simulated a fully three dimensional NSE on the GPU. If the grid size is kept at a very low level (e.g. 64x17x16) interactive frame rates ranging from 20 to 40 fps can be achieved with their solution. Unfortunately using larger grid sizes is limited by the rapidly increasing memory consumption which has a significant negative impact on the frame rates or makes it even impossible to run.

Other methods of fluid simulation like the Lattice Boltzmann Method (LBM) were reviewed in *Implementing Lattice Boltzmann Computation on Graphics Hardware* [17]. Although their GPU solution shows a significant speedup over a similar software solution it has basically the same drawbacks as every other 3D solver: the memory requirement. More recent implementations show significant performance improvements due to enhanced graphics hardware, but the limit is still the high memory requirement [18] (e.g. D3Q19@128x128x128 $\Rightarrow$ 2.097.152 Cells with 96 Bytes per Cell, needed twice for flip-flopping $\Rightarrow$ 384MB - D3Q19@256x256x256 $\Rightarrow$ 3072MB)

As discussed before, a fully three-dimensional approach is not affordable on a larger scale, therefore other researchers focused on two-dimensional shallow water problems leading to the Shallow Water Equations (SWE). These equations are derived from depth-integrating the Navier-Stokes equations and are only applicable if the horizontal length scale is much greather than the vertical length scale. Under this circumstances the vertical velocities are very small and the horizontal velocities can be assumed to be constant throughout the depth at a given point [19].

Since Shallow Water Equations are two-dimensional they have some obvious limitations: the water cannot splash and waves cannot break. However there are efficient methods to simulate SWE's and as long as the forces are sufficiently gentle, the assumption produces plausible results. Kass and Miller [20] showed in 1990 an implicit numerical method which was

implemented in software and ran at 32fps on a 32x32 grid at that time. Later on Beneš [21] employed the same method for a real-time erosion simulation and achieved 5-10 fps on a 300x300 grid.

Another SWE variant was introduced in 1995 by O'Brien et al. [22]. They proposed a virtual pipe model which is composed of a volume of water which is divided into vertical columns in a rectilinear grid. In this heightfield, each cell is connected to its neighbours by virtual pipes. The flow in the pipes is calculated from the physical laws for hydrostatic pressure. O'Brien et al. also considered external forces on the surface which are applied as external pressure. Furthermore spray particles are generated if the upward velocity exceeds a certain threshold.

2.2 Fast Hydraulic Erosion Simulation and Visualization on GPU

In their paper *Fast Hydraulic Erosion Simulation and Visualization on GPU* Mei et al. [23] propose a new and very fast erosion simulation method suitable to run on graphics hardware.

Basically their approach uses an adapted virtual pipe model on a two dimensional heightfield for the water simulation (see O'Brien et al. [22]). Unlike in *Dynamic Simulation of Splashing Fluids* there is no explicit scaling back process for cells where the updated water height is negative, because of the dependent data processing which is not affordable on GPUs. Instead it is assumed that the outflow of each cell is limited by its water amount. To enforce this condition, the water outflow is scaled down by a calculated factor if the amount would be exceeded.

The erosion process is based on the velocity field which is calculated from the water pressure differences. With the velocity field and other terrain parameters a sediment transport capacity is calculated from an empirically determined erosion formula and compared to the current suspended sediment. Then the suspended sediment is transported along the velocity field by a simple semi-Lagrangian advection method which was introduced in *Stable Fluids* [12]. Following Beneš et al. [8] an evaporation step is also included in this model.

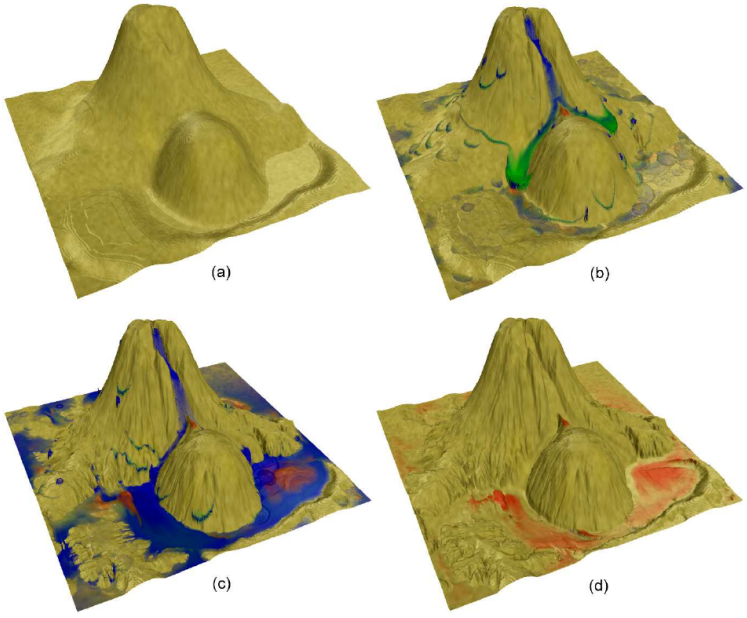


Figure 2.1: Erosion simulation on GPU with rainfall and a river source (source: [23], page 8)

The implementation uses three 2D textures for the computation data. In multiple fragment shader passes the new values are obtained and written into another texture stack for the next pass. The simulation takes fully place on the graphics card - no data transfer between the host and the graphics memory is involved. In total seven passes are used for the simulation part. After that the data from the textures is directly used to displace the height of a predefined regular grid mesh. A simple fragment shader computes the final color where the transparency varies with the water height of the cells. Multiple test scenes are presented (see figure 2.1 for an example) and also the performance is measured. For a grid size of 256x256 a frame rate of approximately 400fps is achieved - dropping to 185fps for a 512x512 grid and 59fps for a 1024x1024 grid. The performance was measured using an Pentium IV 2.4GHz equipped with an nVidia GeForce 8800 GTX graphics card.

2.3 Interactive Terrain Modeling Using Hydraulic Erosion

Later on Stava et al. [24] also took a similar approach in their paper *Interactive Terrain Modeling Using Hydraulic Erosion*. In contrary to Mei et al. they also focus on the interactivity. This was accomplished by combining two hydraulic erosion algorithms and simulating multiple material layers. Also the slippage of material due to gravity was taken into account. To address the limited GPU memory the terrain was divided into multiple tiles which can be processed independently.

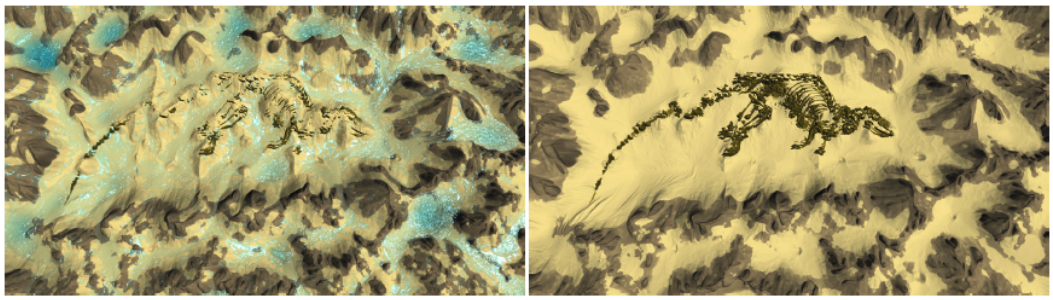


Figure 2.2: Real-time simulation of erosion exposing a fossil skeleton (source: [24], page 2)

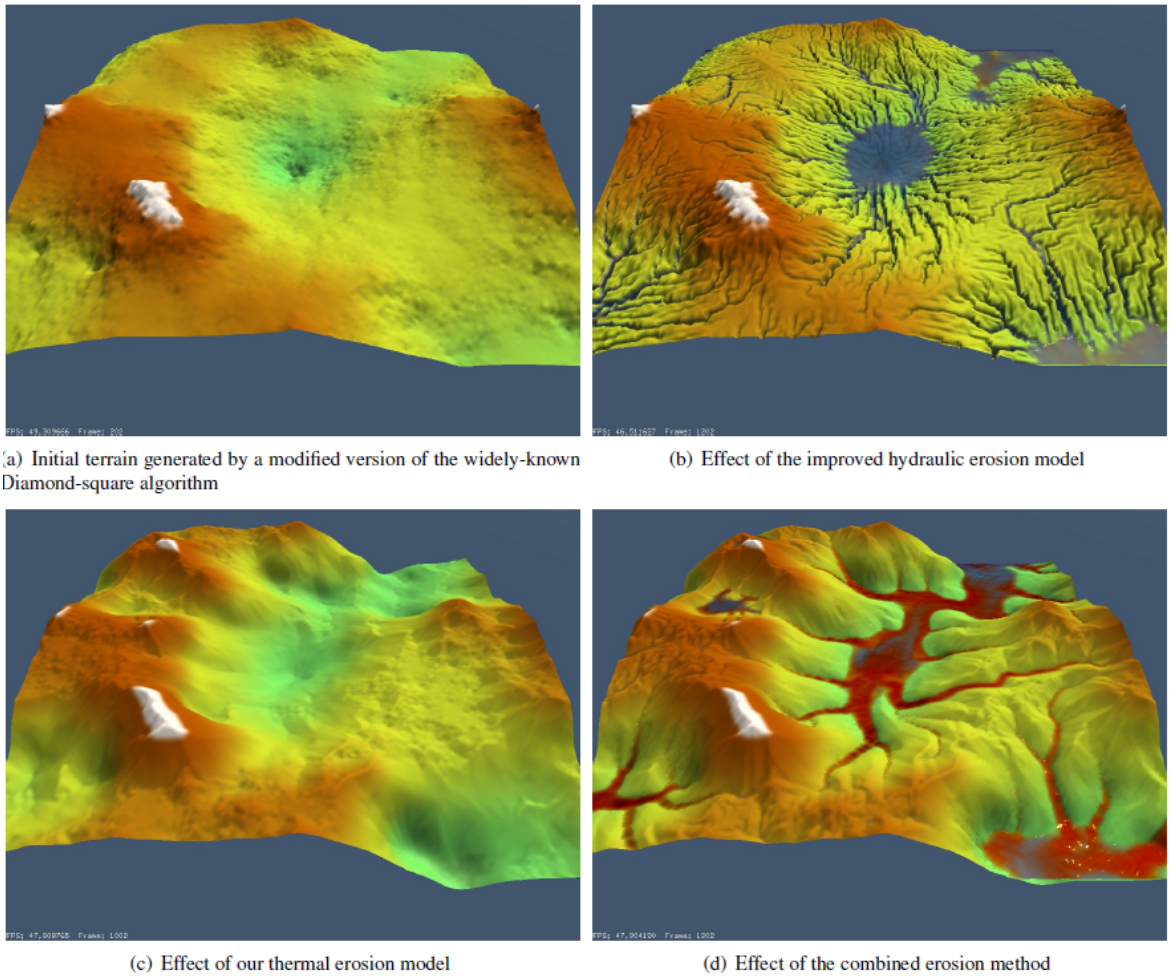
To describe a scene, a layered height field with different material constants like dissolution traits or resistance to the water movement is used. For the visual representation, a real-time renderer is used, but an export of the heightfield into a third-party package for further modeling and rendering is also possible. The erosion and deposition process only takes place between the topmost material layers. Stava et al. also introduced an improved sediment transportation step which uses the MacCormack advection scheme, a second order accuracy method (see [25]). An example of this method can be seen in figure 2.2.

2.4 Fast Hydraulic and Thermal Erosion on the GPU

Recently another paper which also uses the water pipe model was published [26]. In his work Jákó proposes a new interpretation of "Fast Hydraulic Erosion Simulation and Visualization on GPU" [23] in which he extends the original model by thermal erosion simulation and a true 3D collision between water and terrain surface. He also attempts to fix some minor drawbacks of the original

model e.g. the dissolved sediment is only subtracted from the terrain without adding the volume to the water height.

The thermal "erosion" model is mainly a GPU adapted form of *Thermal Weathering* as proposed by Musgrave et al. [5] but instead of moving the calculated sediment volume directly to the lowest neighbours, the quantities are distributed over another set of virtual pipes. In an additional simulation step the terrain height is then updated for each cell by subtracting the cell's outgoing material flow from the incoming material flow of the neighbour cells. Figure 2.3 shows an example from the paper.



(a) Initial terrain generated by a modified version of the widely-known Diamond-square algorithm

(b) Effect of the improved hydraulic erosion model

(c) Effect of our thermal erosion model

(d) Effect of the combined erosion method

Figure 2.3: Fast Hydraulic and Thermal Erosion (source: [26], page 6)

3 Erosion Model

At the bottom the virtual pipe model as used in *Fast Hydraulic Erosion Simulation and Visualization on GPU* [23] defines the data layout for the water and erosion simulation. As described in section 2.2 this algorithm works on a regular grid where each cell is composed of various properties. The basic layout is extended by additional data fields for e.g. the random rain distribution or the thermal weathering simulation which where not part of the original approach.

- Terrain Height *b*
- Water Height *d*
- Suspended Sediment Amount *s*
- Water Volume Outflow *Q*, consisting of (Q^L, Q^R, Q^T, Q^B), respectively $(Q^L, Q^R, Q^T, Q^B, Q^{LT}, Q^{RT}, Q^{RB}, Q^{RT})$
- Water Surface Velocity Vector $\vec{v} = \begin{pmatrix} u \\ v \end{pmatrix}$
- Soil Volume Outflow *S*, consisting of $(S^L, S^R, S^T, S^B, S^{LT}, S^{RT}, S^{RB}, S^{RT})$
- Local Hardness Coefficient *h*
- Regolith Wetness *sq*
- Cell Normal *n*
- Pseudo Random Value *rand*

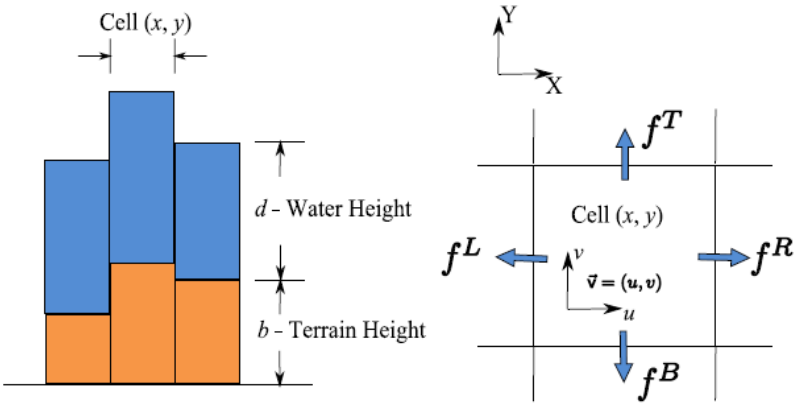


Figure 3.1: Basic data structure and neighbouring information (source: [23], page 3)

The simulation itself is divided into several distinct steps which will be described in detail in the following sections:

1. Rand Buffer Update
2. Terrain Normal Generation
3. Water Increase due to rainfall or river sources
4. Water Simulation:
   Outflow Simulation and Water Height Update
   Velocity Field Calculation
5. Force based Erosion:
   Erosion-Deposition Process Simulation
   Suspended Sediment Advection, caused by the Velocity Field
6. Thermal Erosion:
   Material Outflow Calculation
   Terrain Height Update
7. Water Evaporation

The input data of many steps is dependant on data from previous steps. To emphasize this, number subscripts are used for intermediate values (d_1, d_2, etc. ...) whereas the value at the beginning of an iteration is denoted as e.g. d_t (water height at time t). The timestep for a full simulation iteration is termed $\triangle t$. The fully updated value is then called e.g. $d_{t+\triangle t}$.
Letters $L, R, T, B, LT, RT, RB, LB$ are used to indicate directions from the current cell to the according Left, Right, Top, Bottom, Left-Top, Right-Top, Right-Bottom and Left-Bottom cell.

The above enumeration does not imply that the actual implementation has the same processing order. It is merely a logical grouping of the simulation processes.

3.1 Pseudo Random Number Generator

For the random number generation a simple *Linear Congruential Generator* (LCG) is used. Its implementation is comparatively lightweight, fast and adequate for the problem. Each cell holds a random value which is initialized at program startup by another offline RNG. At every simulation step the current *rand* value is read and updated by the standard LCG formula:

$$R_{t+1}(x, y) = (R_t(x, y) * a + b) \mod m \tag{3.1}$$

3.2 Terrain Surface Normal Generation

For the erosion capacity calculation and for the lighting terrain normals are necessary. Since the terrain is dynamically changing, it makes sense to offload the generation of the terrain normals to

the GPU as well. Again the simplest affordable approach is used to minimize gpu ressource usage.

$$\begin{aligned}
\vec{x1} &= P(x+1,y) - P(x-1,y) \\
\vec{y1} &= P(x,y+1) - P(x,y-1) \\
\\
\vec{x2} &= P(x-1,y-1) - P(x+1,y+1) \\
\vec{y2} &= P(x+1,y-1) - P(x-1,y+1) \\
\\
\vec{n1} &= \vec{x1} \times \vec{y1} \\
\vec{n2} &= \vec{x2} \times \vec{y2} \\
\\
\vec{n} &= -normalize(n0+n1)
\end{aligned} \tag{3.2}$$

3.3 Water Increment

Water has two sources: springs and rainfall. Springs are static water sources with distinct radii, positions and water amounts. Rainfall can be either random or "constant" whereupon the random rainfall is based on a certain threshold and $R_t(x,y)$ of the current cell. For "constant" rainfall the rain intensity is simply added to every cell. The intensity is variable and can be regulated from within the program.

$$\begin{aligned}
rand_rain_t(x,y) &= \begin{cases} 0 & , R_t(x,y) \leq threshold \\ rain_intensity & , R_t(x,y) > threshold \end{cases} \\
rain_t(x,y) &= rand_rain_t(x,y) + constant_rain(x,y) \\
d_1(x,y) &= d_t(x,y) + \triangle t \cdot (spring(x,y) + rain_t(x,y))
\end{aligned} \tag{3.3}$$

3.4 Water Simulation

The pipe model from O'Brien et al. [22] works by exchanging water with the neighbouring cells through virtual pipes. Every cell maintains an input/output volume flow from/to its neighbour cells. The new height is then calculated from the sum of all incoming and outgoing volume flows of that cell. If the resulting water height is negative, the pipes which are removing fluid of the cell must be scaled back until all cells have a positive volume again. For real-time GPU solutions this approach is not suitable, since it involves dependant data processing over multiple steps. Nevertheless the algorithm from O'Brien et al. [22] provides the foundation of the improved GPU-suited model of "Fast Hydraulic Erosion Simulation and Visualization on GPU".

3.4.1 Outflow Calculation

The equations for water flow rate calculation rely on the physical law for hydrostatic pressure which is defined as

$$p = d \cdot \rho \cdot g + p_0 \tag{3.4}$$

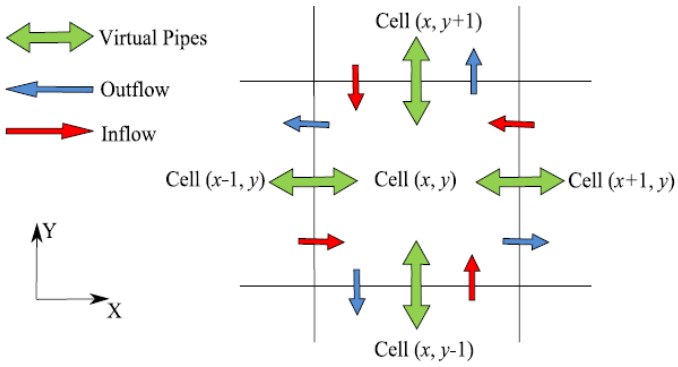


Figure 3.2: Pipe model notations in Mei et al. (source: [23], page 4)

where d stands for the water height, ρ is the fluids density, g is the gravitation acceleration and $p0$ is the atmospheric pressure in the system.

From the view of a regular grid we can rewrite this equation to

$$p(x, y) = d(x, y) \cdot \rho \cdot g + p_0 \tag{3.5}$$

where each cell has its distinctive water height $d(x, y)$ and therefore different pressures. The water height of a cell can be determined from its volume $V(x, y)$ and the given cell size l_{cell}

$$d(x, y) = \frac{V(x, y)}{l_{cell} \cdot l_{cell}} \tag{3.6}$$

To calculate the water volume flow rate Q of a cell to its neighbour we need to consider the basic formula of the flow rate

$$Q = \Delta t \cdot A_{surface} \cdot a \tag{3.7}$$

Following the definition of the flow rate, we need to compute the acceleration a of the current water mass in the pipe traveling from a cell to its respective neighbour. In our case the cross sectional area $A_{surface} = A_{pipe}$ is directly related to the cell geometry and can be described as

$$A_{pipe} = d(x, y) \cdot l_{cell} \tag{3.8}$$

To get the acceleration of the displaced mass we have to consider Newton's Second Law

$$F = m \cdot a \tag{3.9}$$

and the formula for pressure

$$p = \frac{F_\perp}{A} \Rightarrow F_\perp = p \cdot A \tag{3.10}$$

where $F_\perp$ is the acting effective force on the area A_{pipe}.

Looking at the two formulas from a pipes view we can write

$$F_{pipe} = m_{water} \cdot a_{water} = p_{pipe} \cdot A_{pipe} \tag{3.11}$$

where p_{pipe} is the pressure difference Δp between the current cell and its respective neighbour. Combining this knowledge gives us the acceleration in a pipe as

$$a = \frac{(p - p_{neighbour}) \cdot A_{pipe}}{m_{water}} \tag{3.12}$$

Again A_{pipe} stands for the virtual pipe's cross-sectional area on which the water's driving force stands perpendicular. The term $p - p_{neighbour}$ expresses the effective statical pressure in the pipe and m_{water} is the current mass of the water in the pipe

Derived from the basic formula of density we can express the mass by its volume V and the water density ρ.

$$\rho = \frac{m}{V} \Rightarrow m = \rho \cdot V \tag{3.13}$$

The water volume in the pipe can be calculated as

$$V = A_{pipe} \cdot l_{pipe} \tag{3.14}$$

where A_{pipe} itself is defined by the current water height $d1(x, y)$ and cell side length l_{cell} which yields the formula for m_{water}

$$m_{water} = \rho_{water} \cdot A_{pipe} \cdot l_{pipe} \tag{3.15}$$
$$A_{pipe} = d_1(x, y) \cdot l_{cell} \tag{3.16}$$
$$\Rightarrow m_{water} = \rho_{water} \cdot d_1(x, y) \cdot l_{cell} \cdot l_{pipe} \tag{3.17}$$

Together with the original formula of static fluid pressure (see equation 3.4) equation 3.12 can be rewritten to

$$a^i = \frac{\rho_{water} \cdot g \cdot (d_1(x, y) - d_1^i) \cdot d_1(x, y) \cdot l_{cell}}{\rho_{water} \cdot d_1(x, y) \cdot l_{cell} \cdot l_{pipe}} \tag{3.18}$$
$$a^i = \frac{g \cdot (d_1(x, y) - d_1^i))}{l_{pipe}} \tag{3.19}$$
$$i=L,R,T,B$$

Assuming the acceleration being constant over the time period $\triangle t$, the flow rate $\{Q^i(x, y), i = L, R, T, B, ...\}$ for the pipe can be expressed as

$$Q^i_{t+\triangle t}(x, y) = Q^i_t(x, y) + \triangle t \cdot A_{pipe} \cdot a^i(x, y) \tag{3.20}$$
$$\tag{3.21}$$

leading to a volume change of

$$\triangle V(x, y) = \triangle t \sum_{i=L,R,T,B} \left(\frac{Q^i_{t+\triangle t}(x, y) + Q^i_t}{2} \right) \tag{3.22}$$

Notice that all of the equations are only approximations which rely on the assumption, that the fluid is shallow and not moving rapidly. As mentioned before all of the cells need to be tested

for negative updated volumes. If a cell has a negative volume all pipes which are removing fluid from this cell must be scaled back.

Mei et al. [23] solved this problem by updating the new water height step by step. At first the outflow volume of every cell is calculated just like before. For example the updated outflow to the left neighbour cell Q^L (Q^R, Q^T, Q^B, etc. ... are defined similar)

$$Q_1^L(x,y) = max(0, Q_t^L(x,y) + \triangle t \cdot A_{pipe} \cdot \frac{g \cdot \triangle h^L(x,y)}{l_{pipe}}) \tag{3.23}$$

where $Q_t^L(x,y)$ is the current outflow value to the left neighbour, A_{pipe} is the cross-sectional area of the pipe, g is the acceleration due to gravity and $\triangle h^L(x,y)$ is the total height difference (terrain b + water d) between the left neighbour and the current cell which is defined by the formula:

$$\triangle h^L(x,y) = b_t(x,y) + d_1(x,y) - b_t(x-1,y) - d_1(x-1,y) \tag{3.24}$$

Equation 3.21 and 3.23 are essentially the same. The difference lies in the outflow which is limited to positive values. If the outflow exceeds the water volume it is scaled down

$$K = min(1, \frac{d_1(x,y) \cdot l_{cell} \cdot l_{cell}}{(Q_1^L + Q_1^R + Q_1^T + Q_1^B) \cdot \triangle t}) \tag{3.25}$$

leading to a flow rate of

$$Q_{t+\triangle t}^L(x,y) = K \cdot Q_1^L(x,y) \tag{3.26}$$

At the borders a "no slip" boundary condition is specified so that no water can flow out of the grid. This is accomplished by setting the respective cell outflows to zero: e.g. a cell at the left border has an outflow rate of $Q^L(0,y) = 0$ whereas a cell at the top border has an outflow of $Q^T(x,0) = 0$. Other configurations are possible as well, simulating sinks or sources.

Another difference which is a result of limiting the outflow to positive values, is the calculation of the net volume change. Instead of taking the average volume flow of the last timestep $\triangle t$ the change is calculated by subtracting the sum of outflow of the current cell from the sum of the neighbour outflows to the current cell (=inflow)

$$\triangle V(x,y) = \triangle t \cdot (\sum Q_{in} - \sum Q_{out}) \tag{3.27}$$

The next intermediate water height $d2$ is then updated as

$$d_2(x,y) = d_1(x,y) + \frac{\triangle V(x,y)}{l_{cell} \cdot l_{cell}} \tag{3.28}$$

3.4.2 Velocity Field Calculation

O'Brien et al. then calculate a vertical velocity as well as a horizontal velocity for each cell. The vertical velocity is used for a spray model where particles are created when the velocity exceeds a certain threshold. Mei et al [23] basically use the same properties for their horizontal velocity field,

whereas the spray model is omitted. To calculate the horizontal velocity $v = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$ the average flow rate of a cell in x-/y-direction during a timestep $\triangle t$ is computed as

$$\triangle Q_x(x,y) = \frac{Q^R(x-1,y) - Q^L(x,y) + Q^R(x,y) - Q^L(x+1,y)}{2} \tag{3.29}$$

O'Brien et al. directly use $\triangle Q_x$ respectively $\triangle Q_y$ as a measurement for the surface velocity even though its a volume rate. Mei et al. take this into account by dividing the flow rate by the cell's cross-sectional area $l_{cell} \cdot d_{avg}$ during the timestep $\triangle t$. Water height d_{avg} is defined as the average water height between the first two water height update steps.

$$d_{avg}(x,y) = \frac{d1(x,y) + d2(x,y)}{2} \tag{3.30}$$

$$v_x(t+\triangle t) = \frac{\triangle Q_x(x,y)}{l_{cell} \cdot d_{avg}(x,y)} \tag{3.31}$$

The other component of the flow velocity $v_y(t+\triangle t)$ is calculated in a similar way.

3.4.3 Extension to Moore-Neighbourhood

Mei et al. use only four neighbours for the water simulation, but it can be easily extended to handle all eight cell neighbours. Care must be taken because of the different pipe lengths in the diagonals e.g. for equation 3.23

$$Q_1^{LT}(x,y) = max(0, Q_t^{LT}(x,y) + \triangle t \cdot A_{pipe} \cdot \frac{g \cdot \triangle h^L(x,y)}{\sqrt{2} \cdot l_{pipe}}) \tag{3.32}$$

The scaling factor also changes to

$$K = min(1, \frac{d_1(x,y) \cdot l_{cell} \cdot l_{cell}}{(Q_1^L + Q_1^R + Q_1^T + Q_1^B + Q_1^{LT} + Q_1^{RT} + Q_1^{RB} + Q_1^{LB}) \cdot \triangle t}) \tag{3.33}$$

3.5 Hydraulic Erosion and Deposition

The basic erosion process is a force-based hydraulic erosion algorithm. It is dependant on the velocities which are generated by the running water and its effects on the terrain. Many prediction models have been proposed for this type of erosion in soil science. Mei et al. [23] chose a simplified empirical approach from [27] which is mostly determined by the sediment transport capacity C. C is calculated as

$$C(x,y) = K_c \cdot sin(max(\alpha_{min}, \alpha(x,y))) \cdot |\vec{v}(x,y)| \tag{3.34}$$

where K_c is a scaling constant, $\alpha(x,y)$ is the local tilt angle and $\vec{v}(x,y)$ is the local surface velocity. The local tilt angle can be calculated from the dot product of the cells surface normal and the up vector. C is strongly related to the terrain geometry and the surface velocity which is a problem for very flat terrains where the tilt angle approaches zero. As a result C is very small and little soil, or no soil will be picked up from the ground at all. To overcome this problem α can be limited to a minimum threshold α_{min}.

In a paper recently published by Jákó [26] some improvements where added to the Erosion/Deposition step. At this point of the algorithm, the calculation of the sediment capacity is extended by the ramp function l_{max} to associate the sediment capacity (and therefore the erosion depth) with the water height.

$$l_{max}(val) = \begin{cases} 0 & , val \leq 0 \\ 1 - \frac{K_{dmax} - val}{K_{dmax}} & , 0 < val < K_{dmax} \\ 1 & , val \geq K_{dmax} \end{cases} \tag{3.35}$$

which leads to

$$C(x,y) = K_c \cdot sin(max(\alpha_{min}, \alpha(x,y))) \cdot |\vec{v}(x,y)| \cdot l_{max}\left(d1(x,y)\right) \tag{3.36}$$

The purpose of the function is to scale down the sediment capacity if the water height is in the range of $0 - K_{dmax}$. If the water height is above K_{dmax} the ramp function has no effect.

After the calculation of the current transport capacity the updated value is compared to the suspended sediment amount s_t. If the capacity is greater than the suspended sediment amount ($C > s_t$) the water can collect some more soil from the ground:

$$dissolvedSoil = K_s \cdot h(x,y) \cdot (C - s_t) \tag{3.37a}$$
$$b_1 = b_t - dissolvedSoil \tag{3.37b}$$
$$d_3 = d_2 + dissolvedSoil \tag{3.37c}$$
$$s_1 = s_t + dissolvedSoil \tag{3.37d}$$

otherwise ($C \leq s_t$) some suspended sediment is deposited on the ground:

$$depositedSoil = clamp(K_d \cdot (s_t - C), 0.0, d_2) \tag{3.38a}$$
$$b_1 = b_t + depositedSoil \tag{3.38b}$$
$$d_3 = d_2 - depositedSoil \tag{3.38c}$$
$$s_1 = s_t - depositedSoil \tag{3.38d}$$

K_s and K_d are global erosion scale parameters which control the dissolving respectively the deposition speed.

Jákó also suggests a fix to the water simulation by adding the dissolved soil which is taken from the ground to the water height (equations 3.37c and 3.38c). This ensures that the cells overall height in the current timestep does not vary and improves longterm stability. Not adding the dissolved soil causes unwanted feedback to the water flow simulation in the original simulation [26]. If soil gets deposited it is subtracted from the water height again. To ensure a positive water height after this step, $depositedSoil$ needs to be limited by the current water heigth $d2$

Another improvement is the local hardness coefficient h which varies between 0.0 and 1.0 and scales down the dissolved sediment amount. The cell's hardness coefficient is lowered when some soil is deposited at the cell position, causing the erosion to take less soil the next time the cell contributes ground to the erosion process.

$$h_{t+\triangle t}(x,y) = max\left(Rmin, h_t(x,y) - \triangle t \cdot K_h \cdot K_s \cdot (s_t - C)\right) \tag{3.39}$$

3.5.1 Sediment Advection

After the suspended sediment update step, it is transported to a new location by the velocity field. This can be described by the advection equation:

$$\frac{\partial s}{\partial t} + (\vec{v} \cdot \nabla s) = 0 \tag{3.40}$$

A simple forward Euler step would lead to problems regarding the numerical stability as well as the numerical error [23]. Therefore Mei et al. use the semi-Lagrangian approach which was introduced by Stam [12] to solve the advection equation. Unlike Stam who uses a particle tracer to find the velocity of a point a timestep $\triangle t$ ago, Mei et al. use the velocity to go back in time and forward the sediment found at this position.

Therefore the new suspended sediment of a cell can be obtained by taking an Euler-step backward in time which has been proven to be unconditionally stable [12] [23]:

$$\begin{pmatrix} x_n \\ y_n \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} - \triangle t \cdot \begin{pmatrix} v_x(x,y) \\ v_y(x,y) \end{pmatrix} \tag{3.41}$$

$$s_{t+\triangle t}(x,y) = s_1(x_n, y_n) \tag{3.42}$$

Since in most cases the resulting position $\begin{pmatrix} x_n \\ y_n \end{pmatrix}$ does not directly correspond to an integral grid position, the actual value of $s_{t+\triangle t}$ is computed using a bilinear interpolation of the four nearest neighbours.

Modified MacCormack scheme

Stava et al. [24] suggest an improvement for the transportation step, using a second order accuracy back and forth error compensation and correction (BFECC) advection method to minimize the numerical diffusion of sediment transport. They solve the advection equation using a modified MacCormack advection scheme.

The original BFECC scheme can be described as

$$\hat{\phi}^{n+1} = A(\phi^n) \tag{3.43a}$$

$$\hat{\phi}^{n} = A^R(\hat{\phi}^{n+1}) \tag{3.43b}$$

$$e = \frac{\hat{\phi}^n - \phi^n}{2} \tag{3.43c}$$

$$\bar{\phi}^n = \phi^n - e \tag{3.43d}$$

$$\Rightarrow \phi^{n+1} = A(\bar{\phi}^n) \tag{3.43e}$$

At first the forward advection operator A is applied to the initial value to compute an intermediate result $\hat{\phi}^{n+1}$. Then the backward advection operator A^R is used on the intermediate result to calculate $\hat{\phi}^n$. The difference of these two values is approximately twice the advection error e. With the knowlege of the advection error, the initial value can be adjusted and advected again with A thus yielding the end result.

Following Selle et. al [25] the scheme can also be written in its equivalent but computationally cheaper variant (modified MacCormack scheme)

$$\hat{\phi}^{n+1} = A(\phi^n) \tag{3.44a}$$

$$\hat{\phi}^{n} = A^R(\hat{\phi}^{n+1}) \tag{3.44b}$$

$$\phi^{n+1} = \hat{\phi}^{n+1} + \frac{\phi^n - \hat{\phi}^n}{2} \tag{3.44c}$$

where each advection operator is a first order accurate semi-Lagrangian unconditionally stable building block.

The big advantage of the MacCormack scheme is that only twice the effort of a first order accurate scheme is necessary in contrary to other second order methods which makes it affordable to use. The method can be further improved by automatically limiting the values to a simple first order scheme if the method pulls data from where it should not.

3.6 Thermal Weathering

The thermal weathering simulation works similar to the water simulation. First each cell calculates a soil outflow which is then distributed through virtual pipes in a following step. In general the term "thermal weathering" is used in this context to sum up the effects of any process that causes material to displace and pile up at another position [26]. Overall it clearly has a smoothing effect on the terrain.

Figure 3.3: Talus Angle in Nature, photograph by Michelle Lamberson [1]

3.6.1 Material Displacement Calculation

The calculation of the material displacement is based on the work of Musgrave et al. [5] and was recently adopted to GPU hardware by Jákó [26]. The key observation is that each solid granular

[1] http://www.flickr.com/people/vitrain/

material like sand or earth has a certain maximum angle for its slope (see figure 3.3). If there is material above this maximum angle it starts moving to lower levels. Once the material reaches the critical angle it stops moving.

This so called talus angle α can be measured by dumping the material slowly to a flat surface between two transparent plates (see figure 3.4). From the emerging slope the angle can be easily determined [26].

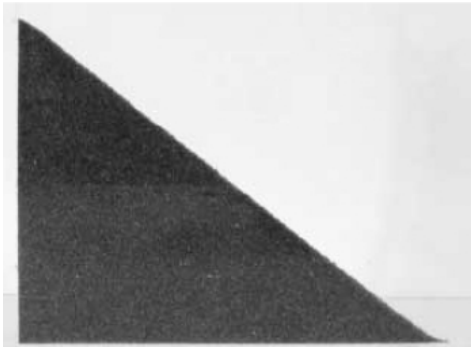


Figure 3.4: Talus angle measurement (source [26])

Following the implementation of Jákó, each cell has to compute the surface tilt angle to its eight neighbours. This is accomplished by looking at the height differences between the current cell and its neighbours. The maximum moved volume per timesteps is defined as

$$\triangle V = A_{cell} \cdot \frac{H}{2} \tag{3.45}$$

where A_{cell} is the cell's surface area $l_{cell} \cdot l_{cell}$ and height H is defined as

$$H = max\left\{b - b^i, i = L, R, T, B, LT, RT, RB, LB\right\} \tag{3.46}$$

otherwise the algorithm would oscillate [26].

Jákó extended the algorithm to also use the local hardness coefficient $h(x, y)$ and introduced a scaling factor K_t to control the thermal weathering speed. The full formula for the volume $\triangle V$ which is going to be moved during a timestep $\triangle t$ is then given as

$$\triangle V = A_{cell} \cdot \triangle t \cdot K_t \cdot h(x, y) \cdot \frac{H}{2} \tag{3.47}$$

The slope angle between two neighbouring cells is computed as

$$\alpha^i = \tan\left(\frac{b - b^i}{l_{cell}}\right), \text{i = L, R, T, B} \tag{3.48}$$

$$\alpha^i = \tan\left(\frac{b - b^i}{\sqrt{(2)} \cdot l_{cell}}\right), \text{i = LT, RT, RB, LB} \tag{3.49}$$

Where $b - b^i$ denotes the height difference between the neighbouring cells and l_{cell} is used as the distance between cells. For diagonal neighbours this distance should be corrected accordingly.

Material	Angel of Repose
fine grained sand	35°
fine grained sand (wet)	45°
fine grained sand (water filled)	15-30°
coarse grained sand	40°
coarse debris	45°

Table 3.1: Typical values for Talus Angles (source [28])

The algorithm then uses α to determine the set A of neighbours which receive a part of the volume $\triangle V$

$$A = \left\{ b^i, b - b^i < 0 \wedge tan(\alpha) > (h(x,y) \cdot K_a + K_i), \text{i = L, R, T, B, LT, RT, RB, LB} \right\} \tag{3.50}$$

where K_a is a global constant to regulate the influence of the local hardness coefficient $h(x,y)$. K_i is the material slope angle (see table 3.1 for typical values). If the local hardness is close to one the effective material slope angle gets bigger, thus allowing the material to withstand bigger angles.

The calculated amount $\triangle V$ is then divided proportionally amongst the cells of set A according to the cell height, respectively the height difference of the particular cell

$$\triangle V^k = \triangle V \cdot \frac{b^k}{\sum_{\forall b^k \in A} b^k} \tag{3.51}$$

As last step the value for each V^k is written to the soil outflow pipes of the current cell. The pipes to cells which are not in set A are reset to zero.

3.6.2 Terrain Height Update

The second step of the thermal weathering simulation simply subtracts the outflowing material from the incoming material and updates the height of the current cell accordingly.

$$b_{t+\triangle t}(x,y) = b_1(x,y) + \left(\sum \triangle V_{in} - \sum \triangle V_{out} \right) \tag{3.52}$$

3.7 Water Evaporation

Mei et al. use a simple formula to describe the evaporation

$$d_{t+\triangle t}(x,y) = d_3(x,y) \cdot (1 - K_e \cdot \triangle t) \tag{3.53}$$

where K_e is a global evaporation constant. Figure 3.5 shows different evaporation constants and their influence on the water level over time.

Evaporation usually depends on various parameters like the flow rate of air, atmospheric pressure, surface area, fluid density, etc... For example the dalton equation (as formulated by Penman [29]):

$$E_0 = (e_s - e_d) \cdot f(u) \tag{3.54}$$

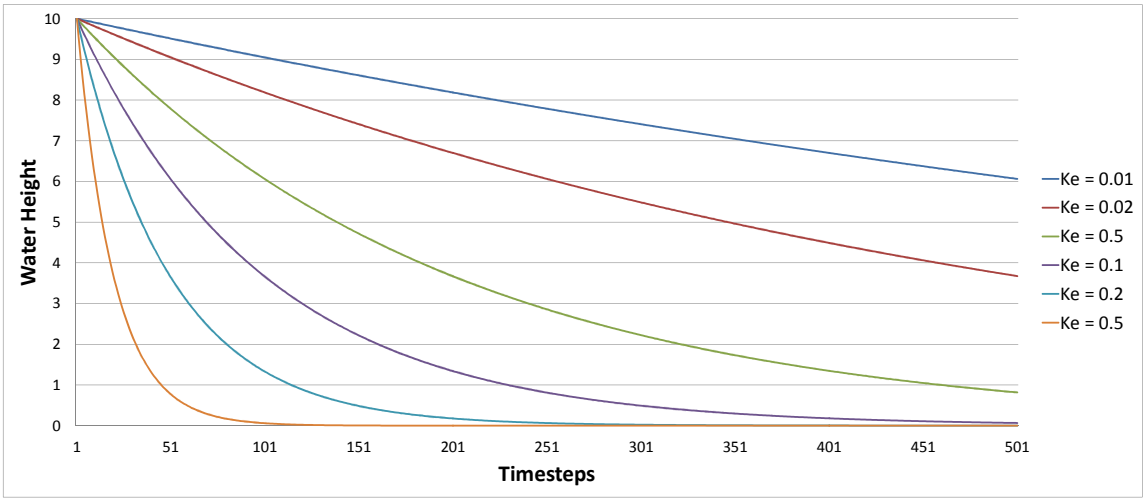


Figure 3.5: Comparison of different evaporation constants Ke, $\triangle t = 0.1$, Start Value = 10.0

takes into consideration the vapour pressure at the evaporation surface e_s as well as the vapour pressure in the athmosphere above the surface e_d. $f(u)$ is a function of the horizontal wind velocity which is empirically determined for a particular location. It is also directly related to the shape and the surface geometry of the evaporating fluid.

Since most of the parameters are assumed to be constant during the simulation, the exponential decrease of the water level can be seen as a coarse approximation of the surface area's influence in evaporation. Although the assumption that a large water height is directly related to a big water surface does not always apply (e.g. deep riverchannels) and the formula has no direct physical background, it is sufficient and fast to implement for this kind of erosion simulation.

4 A Brief Introduction to OpenCL

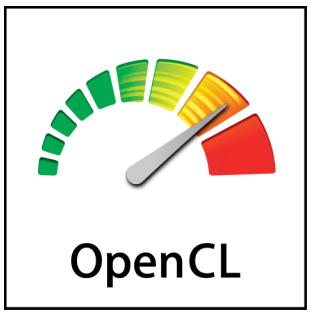


Figure 4.1: OpenCL Logo - Trademark of Apple Inc.

OpenCL (Open Computing Language) is an emerging framework for highly parallel computing applications. It allows the utilization of heterogenous devices (CPUs, GPUs, DSPs) for various computing tasks in an easy to use environment.

Initially developed by Apple the first proposal was finalized in cooperation with technical teams from AMD, IBM, Intel and nVidia and submitted to the Khronos Group for standardization in mid 2008. The Khronos Group released the specification for OpenCL 1.0 on Dezember 8th, 2008. The first specification update OpenCL 1.1 was released in June 2010. The update adds a lot of important features e.g. new datatypes, buffer improvements when working with multiple devices, an enhanced event system, additional built-in functions as well as an improved OpenGL interoperability. OpenCL 1.2 was released the following year on November 16th 2011 and introduced several new features as well as other improvements. The interoperability with DirectX and OpenGL was extended again. Further a feature called "device partitioning" was introduced, allowing applications to partition a device into sub-devices. The compiling and linking of objects was also revised to allow the creation of OpenCL libraries. The ability to include built-in kernels which represent the capabilities of specialized device components gives interesting new opportunities. OpenCL 1.2 also introduced the possibility to create arrays of images [1]

The first full implementation of OpenCL 1.0 was released by Apple with their MacOS X Update "Snow Leopard" on August 28th, 2009. Other implementations from other companies followed in that year. Since the beginning of 2012 most of the major companies (nVidia, AMD, Intel, IBM, ...) provide and support an OpenCL 1.1 implementation. Previews of upcoming OpenCL 1.2 implementations are available as well. [2] [3] [4]

While the refinement and further development is done by the Khronos *Compute Working Group*, the trademark rights are held by Apple.

4.1 Motivation

One of the main ideas behind OpenCL is best described by the term "data parallel execution". For example consider an image which needs to be processed. In a simple single threaded program we need to write a loop which executes our image processing function on each pixel or group of pixels

[1] http://www.khronos.org/opencl/
[2] http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/
[3] http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx
[4] http://developer.nvidia.com/opencl

one after another. A recent midclass Intel Quad Core CPU[5] provides 4 hardware threads while a current AMD GPU[6] actually has a total of 224 stream processors available for computation. Wouldn't it be nice to utilize the full power of the GPU?

In the recent past, various solutions have already been invented to overcome this problem but they rather introduced other barriers e.g. nVidia's CUDA as well as older AMD Stream SDKs are only available for the manufacturers own devices. Others like the nVidia Cg Language are built on top of programmable shaders and are more difficult to handle and strictly limited to GPU devices. OpenMP is available for x86 CPU's but lacks the support of other devices (e.g. GPUs, DSPs, ...).

OpenCL tries to step in at this point and introduces a cross platform/cross device framework which allows developers to write high performant portable code and run it on all OpenCL capabable devices.

4.2 Overview

To achieve these goals OpenCL's platform model incorporates a Client-Server model. In OpenCL terminology the Host (Client) runs the Host Code and uses Kernels which are executed in parallel on the OpenCL Devices (Servers). To communicate with the devices a vendor specific OpenCL Platform runtime which provides the standardized OpenCL API as well as libraries is needed (e.g. AMD APP SDK, Intel OpenCL SDK, nVidia GPU Computing SDK, ...) [30] [31].

The Host Code can be written in any programming language and is managed by the operating system. Its main purpose is to interface with, manage and submit work to the computing devices.
A Device can be any CPU, GPU or other accelerator hardware which is supported by the used OpenCL runtime, called OpenCL Platform.

Each Computing Device consists of compute units which themselves consist of processing elements (see figure 4.2). For example: the Intel Core i5 Quad Core CPU is a computing device with four compute units and can execute four calculations in parallel on its multiple Integer, Floating Point and/or SIMD processing elements whereas the former mentioned AMD GPU has a total of 14 available compute units where each unit consists of 16 processing elements [30] [32].

Work Groups & Work Items

OpenCL supports 1- to 3-dimensional problem domains (called NDRange). Each element in this n-dimensional domain is called a Work Item. Work Items are grouped into Work Groups which have their own local dimensions. Each Work Group shares local memory and synchronization mechanisms and is guaranteed to be executed on one Compute Unit (see figure 4.3).

When looking for example at the hardware implementation of OpenCL on the Radeon 6870 GPU architecture [32] each Work Item runs on a Processing Element which is part of a bigger organizational structure - the Compute Unit. To hide latencies, up to 4 Work Items are pipelined

[5] Intel Core i5 750
[6] AMD Radeon HD 6870

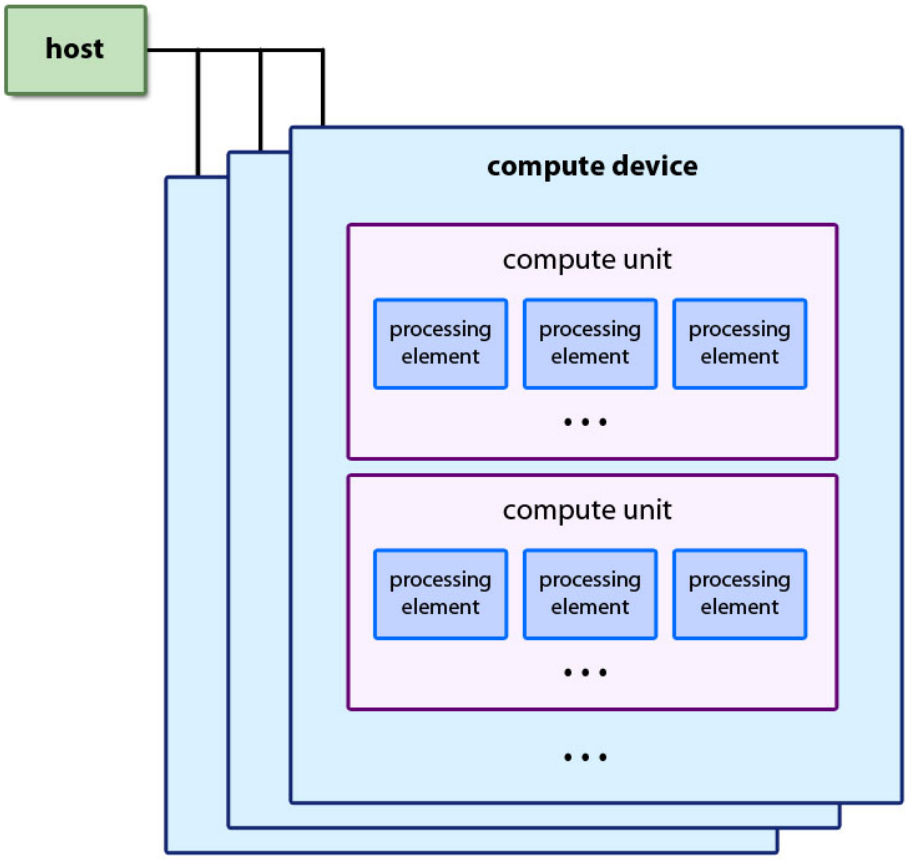


Figure 4.2: OpenCL Platform Architecture

on the same Processing Element. As mentioned before every Compute Unit has 16 Processing Elements. The optimum Work Item count per Compute Unit is therefore given as 16 x 4 = 64 Work Items which is called a "Wavefront" (or "Warp" in nVidia Terminology). This leads to the conclusion that an optimal Work Group Size for this architecture is always a multiple of 64 x N [32]. Although a single Processing Element of the Radeon HD6870 is able to compute up to 5 instructions per clock in theory, it is not always possible for the compiler to generate an instruction stream which fully utilizes the processing elements of the VLIW5 architecture [32].

4.3 OpenCL Memory Model

On the Host-side the memory is managed by the Operating System as usual. The OpenCL Context on the other side has different memory regions and needs to be handled with care (see figure 4.4).

First of all there is global memory space which can be accessed by all devices and all workgroups within the same context. The global memory access is not synchronized in any way and must be handled by the developer. Alongside the global memory space is the memory for constants which is also kernel wide accessible. Global memory is not only the biggest available memory storage

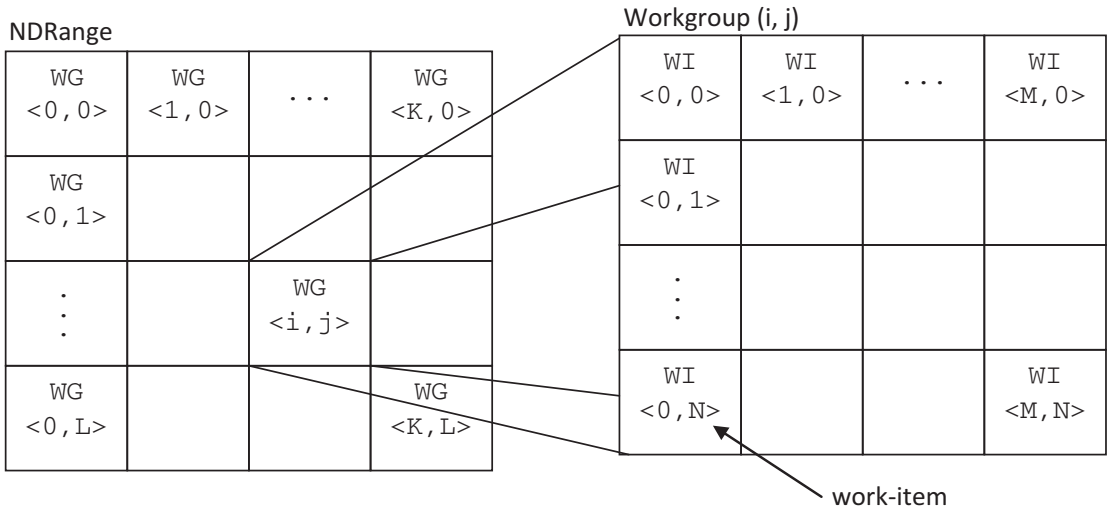


Figure 4.3: OpenCL Work Groups & Work Items (source: [31], page 19)

but also slower compared to the later described other memory variants [31].

Following the design of current GPU hardware the next memory in the hierarchy is a Work Group's local memory which is shared across all Work Items in the group. Technically every Work Group gets assigend to a Compute Unit where each Processing Element executes a Work Item at runtime. Therefore the Work Group's local memory corresponds to a Compute Units local memory. In case of the Radeon 6870 this maps to 32kB of Scratchpad memory [31][32].

Finally the private memory is memory that is only accessible by an individual Work Item. Local kernel variables as well as nonpointer kernel arguments are private by default. Usually the private memory is mapped to local registers which is the fastest possible on-chip memory available. If there's not enough space in the registers the memory must be mapped to global memory, which has a much higher access latency and should therefore be avoided [31][32].

Memory transfers from the host to the context/device memory and vice-versa must be explicitely triggered by the developer.

4.4 OpenCL Execution Environment & Objects

There exist several Objects in OpenCL to simplify the Setup process as well as the handling of different kernels on different devices and the interoperability with the Host application.

4.4.1 Overview

Setup

- Platform - Vendor-specific Runtime Support
- Context - Device & Buffer Management
- Devices - GPU, CPU, DSP, ...

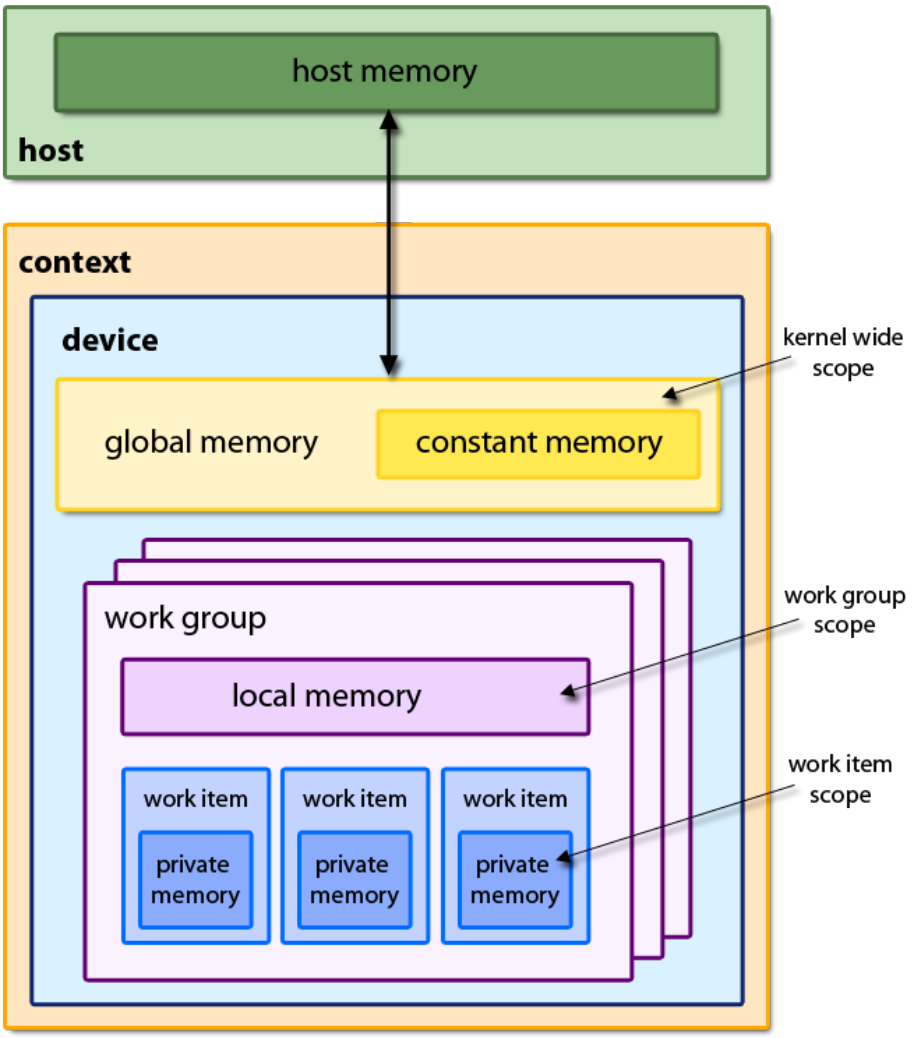


Figure 4.4: OpenCL Memory Model

Memory

- Buffers - Blocks of memory, Arrays
- Images - 2D or 3D images/textures

Execution

- Command Queues - Used to submit work or set/get buffers to or from the device
- Kernels - Execution Primitives
- Programs - Collections of Kernels
- Events - for Synchronization/Profiling

4.4.2 Setup

Platform

The starting point for the initialization of an OpenCL Application is clearly the OpenCL Platform Object. Since multiple platforms can coexist in the same operating system environment, it is mandatory to choose a specific platform at startup. The platform provides the vendor-specific facilities in which the context and all other following OpenCL Objects are constructed and the CL-Kernel source code is compiled. As mentioned in section 4.2 not every platform can communicate with every OpenCL capable device in the system (e.g. the Intel OpenCL Runtime only works with CPUs that support SSE4.1 or SSE4.2 - no GPU support) [31].

Context

The environment where the information of the used devices, the host platform and the allocated buffers are managed is called OpenCL Context. A Context also keeps track of the available Programs and Kernels that are created for a distinct device. A context is always created for a specific platform. Since OpenCL 1.1 a built in OpenCL extension is provided by the Khronos Group to make it possible to share Buffers with an OpenGL or DirectX context which is achieved by specifying properties during the creation of the OpenCL Context [31].

Devices

A Computation Device can be any device which is supported by the used OpenCL runtime. Usually the device will be a GPU or the systems CPU. If the system incorporates multiple graphic cards, each card shows up as a distinct OpenCL Device [31].

4.4.3 Memory

Memory on an OpenCL Device is represented by Buffer respectively Image Objects. They are used to reserve memory space in a given context. Internally they can be thought of as pointers to arrays in the device's memory. Some considerations need to be made, since images can profit from special image processing hardware which can be found on e.g. graphics processing hardware. Since Memory Objects are linked to contexts it is up to the implementation at which exact time a Memory Object is copied to the memory of a device [31].

Buffers

To create a Buffer the size and the Context have to be known in advance. Since the buffers are managed by the Context they are visible to all devices that are associated with a Context. Access to the Buffers can be restricted by supplying flags (read only, write only, read write) to the allocation function. To initialize a Buffer a pointer to an array in the host memory can be supplied at creation time [31].

Images

Image objects are very similar to Buffers but abstracted from the method of storage to allow device-specific optimizations (e.g. texture lookup hardware). Support for Images is optional, but supported by all major vendors at the moment. As mentioned before, access to the memory

is abstracted. Access functions instead of direct array access have to be used to read or write from Image Objects. Furthermore Images have a fixed format which is specified at creation time (e.g. channel ordering, used elements, ...). Similiar to texture samplers in GPU shader programs, Sampler Objects are used to define out-of-bounds handling or interpolation methods when reading from an Image. Since the real hardware implementation is hidden from the programmer it is possible that adjacent data elements are not contiguously laid out in memory [31].

4.4.4 Execution

Command Queues

To utilize an OpenCL Device a Command Queue has to be created for it first. All commands which should be executed by the Device need to be submitted to this Queue. A Command Queue is always associated with one specific Device and Context. A Queue can be an "in-order" queue (default) where the commands are processed in the order they where pushed onto the queue or an "out-of-order" queue where the OpenCL implementation can reorder the commands to allow certain optimizations [31].

Kernels

An OpenCL Device executes compiled OpenCL C Code (a derivative of the C99 standard) in chunks of so called Kernels. A Kernel is the basic unit of a device executable and is similar to a C-Function but it will get executed in parallel on a OpenCL Device. Every OpenCL program needs to be compiled for a specific computing device [31].

Programs

Programs are collections of one or more Kernels and other support functions. Only Kernel-functions can be invoked from the Host application directly. Before it is possible to use the Kernels contained in a Program it is necessary to compile the OpenCL source code. All major vendors do not compile the source to machine code directly. Instead an intermediate language based on a language independant instruction set is generated. That way it is easier to compile binary code for different platforms (e.g. CPU's need x86 Code, AMD GPUs use another intermediate language called IL as input, whereas nVidia use their CUDA language as intermediate output). Since the introduction of OpenCL 1.2 it is also possible to write libraries which can be linked in during the Program Build phase [31].

Events

If the used queue is an "out-of-order" queue it is occasionally necessary to set execution dependencies between kernels. This is where Event Objects come into play. Therefore the OpenCL "clEnqueue*" commands take a list of events as an extra parameter which need to be completed before the execution of the command starts. In addition it is also possible to get an Event Object for the current enqueued command. Events are also heavily used for profiling purposes.

Because of the sophisticated event model system, it is also possible to use OpenCL for task parallel computations [31].

4.5 Execution Flow

After obtaining the OpenCL Platform a typical Application would create the OpenCL Context and a Command Queue for a specific device in a specific Context. The next step is to create the Memory Objects in the given Context and fill them with data. After that the program's source code must be built for the used device. If the compilation was successful the default arguments of the kernel can be set.

The upload and the download of data must be issued manually by pushing enqueueRead* or enqueueWrite* commands to the command queue.

To run an OpenCL Kernel it is necessary to specify the problem domain as explained in section 4.2.

Lets assume a two-dimensional computational domain (e.g. an image) where the total number of Work Items is defined by the global problem dimension (the image size: 800x600 = 480.000 work items). Each of these Work Items execute the image processing Kernel function. Depending on the underlying hardware multiple Work Items can be processed in parallel (see figure 4.5).

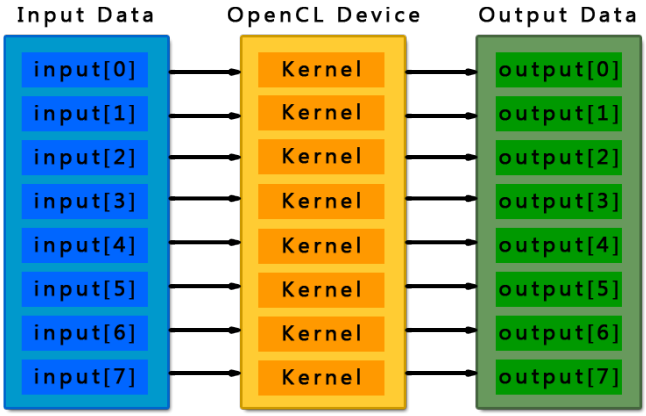


Figure 4.5: OpenCL Example Work Distribution

Because of the scheduling of Work Groups and the maximum Work Group size imposed by the hardware, it is only possible to synchronize the Work Items inside the same Work Group using Barriers or Memory Fences. If coarser synchronisation is desired Event Objects or calls to "clFinish" must be used instead to ensure that a Kernel has been executed (see figure 4.6).

4.6 OpenCL C

OpenCL Kernels are written in OpenCL C, a derivative of the C99 standard which was extended to meet the needs of parallel programming.

In addition to common C99 datatypes like char, int, float, etc. ... OpenCL C also supports the following types [30]:

- half: 16 bit floating point type
- vector data types: all datatypes can also be used as vectors with 2, 4, 8 and 16 elements e.g. float4, int8, etc. ... (since OpenCL 1.1 it is also possible to use 3-dimensional vectors)

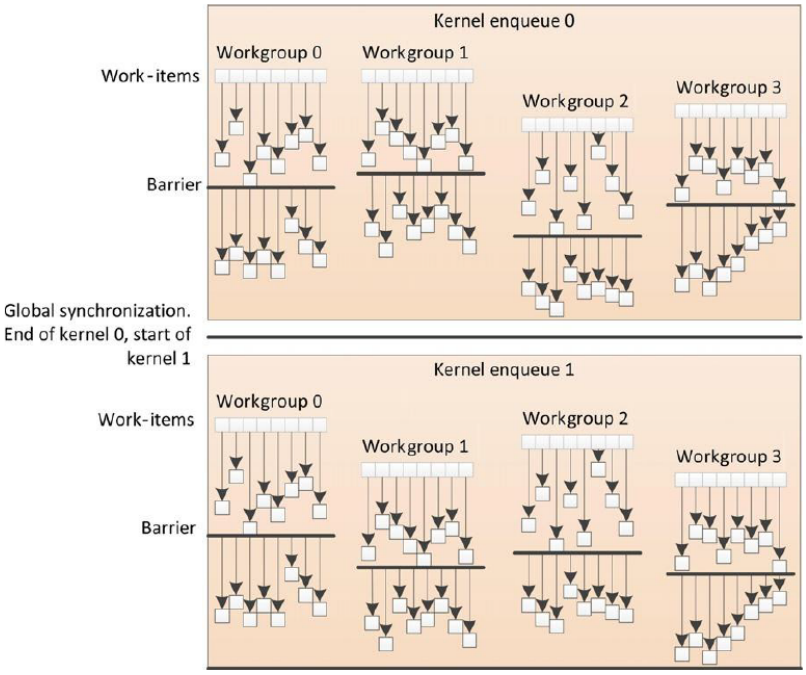


Figure 4.6: OpenCL Work Group & Kernel Bound Synchronization (source [31], page 91)

- image1d_t: 1-dimensional image (since OpenCL 1.2)
- image2d_t: 2-dimensional image
- image3d_t: 3-dimensional image
- sampler_t: sampler which is used to read values from an image
- event_t: an event handler

Vector literals also play an important role in OpenCL C. With them it is possible to create vectors from a set of scalars or other vectors. They can be used in initialization statements or as constants. If only a scalar values is specified it gets replicated to all vector components [30].

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uint4 u = (uint4)(1); // u = (1, 1, 1, 1);
float4 f = (float4)( (float2)(1.0f, 2.0f), (float2)(1.0f, 2.0f) );
```

OpenCL C also supports the use of vector component specifiers. With the used vector adressing it is possible to *swizzle* the components as well as to replicate them. It is also valid in OpenCL C to use numeric indices (".sNNN...") [30].

```
float4 a, b, c;
a.wzyx = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
b.xyzw = a.xxyy;
c = a.s0123;
```

Another alternative of handling vector components is the ".lo" and the ".hi" suffix which refers to the lower respectively upper half of a given vector [30].

```
float4 vf;
float2 low = vf.lo;
float2 high = vf.hi;
```

Type casts can be performed just like in C with the exception that explicit casts between vector types are not legal. For this purpose conversion functions "convert_<dest type name>(srctype)" where defined [30].

```
uchar 4 u;
int 4 c = convert_int4(u);
```

To reinterpret types as other types the "as_type()" function can be used [30].

```
float f = 1.0f;
uint u = as_uint(f);

float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
int4 i = as_int4(f);
```

To tell the OpenCL compiler exactly where a variable is residing, or a pointer where it is pointing at, address space qualifiers need to be used. The qualifiers correspond directly to the different memory locations as described in section 4.3: *__global, __constant, __local, __private*. If no prefix is specified *private* address space is assumed automatically [30].

All operators that are applicable to scalars are also applicable to vector datatypes. To simplify the computational use, some additional mathematical functions where incorporated as well e.g. sin, cos, min, max, etc. [30].

A Kernel function must be marked with the attribute __kernel. The basic form of a Kernel is therefore given as

```
__kernel myFirstKernel( __global float * array )
{
   ...
}
```

4.6.1 Limitations of OpenCL C

However there exist some limitations for OpenCL C too [30]:

- Pointer arguments to Kernel Functions must be __global, __constant or __local
- Pointer arguments to Kernel Functions cannot be pointers to pointers
- Function pointers are not allowed
- Image-type variables (e.g. image2d_t) can only be specified as arguments to a function
- Samplers can neither be declared as arrays, pointers, local variables or return values nor can they be arguments to non Kernel functions or members of structs

- Bit-fields are currently not supported
- No variable-sized arrays
- No variadic macros and functions
- No library functions from various C99 standard headers (assert.h, stdio.h, stdlib.h, string.h, time.h, signal.h ...)
- No recursive functions
- Kernel functions can only have the return type "void"
- Irreducible control flow (e.g. endless loops with break statements) is illegal in OpenCL versions prior to 1.2 and implementation defined in 1.2
- Writes to a pointer (array) of integral types or structs that contain integral types less than 32-bits in size are not supported in OpenCL versions prior to 1.2
- Arguments which are structs are not allowed to contain OpenCL objects

4.7 Basic Example

After giving an overview of the features of OpenCL a short example on basic initialization and OpenCL C Code Structure is presented to get a better understanding on how the different parts work together.

The Host Application Initialization Code code can be written directly with the OpenCL's C-style API or with a C++ Wrapper (cl.hpp). Since the implementation of this master thesis was developed with C++ the example code also uses the C++ Wrapper.

First of all an OpenCL Platform and a proper device must be selected. The functions accept standard template library (STL) vectors because it is possible to install more than one platform in the operating system environment and to have more than one device with the given type in a personal computer (e.g. SLI, Crossfire, ...). To keep the application simple we choose the first available platform and ask for all available devices with type GPU. The next step is to define properties for the context which is going to be created afterwards. In our case we only want to specify the platform the context should use. At last a Command Queue for the chosen device is created within the context.

```
// get the available platforms
cl::vector< cl::Platform > platformList;
cl::Platform::get(&platformList);

// get all available devices with type GPU
cl_device_type devicesToUse = CL_DEVICE_TYPE_GPU;
cl::vector<cl::Device> allDevices, chosenDevices;
platformList[0].getDevices(devicesToUse, &allDevices);

// simply choose the first available device
```

```
chosenDevices.push_back( allDevices[0] );

// define context properties
cl_context_properties contextProperties[] =
    { CL_CONTEXT_PLATFORM, (cl_context_properties) (platformList[0])()
      ,
      0 };

// create the context with the chosen devices
context = cl::Context(chosenDevices, contextProperties);

// obtain a commandqueue for the first device
// within the given context
commandQueue = cl::CommandQueue( context, chosenDevices[0] );
```

Listing 4.1: OpenCL Basic Startup Code

In the next step the OpenCL Kernel sources are loaded into a Program Object. To generate binary code out of the sources the Program Object has to be built against the used OpenCL Devices. If there was no error during compilation, a kernel object can finally be obtained from the Program Object. At last some data and OpenCL Buffers are generated and filled with random data. Non changing Kernel Parameters can be set immediately after the Kernel construction with the *setArg* function.

```
// load cl program code to a string
std::ifstream file("example.cl");
std::string prog( std::istreambuf_iterator<char>(file), (std::
    istreambuf_iterator<char>()) );

// construct a program object
cl::Program::Sources source( 1, std::make_pair(prog.c_str(), prog.
    length()+1) );
cl::Program program(context, source);

// build program for the devices in chosenDevices
program.build(chosenDevices);

// after the program has been successfully built obtain a kernel
cl::Kernel kernel(program, "add");

// create input data
int *id1 = new int[1024];
int *id2 = new int[1024];
for(int i=0; i<1024; ++i)
{
  id1[i] = rand();
  id2[i] = rand();
}

// create output array
int *out = new int[1024];

// create opencl buffers for working
```

```
cl::Buffer clInput1(context, CL_MEM_READ_ONLY, sizeof(int)*1024);
cl::Buffer clInput2(context, CL_MEM_READ_ONLY, sizeof(int)*1024);
cl::Buffer clOutput(context, CL_MEM_WRITE_ONLY, sizeof(int)*1024);

// set the buffers as kernel arguments
kernel.setArg(0, clInput1);
kernel.setArg(1, clInput2);
kernel.setArg(2, clOutput);
```

Listing 4.2: Kernel Generation

At this point a Command Queue as well as the Kernel and some data to work with have been created. To start working the Command Queue needs to know what to do. For this simple example we instruct the queue to write the input data to our input buffers. Since we specified CL_TRUE as second paramter the call is blocking until the buffer has been written to the device.
Then we enqueue the "add" Kernel and wait for its execution. For this call we tell the kernel to execute on a global problem dimension of 1x1024 elements (-> NDRange(1024)). If the Work Group size parameter is omitted the OpenCL implementation will determine an appropriate Work Group size.
Finally the calculated values are read back from the CL-Device to the "out" array.

```
// enqueue input buffers blocking
commandQueue.enqueueWriteBuffer(
  clInput1, CL_TRUE, 0,
  (size_t) sizeof(int)*1024, id1 );
commandQueue.enqueueWriteBuffer(
  clInput1, CL_TRUE, 0,
  (size_t) sizeof(int)*1024, id2 );

// enqueue the kernel
commandQueue.enqueueNDRangeKernel(
  kernel,
  cl::NullRange,
  cl::NDRange(1024),
  cl::NullRange );

// wait for the queue to finish all work
commandQueue.finish();

// read back the results to the out array blocking
commandQueue.enqueueReadBuffer(
  clOutput, CL_TRUE, 0,
  (size_t)sizeof(int)*1024, out );
```

Listing 4.3: Get the Queue to Work

The Kernel itself performs an addition between equally indexed elements of $clInput1$ and $clInput2$ and writes the result back to $clOutput$. This is achieved by obtaining the array index first. In this example's 1-dimensional case, a call to $get\_global\_id(0)$ yields directly the array index.
If a 2-dimensional NDRange would have been specified in the *enqueueNDRangeKernel* call, the Kernel's array index would be computed as $tid = get\_global\_id(0) + get\_global\_id(1) *$

$get\_global\_size(0)$.

With the computed id the input vectors can be accessed like normal arrays.

```
__kernel void add(
  __global int* clInput1,
  __global int* clInput2,
  __global int* clOutput )
{
  uint tid = get_global_id(0);  // get the array index
  clOutput[tid] = clInput1[tid] + clInputt[tid];
}
```

Listing 4.4: The OpenCL Kernel "add"

5 Implementation of a Test Framework

This chapter focuses on the implementation of a test framework which utilizes the algorithms explained in chapter 3 before.

5.1 Host Application

The Host Application of the Test Framework is completely written in C++. Its main purpose is to initialize, manage and coordinate the used frameworks.

To interface with the window manager and the input handling of the system a GLUT variant called *freeglut* is used. freeglut is nearly a full replacement for the GLUT API but also comes with some improvements, since the last version of GLUT was released in 2001. In contrary to GLUT, freeglut is an open source project and uses the *MIT/X Consortium License* which allows free distribution as well as modifications [1].

A minimal GUI toolkit called AntTweakBar[2] was used to simplify the parameter input to the simulation stage. AntTweakBar is a lightweight and easy to use C/C++ library which can be integrated in many rendering environments.

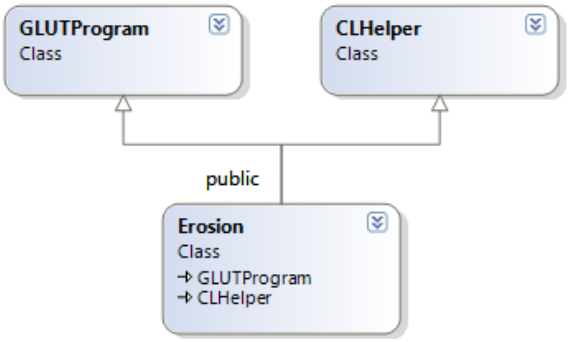
For convenience a wrapper class *GLUTProgram* was written. It handles many of GLUTs functions and channelizes them in class member functions which can be overloaded in implementing classes. It also adds standard features like a camera and a stopwatch for the time measurement in the update loop. The same was done for the OpenCL functionality in the class *CLHelper* including the initialization from an existing OpenGL Context as well as the creation and management of shared Buffers which can be used in both OpenGL and OpenCL contexts. Additionally *CLHelper* provides Facilities to load and build OpenCL source code and obtain Kernel Objects after a successful build.

The implementing class *Erosion* inherits these two convenience wrappers and extends them where needed.

Program Startup Sequence:

- Create Erosion class instance
- Call Erosion's Init function which is a start point for further initializations
  - Parse commandline parameters
  - Create OpenGL environment

[1] `http://freeglut.sourceforge.net/`
[2] `http://www.antisphere.com/Wiki/tools:anttweakbar`



  - Create OpenCL environment from OpenGL environment
  - Init other libraries
  - Build GUI
  - Load heightmap and setup used buffers
  - Load textures
  - Load GLSL shaders for visualization
  - Set desired OpenGL states (Camera, Light, etc. ...)

- Hand over control to the glutMainLoop which calls the main *UpdateScene* function of the running GLUTProgram as fast as possible

The main work tasks are then handled in the *UpdateScene* function of the *Erosion* class.

Like in every physics simulation a stable framerate for the simulation loop is mandatory. Spikes in the temporal dimension could introduce instabilities in the water simulation. To avoid this, the computation function uses a fixed internal timestep m_dt which is independant from the framerate.

Besides the timestep m_dt we also do not want the visualization to vary in speed. To accomplish this the simulation update function *RunComputations* gets called at a fixed but different rate $m\_SimulationTime$ (see listing 5.1) [3].

```
static double sumTime = 0.0;
double frameTime = m_Clock.GetLaptimeSeconds();
sumTime += frameTime;
if( sumTime > 1.0 )
{
  sumTime = 0.0;
  RunComputations();
}
else
{
  while( sumTime >= m_SimulationTime )
  {
```

[3] http://gafferongames.com/game-physics/fix-your-timestep/

```
      sumTime -= m_SimulationTime;
      RunComputations();
    }
  }
```

Listing 5.1: Simulation Update Rate

If the update function is not called over a longer period of time (e.g. the window is moved around) the $sumTime$ would increase drastically and never return to a normal execution. Therefore it is capped at 1 second. Normally the simulation should take far less than 1 second. If this is not the case for some reason *RunComputations* gets executed every frame only once, which results in a reduced simulation speed and a non constant external simulation rate. Nevertheless this is the only choice if the simulation takes too much time.

5.1.1 Tool Classes

Additional tool classes where written to simplify the handling of the scene further. In this subsection the most important classes will be briefly explained:

- StopWatch
- ShaderProgram
- Camera
- PointLight
- SimpleMarker
- Texture, Cubemap
- EnumGenerator
- Logging

StopWatch

StopWatch is a small class for measuring time beween two code segments.

```
StopWatch simTime;
simTime.TakeTime();
  ... code under test ...
simTime.TakeTime();
cout << "code took: " << simTime.GetSeconds() << " s" << endl;
```

Listing 5.2: StopWatch Use Case

Internally *StopWatch* uses the PerformanceCounters provided by *windows.h*[4]. A small use case example can be seen in listing 5.2.

[4] http://support.microsoft.com/kb/172338/EN-US

ShaderProgram

The ShaderProgram is the central hub of the object system. ShaderPrograms can incorporate OpenGL GLSL Vertex, Geometry and Fragment shaders. To build a shader set, a new program object has to be obtained from the *ShaderProgram* factory function. After this the different shader files can be added to the *ShaderProgram* object with the function *CompileAndAddShader* by specifying the source file and the shader type. Before linking, shader attributes can be added as needed. As first parameter an *EnumIdentifier* which is described later must be provided. The second parameter is the corresponding variable name in the shader source. If everything is in the right place, the program can be linked with a call to the *Link* function (see listing 5.3).

```
m_Shader = ShaderProgram::NewProgram("Default");
if(  0 == m_Shader->CompileAndAddShader(
  "vertex_std.glsl", GL_VERTEX_SHADER)  )
  return false;
if(  0 == m_Shader->CompileAndAddShader(
  "fragment_std.glsl", GL_FRAGMENT_SHADER)  )
  return false;
m_Shader->AddAttribute(
  ErosionShaderAttribute::POSITION, "in_Position");
m_Shader->AddAttribute(
  ErosionShaderAttribute::NORMALS, "in_Normals");
m_Shader->AddAttribute(
  ErosionShaderAttribute::TEXCOORDS, "in_TexCoords");
if(  0 == m_Shader->Link()  )
  return false;
```

Listing 5.3: GLSL Shader Compilation

During the compiling respectively linking process an error log gets printed to the logfile as well as to the output window.

Camera

The *Camera* class was designed to be controlled by the keyboard (position) and mouse (rotation) and is already built in at the *GLUTProgram* level. It is possible to zoom in and out as well as switch projection matrices from perspective to orthogonal. The *SetupForScene* function takes a pointer to the currently used shader and uses uniform variable names from the header file *ShaderVars.h*.

PointLight

The *PointLight* class is a simple wrapper around the light source parameters. By using the lights setup function *LetThereBeLight* the uniform variables for the current shader are set.

SimpleMarker

SimpleMarker (see figure 5.1) is a simple scene object without a texture and is used to show the current mouse cursor position in 3d space during edit mode.

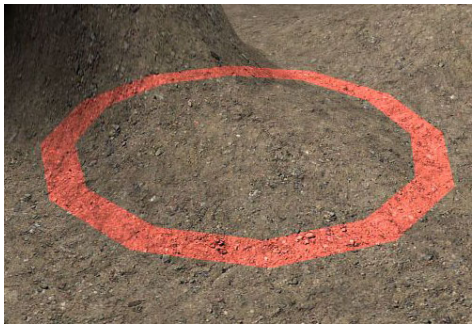


Figure 5.1: SimpleMarker

Texture

Texture encapsulates an OpenGL texture object in addition to the image loading facilities of the DevIL framework. DevIL[5] (former OpenIL) is an OpenGL-style image loading library which comes with support for many image formats (e.g. jpeg, png, psd, tiff, ...). With DevIL it is possible to create an OpenGL texture with very few commands. The *Texture* class also offers the possibility to enable or disable Anisotropic Filtering as well as different texture wrap modes. While the *Texture* class is used for common 2d-images there also exists a class for loading CubeMaps *TextureCubeMap*.

EnumGenerator

The *EnumGenerator* provides macros for an easy and painless generation of enums encapsulated in a namespace with a default name and a string-conversion function e.g. the DefaultShaderAttribute enum (listing 5.4:

```
// in ShaderProgram.h
#define DefaultShaderAttributeEnum \
  (POSITION)(COLOR)(NORMALS)(TEXCOORDS)
DEF_ENUM_HEADER(DefaultShaderAttribute, DefaultShaderAttributeEnum)

// in ShaderProgram.cpp
DEF_ENUM_DATA(DefaultShaderAttribute, DefaultShaderAttributeEnum)
```

Listing 5.4: EnumGenerator

In the background the macro DEF_ENUM_HEADER generates a class *ENUM* in namespace *DefaultShaderAttribute* with the instances POSITION, COLOR, etc. which also live in the *DefaultShaderAttribute* namespace. The *ENUM* class implements operators like "==" and "!=" and does auto convert to an *EnumIdentifier* which also has those operators defined. The *ENUM* class also provides a *String()* function which returns the name of the enum. The string is automatically obtained from the given names (POSITION, COLOR, ...) and lives in a class static variable to save memory.

[5] http://openil.sourceforge.net/

5.1.2 Usage

Figure 5.2: After program startup

At the commandline several parameters can be specified to override certain default values:

--help	print all possible arguments and exit
--loglevel arg	set log level between 0 (fatal) - 4 (detail)
--width arg	change the window width (default: 1280)
--height arg	change the window height (default: 800)
--heightmap arg	the heightmap to load at startup
--clplatform arg	override platform to use (default: "AMD APP", in case the specified platform is not found the first available is used)
--cldevice	override opencl device type, valid options: 0=GPU, 1=CPU (default: 0)

If called without parameters a default map gets loaded and a screen like in figure 5.2 is shown.

On the left the *StateBar* where different runtime switches can be changed or turned on/off can be seen. On the right resides the *VarBar* where algorithmic settings as well as other parameters of the simulation can be tuned and/or viewed.

InputState

Possible modes are: *Navigate, ModifyHeight, ModifyWater, ModifySuspendedSediment* and *ModifyWaterSources*.

The *InputState* determines the active action for the left mouse button. If the state is *Navigate* no action is carried out when clicking the left mouse. Internally this also switches off calls to

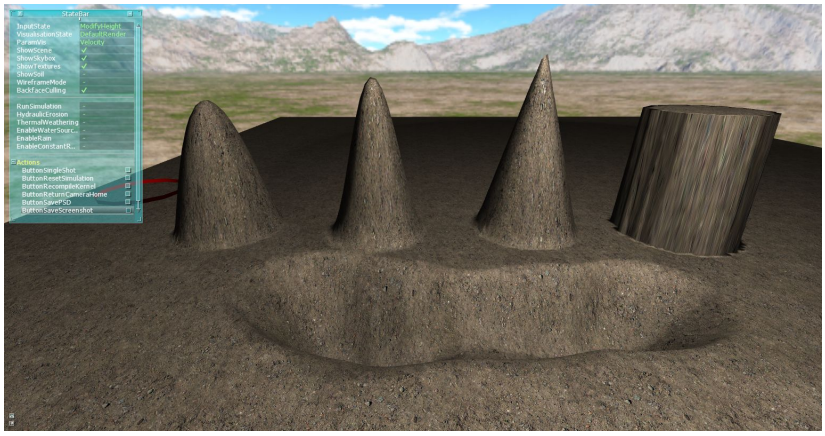


Figure 5.3: Edit Mode Functions

the *EditMode* Kernel. If one of the *Modify** states is enabled, the mouse cursor controls a 3-dimensional representation of the mouse position (*SimpleMarker*) in the scene. The size of the mouse marker can be changed by holding the *ALT* key on the keyboard while scrolling the mouse wheel or by changing the *EditModeRadius* in the EditMode group of the *VarBar*. When holding *STRG* in edit mode, the position of the marker is locked. Per default a left click adds an amount of *EditModeScale* with the given function from *EditModeFunction* to the set Modify* value (e.g. *ModifyHeight* adds the amount to the heightfield). If *ALT* is used together with the left mouse click the amount is instead removed from the value array. Figure 5.3 shows the different edit mode functions: cos, cos^2, $linear$, $constant$. The not shown $smooth$ function minimizes the differences in height.

VisualisationState

The *VisualisationState* can be: *DefaultRender*, *DebugParamVis*, *DefaultDebug* and *DebugWithNormals*. The default mode is *DefaultRender* where the standard visualisation shader is used. By setting the mode to *DebugParamVis* it is possible to change the camera view from perspective to orthogonal mode and observe the algorithms output arrays visually (see figure 5.4). The desired output array can be selected via the *ParamVis* dropdown. If *VisualisationState* is not *DebugParamVis* the *ParamVis* selection has no effect. When selecting the Visualisation states *DefaultDebug* and *DebugWithNormals* an additional debug shader is switched on, where the velocity (and the normal) vectors are drawn for every grid cell in the scene.

Navigation

To move around in the scene the keys *w, a, s, d* (forward, backward, left and right) can be used. Up- and downward movement can be achieved with the keys *space* and *shift*. The Camera can be rotated by holding down the right mouse button while moving the mouse.

Render and Simulation Switches

The other buttons in the upper half of the *StateBar* consist of switches which enable certain render features. Via the *Show** parameter family the rendering of the scene, textures or skybox

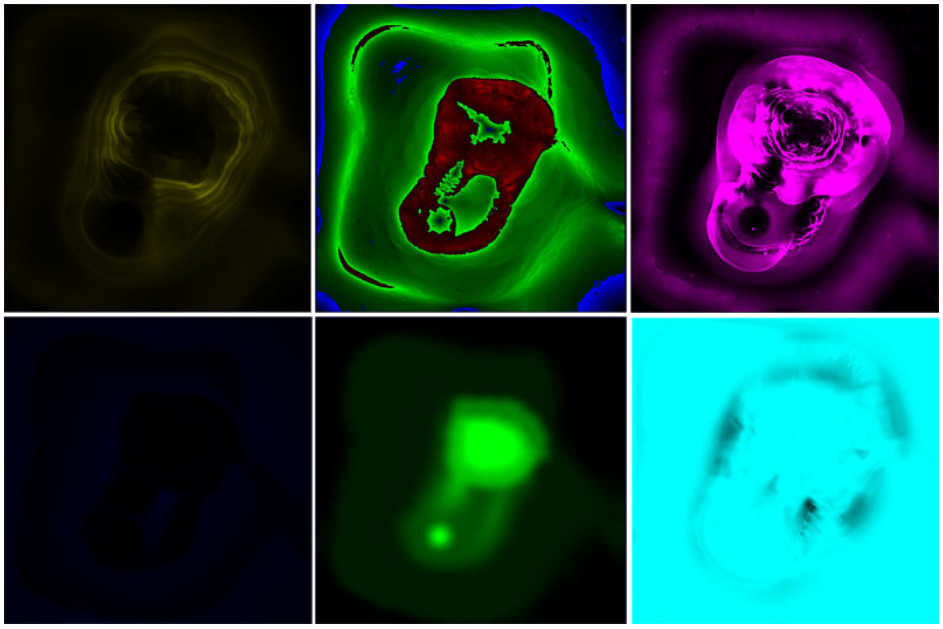


Figure 5.4: Parameter Visualisation

can be turned on or off. *ShowSoil* toggles a color overlay of suspended sediment (cyan) and accumulated soil (red). The obligatory wireframe and backface culling switches are used to toggle wireframe rendering respectively backface culling. The switches *RunSimulation*, *HydraulicErosion*, *ThermalWeathering*, *EnableWaterSources*, *EnableRain* and *EnableConstantRain* enable or disable parts of the erosion simulation. After the simulation switches a group of function buttons where integrated to trigger various features of the implementation

SingleShot	run a single simulation step and stop afterwards
ResetSimulation	reloads the heightmap from the file and resets all simulation arrays
RecompileKernel	triggers a recompilation of the opencl Kernels (useful for on-the-fly changes)
ReturnCameraHome	returns the camera to the home position and resets rotation
SavePSD	saves the heightmap to a psd file in the textures directory (the new name consists of the current heightmap's name plus a sequential number)
SaveScreenshot	saves a screenshot of the window in the current executing directory
EnableFullScreen	switches the current window to fullscreen mode

Simulation Information and Variables Bar

Placed on the right side of the screen is the *VarBar*. In the top section simulation information like *Frames per Second (FPS)* of the main loop and *Simulation time (ms)* of the computations

are presented. *Iterations* shows the current iteration index of the simulation. With the *StopAtIteration* field it is possible to specify an iteration index where the simulation should stop at. *CellGridSize** and *PointsSideSize** show sizes of the arrays on which the algorithms are running. *Cells* is the number of cells in the simulation. *MarkerPosition* displays the current position of the MouseMarker if an edit mode is active.

In the next section main simulation switches can be set: *KernelBuild* displays the current mode with which the running Kernels where built. Other options besides *Standard* are *Debug*, where all optimizations are switched off and *Optimized*, where all possible optimizations are switched on.

The dropdown *WaterSimulationQuality* shows the current used pipe neighbourhood. Possible states are *FourPipes* and *EightPipes*.

The next parameter *AdvectionMethod* defines the advection Kernels which are executed for the advection step of the hydraulic erosion simulation. The implemented approaches are discussed in chapter 3 - possible options are *BackwardEuler* and *MacCormack* advection schemes.

In the heightmap field the currently loaded heightmap is shown. By typing in a filename from the "textures" directory a new heightmap can be set.

The parameter *SimulationTime* controls the variable *m_SimulationTime* introduced in section 5.1 which influences the execution rate of the simulation at which a lower value results in a higher simulation frequency.

The *AlgoParam* group contains all changeable parameters from the simulation. They can be directly tuned from the user interface and are immediately applied in the simulation:

TimeStep	the used simulation timestep $\triangle t$
MinErosionAngle	minimum assumed angle for sediment capacity calculation
Kc	sediment capacity scaling factor
Ks	global dissolve scaling factor
Kd	global deposition scaling factor
Kdmax	sediment capacity height limit, if water height is less than Kdmax the sediment capacity is scaled down by the formula described in 3.35
Kh	hardness scaling factor
Khmin	minimum hardness value
Khrestore	restore rate of hardness when no water is in cell
Talus Angle	talus angle for the thermal weathering simulation
Ka	influence of hardness on the weathering simulation
Kt	thermal weathering scaling factor

CellSize	the simulations horizontal/vertical cell size l_{cell}
PipeLength	the virtual pipes assumed length l_{pipe}
OutflowDampening	maximum outflow flux factor
AddSoilToWater	enables the update of the water height in the erosion-simulation step (see section 3.5)
CorrectAdvection	switches on velocity scaling by cell size for the advection steps
RainIntensity	amount scaling factor which is used for random and constant rain
RainFrequency	scales the frequency of random rain, 1 is the highest possible frequency
RaindropSize	raindrop size multiple
EvaporationRate	water evaporation rate scale

In the *EditMode* section the different edit mode functions, the scale and the radius can be directly set.

Also the current parameters of the *PointLight* & the *Camera* can be changed from the *VarBar*.

5.2 Erosion Simulation

The whole Simulation is designed to run entirely on the graphics hardware. The only point where a memory transfer to the device is involved is at startup or when the heightmap is changed (see section 5.1).

The implementation adapts the model described in chapter 3. The arrangement of the data buffers is as follows

- TerrainHeight - **b**
- WaterHeight - **d**
- SuspendedSediment - **s**
- SuspendedSedimentToggle - flip-flop buffer
- OutFlow - $Q^L, Q^R, Q^T, Q^B, Q^{LT}, Q^{RT}, Q^{RB}, Q^{LB}$
- SoilFlow - $V^L, V^R, V^T, V^B, V^{LT}, V^{RT}, V^{RB}, V^{LB}$
- Velocity - $\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$
- SedimentCapacity - the cell sediment capacity **C**
- AccumulatedSoil - contains the accumulated deposited soil
- HardnessCoefficient - the hardness coefficient **h**

- SoilWetness - mimics the soil wetness
- Normals - generated normals $\vec{n}$
- TerrainPosition - contains the terrain x-y positions - only used for rendering
- RandBuffer - random numbers for the rain simulation
- StaticWaterSources - water amounts for static water sources $constant\_rain$
- SedimentClampBuffer - only used with the MacCormack scheme

A single buffer e.g. TerrainHeight consists of a simple float array, whereas the Outflow Buffer is an array of float8 (vector of 8 floats) per array cell. Summing up these values leads to a total of 156 bytes per single cell. Considering a problem size of 256x256 the simulation needs 10 223 616 bytes or 9.75 Megabytes. Table 5.1 shows the memory consumption for different grid sizes.

Grid Size	Memory
128x128	2.4375 MB
256x256	9.75 MB
512x512	39MB
1024x1024	156MB
2048x2048	624MB

Table 5.1: Memory consumption for different grid sizes

The simulation steps described in chapter 3 were rearranged respectively regrouped to suit the simulation's needs. Additional Kernels e.g. for the edit mode where added. After all the following Computation Kernels where written:

1. Edit Mode
2. Water Evaporation / Increase
3. Rand Buffer Update
4. Terrain Normal Calculation
5. Water Outflow Update
6. Water Height Update / Velocity Field Calculation / Sediment Capacity Computation
7. Erosion - Deposition
8. Sediment Advection
   a) Sediment Advection Euler
   b) MacCormack Scheme
      i. MacCormack Advection Forward
      ii. MacCormack Advection Backward
      iii. Advection Limitation

9. Thermal Weathering Soil Outflow Calculation
10. Thermal Weathering Height Update

Since the user interface provides many possible configuration options, not all Kernels are executed at the same time in every configuration. Figure 5.5 shows an overview of the possible Kernel execution paths. The implementation makes use of the OpenCL Event system although the tested GPU hardware was not able to process the Kernels out of order due to a lack of hardware support.

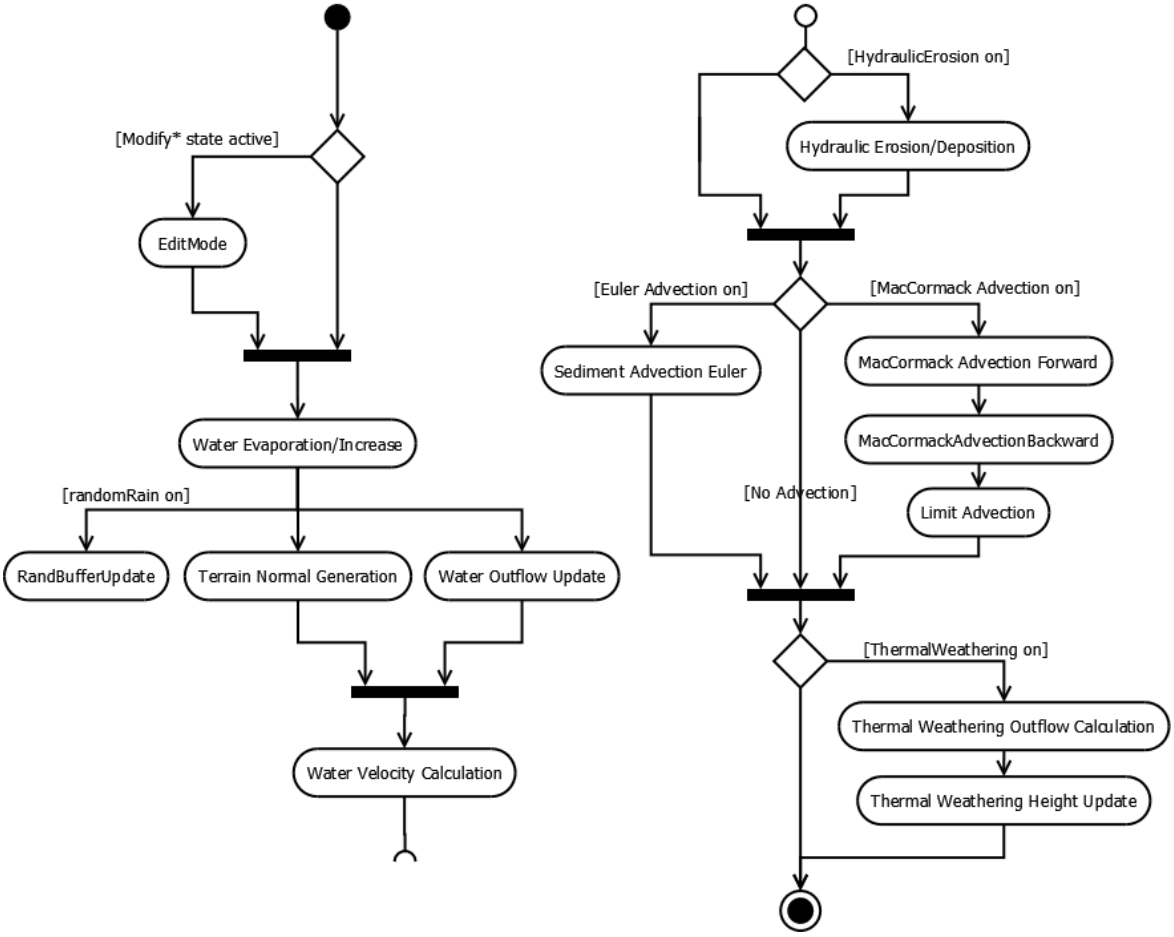


Figure 5.5: Kernel Execution Flow

The reason for the high count of OpenCL Kernels is on the one hand the flexibility to switch on/off different sections of the simulation. On the other hand parts of the algorithm require a fully completed previous step because they rely on values from the cells neighbourhood which can only be guaranteed to be finalized by dissecting the Kernels and synchronize at their boundaries.

5.2.1 Water Evaporation / Increase

```
__Kernel void IncreaseWater( __global float * terrainHeight
                           , __global float * waterHeight
                           , __global float * staticWaterSources
                           , __global float4 * pos
                           , __global float * soilWetness
                           , __global ulong * rand
                           , float dt, float waterOn
                           , float rainOn, float constantRainOn
                           , float rainIntensity, float rainFrequency
                           , uint raindropSize, float evaporationRate
                               )
{
  const uint tid = X + Y * XMAX;

  float d1 = waterHeight[tid];

  // ----- VISUALIZATION PREPARATION
  // write height to pos array
  // height is composed of
  //    * terrain height
  //    * water height
  pos[tid].y = terrainHeight[tid] + d1;

  // ----- EVAPORATION
  d1 = fmax(0.0f, d1 * (1.0f - (evaporationRate * dt)));

  // ----- SOIL WETNESS UPDATE
  float sW = soilWetness[tid];
  if( d1 > 0.0001f && d1 <= 0.8f )
    sW += 100.0f * d1 * dt;
  else if ( d1 > 0.8f )
    sW *= 0.66f;
  else
    sW -= 0.2f * dt;
  soilWetness[tid] = clamp( sW, 0.0f, 100.0f );

  // ----- WATER INCREASE
  float dInc = 0.0f;

  // add static water sources
  dInc += waterOn * staticWaterSources[tid];

  // add random rain water sources
  const uint2 rpos = (uint2)(X, Y) / raindropSize;
  const uint rid = rpos.x + rpos.y * XMAX;
  if( rand[rid] > LKG_M-(LKG_A*raindropSize/rainFrequency))
    dInc += rainOn * rainIntensity * 100.0f;

  // add constant rain sources
  dInc += constantRainOn * rainIntensity;
```

```
  // store d1 as new water height
  waterHeight[tid] = d1 + dt * dInc;
}
```

Listing 5.5: IncreaseWater

The first step is a combination of various substeps. First the position array is updated with the values from the last simulation step. This is part of the visualization preparation. Then the evaporation formula is executed which yields a new value for the water height. At the bottom the water output of the three water sources $constant\_rain$, $random\_rain$ and $spring$ is summed and added to the current water height $d1$. The parameters $waterOn, rainOn$ and $constantRainOn$ are set before the execution of the Kernel and can only have the values 0.0 (disabled) or 1.0 (enabled).

5.2.2 Rand Buffer Update

```
Kernel void UpdateRandBuffer( __global ulong *rand )
{
  uint tid = X + Y * XMAX;
  rand[tid] = (rand[tid] * LKG_A + LKG_B) % LKG_M;
}
```

Listing 5.6: UpdateRandBuffer

The implementation of the RNG update strictly follows the formula from 3.1 and is executed on every cell of the *rand* array if random rain is switched on.

5.2.3 Terrain Normal Generation

```
__Kernel void CalculateNormals( __global float4* pos
                              , __global float4* norm )
{
  const uint tid = X + Y * XMAX;
  int4 position = (int4)(X, X, Y, Y);

  position = clamp( position + (int4)(-1,1,-1,1)
                 , (int4)(0,0,0,0)
                 , (int4)(XMAX-1, XMAX-1, YMAX-1, YMAX-1) );

  // (x+1, y) - (x-1, y)
  float4 x1 = pos[position.y + Y * XMAX] - pos[position.x + Y * XMAX];
  // (x, y+1) - (x, y-1)
  float4 y1 = pos[X + position.w * XMAX] - pos[X + position.z * XMAX];

  // (x-1, y-1) - (x+1, y+1)
  float4 x2 = pos[position.x + position.z * XMAX]
            - pos[position.y + position.w * XMAX];
  // (x+1, y-1) - (x-1, y+1)
  float4 y2 = pos[position.y + position.z * XMAX]
            - pos[position.x + position.w * XMAX];
```

```
  float4 N0 = cross(x1, y1);
  float4 N1 = cross(x2, y2);

  norm[tid] = (float4)(-normalize(N0+N1).xyz, 0.0f);
}
```

Listing 5.7: CalculateNormals

The terrain normal generation is directly performed off the position mesh which is a combination of terrain- and water-height over an equally spaced grid. The normals are computed as cross product of a vector in x-direction (from the left to the right neighbour) and a vector in y-direction (from the top to the bottom neighbour) and additionally the same in the diagonals. The computed positions are clamped against zero and maximum indizes of the grid to ensure that no invalid values are introduced.

5.2.4 Water outflow calculation

```
__Kernel void CalculateOutflowFlux( __global float * terrainHeight
                                  , __global float * waterHeight
                                  , __global float8 * flux
                                  , float dt, float outflowDampening
                                  , float lcell, float lpipe )
{
  const uint tid = X + Y * XMAX;

  // my terrain height
  float b = terrainHeight[tid];
  // my water height
  float d1 = waterHeight[tid];

  // ----- update outflow flux
  // get the height of all neighbour cells, set my height as default
     if there is no cell (e.g. border)
  // get the intermediate water height (d1) of all neighbour cells

#if defined WATER_SIM_L
  float4 BN = get_neumann(b, terrainHeight);
  float4 D1 = get_neumann(d1, waterHeight);

#elif defined WATER_SIM_H
  float8 BN = get_moore(b, terrainHeight);
  float8 D1 = get_moore(d1, waterHeight);

#endif

  // calc cross sectional area of pipe
  const float Apipe = 0.0f, d1 * lcell;

  // compute outflow flux
#if defined WATER_SIM_L
  const float kFlow = dt * Apipe * g / lpipe;
```

```
    const float4 outFlux = fmax( (float4)(0.0f)
        , flux[tid].s0123 + kFlow * ((float4)(b + d1) - BN - D1) );

    const float sumFluxDt = getSum4(outFlux) * dt;

  #elif defined WATER_SIM_H
    const float8 kFlow = dt * Apipe * g
        / (float8)((float4) lpipe, (float4) lpipe * M_SQRT2);

    const float8 outFlux = fmax( (float8)(0.0f)
        , flux[tid] + kFlow * ((float8)(b + d1) - BN - D1) );

    const float sumFluxDt = getSum8(outFlux) * dt;

  #endif

    // max scale is less than 1.0f e.g. 0.99f -> to surpress oscillation
    // if there is no water (d1=0.0f) => Kscale will drop to 0.0f
    const float kScale = clamp( d1*lcell*lcell/sumFluxDt, 0.0f,
       outflowDampening );

    // write output value back to flux array
  #if defined WATER_SIM_L
    flux[tid].s0123 = kScale * outFlux;

  #elif defined WATER_SIM_H
    flux[tid] = kScale * outFlux;

  #endif
  }
```

Listing 5.8: CalculateOutflowFlux

First the current cell's terrain and water height are obtained. Then the terrain and water height of the neighbours are loaded into float4/float8 variables. If the current cell is at a border position, the supplied first parameter is returned by the *get_moore* or *get_neumann* functions in the according vector component. As next step the cross sectional area of the pipe is computed. Based on this area, the heights and water heights and the gravity constant g the different execution paths of the Kernel compute either a float4 or float8 version of the outFlux variable. The final $sumFluxDt$ value is then calculated via a special function which uses the possibility of OpenCL C to add a vectors low and high component - for an example see the getSum2 function in listing 5.9

```
  inline float getSum2( float2 sumvalues )
  {
    return sumvalues.lo + sumvalues.hi;
  }
```

Listing 5.9: getSum2 reduction function

As last step the *outFlux* variable is scaled down by the *kScale* factor which uses the original formula of Mei et al. [23]. The *outflowDampening* parameter was introduced to softly relax the

flow rate over time. This also has a slightly stabilizing effect on the water simulation, since big *outFlux* values fade out because of the dampening.

5.2.5 Water height update / Velocity Field calculation / Sediment Capacity computation

```
__Kernel void CalculateVelocity( __global float * waterHeight
                               , __global float8 * flux
                               , __global float2 * velocity
                               , __global float * sedimentCapacity
                               , __global float4 * norm
                               , float dt, float Kc
                               , float Kdmax, float lcell
                               , float minErosionAngle, float
                                  maxSpeedScale )
{
  const uint tid = X + Y * XMAX;

  // ----- update water height

  float d1 = waterHeight[tid];

#if defined WATER_SIM_L
  float4 fOut = flux[tid].s0123;
  float4 fIn = get_input_neumann(flux);

  const float dV = dt * ( getSum4(fIn) - getSum4(fOut) );

#elif defined WATER_SIM_H
  float8 fOut = flux[tid];
  float8 fIn = get_input_moore(flux);

  const float dV = dt * ( getSum8(fIn) - getSum8(fOut) );

#endif

  // intermediate water height d2
  const float d2 = d1 + dV/(lcell*lcell);

  // ----- velocity update

  const float dAvg = M_ONEHALF * (d1 + d2);

  // restrict inverse average to avoid to large numbers for velocity
  const float invAvgL = clamp( 1.0f / dAvg*lcell, 0.0f, maxSpeedScale
     );

#if defined WATER_SIM_L
  // dWx = fR(x-1,y) - fL(x,y) + fR(x,y) - fL(x+1,y)
  // dWy = fB(x,y-1) - fT(x,y) + fB(x,y) - fT(x,y+1)
  float2 vel = M_ONEHALF * (float2)(
      fIn.y - fOut.x + fOut.y - fIn.x,
```

```
        fIn.w - fOut.z + fOut.w - fIn.z ) * invAvgL;

#elif defined WATER_SIM_H
  // x component from diagonal neighbours
  float tmp = (fIn.s6-fOut.s4+fOut.s5-fIn.s7+fOut.s6-fIn.s4+fIn.s5-
      fOut.s7);
  // horizontal
  float dWx = M_ONEHALF * (fIn.s1 - fOut.s0 + fOut.s1 - fIn.s0 + tmp)
      * invAvgL;

  // y component from diagonal neighbours
  tmp = (fIn.s7-fOut.s5+fOut.s6-fIn.s4+fOut.s7-fIn.s5-fOut.s4+fIn.s6);
  // vertical
  float dWy = M_ONEHALF * (fIn.s3 - fOut.s2 + fOut.s3 - fIn.s2 + tmp)
      * invAvgL;

  float2 vel = (float2)(dWx, dWy);

#endif

  // ----- calc sediment capacity

  const float lVel = length(vel);

  // compute local tilt angle
  float alpha = ( alpha < minErosionAngle ) ? minErosionAngle : acos(
      dot( norm[tid], UP_VECTOR ) );

  // scaling based on the water depth
  const float lmax = (d2>=Kdmax) ? 1.0f : (1.0f - ((Kdmax-d2)/Kdmax));

  // calculate sediment transport capacity of the flow
  sedimentCapacity[tid] = Kc * sin(alpha) * lVel * lmax;

  // save velocity for next step
  velocity[tid] = vel;
  // save d2 for next step
  waterHeight[tid] = d2;
}
```

Listing 5.10: CalculateVelocity

The *CalculateVelocity* Kernel also performs multiple actions at once. First of all the water height is updated to $d2$ by using the difference of the inflows and the outflows of the current cell which where computed in the step before.

Then the average water height $dAvg$ between the water update steps $d1$ and $d2$ is obtained. With this value and the $cellSize$ the inverse of the cell area $invAvgL$ can be calculated.

Because the velocity is computed dividing the flow rate by the area and the area is defined as $dAvg * lcell$ the velocities can get very big for small water heights. This holds true especially at the fluids borders. A way to regulate this is to limit the $invAvgL$ by $maxSpeedScale$. Anyway this value is a key point in the simulation since the velocity is a major influencing factor in the sediment capacity calculation.

The individual components of the velocity vector are calculated by summing up x- respectively the y-parts of the current inflow and outflow.

In the next step the sediment capacity gets evaluated and the resulting values are saved back to the data arrays.

5.2.6 Erosion - Deposition

```
__Kernel void ErosionDeposition( __global float * terrainHeight
                               , __global float * waterHeight
                               , __global float * suspendedSediment
                               , __global float * sedimentCapacity
                               , __global float * accumulatedSoil
                               , __global float * hardness
                               , float dt, float Ks
                               , float Kd, float Kh
                               , float Rmin, float Rrestore
                               , float addSoilToWater )
{
  const uint tid = X + Y * XMAX;

  const float b = terrainHeight[tid];
  const float d3 = waterHeight[tid];
  const float st = suspendedSediment[tid];
  const float C = sedimentCapacity[tid];
  const float a = accumulatedSoil[tid];
  float h = hardness[tid];

  // calculate deposited / dissolved amount
  const float dSoil = ( C > st)? (- Ks * h * (C - st)) : (Kd * (st - C
      ));

  // update hardness coefficient
  h = fmax( Rmin, h - Kh * fmax( 0.0f, dSoil ) );
  // if water height is less than a certain amount
  // restore hardness by Rrestore
  hardness[tid] = (d3 < 0.0001f) ? (fmin(1.0f, h + Rrestore * dt)) : (
      h) ;

  terrainHeight[tid] = b + dSoil;
  waterHeight[tid] = fmax( 0.0f, d3 - addSoilToWater * dSoil );
  suspendedSediment[tid] = fmax( 0.0f, st - dSoil );

  accumulatedSoil[tid] = a + dSoil;
}
```

Listing 5.11: ErosionDeposition

The *ErosionDeposition* Kernel handles the calculation of the dissolved or deposited amount *dSoil*. Also the hardness coefficient is updated in this Kernel. If no or very little water is in the current cell the hardness gets slowly restored with a scale factor of *Rrestore*. Another important part is the limitation of the *waterHeight* and the *suspendedSediment* amount to zero since negative values do not make any sense for water or suspended sediment.

5.2.7 Sediment Advection

The sediment advection has multiple execution paths. If *None* is selected no Kernel gets executed. For the mode *BackwardEuler* the single BackwardEuler Kernel is scheduled. In case of mode *MacCormack* the three Kernels

- AdvectSedimentForward
- AdvectSedimentBackward
- LimitAdvection

are enqueued.

BackwardEuler

```
__Kernel void EulerAdvection( __global float * s1
                            , __global float * stpt
                            , __global float2 * velocity
                            , float dt
                            , float lcell )
{
  const uint tid = X + Y * XMAX;

  const float2 targetPos = (float2)(X, Y)
        - velocity[tid] * dt / lcell;

  stpt[tid] = bilerp(targetPos, s1);
}
```

Listing 5.12: EulerAdvection

The *EulerAdvection* Kernel simply calculates a new grid position from the current position and the given velocity at this point. To obtain the new value for the suspended sediment, a bilinear interpolation between the nearest neighbours is performed.

MacCormack Advection

```
__Kernel void AdvectSedimentForward( __global float * s1
          // s n
                                     , __global float * sedimentInp1
                                               // s^ n+1
                                     , __global float2 * sedimentClamp
                                     , __global float2 * velocity
                                     , float dt , float lcell )
{
  const int2 pos = (int2)(X, Y);
  const uint tid = X + Y * XMAX;

  const float2 targetPos = (float2)(X, Y)
        - velocity[tid] * dt / lcell;
```

```
    const float2 ab = clamp( targetPos - floor(targetPos), (float2)(0.0f
        ), (float2)(1.0f) );

    float4 TMP = get_nearest_neighbours(targetPos, s1);

    sedimentInp1[tid] = (1.0f-ab.x) * (1.0f-ab.y) * TMP.x
                        + ab.x * (1.0f-ab.y) * TMP.y
                        + (1.0f-ab.x) * ab.y * TMP.z
                        + ab.x * ab.y * TMP.w ;

    sedimentClamp[tid] = (float2)(  getMin4(TMP), getMax4(TMP) );
  }
```

Listing 5.13: AdvectSedimentForward

The forward operator corresponds to the euler step backward in time. Although the operations are equal to the *EulerAdvection* Kernel the *AdvectSedimentForward* Kernel does not use the bilerp function. Instead the sediment values of the nearest neighbours are used to determine a minimum/maximum value for the limiter stage of the MacCormack advection. This values are saved in the *sedimentClamp* buffer.

```
  __Kernel void AdvectSedimentBackward( __global float * sedimentInp1
                                      , __global float * errorEstimate
                                      , __global float2 * velocity
                                      , float dt, float lcell )
  {
    const int2 pos = (int2)(X, Y);
    const uint tid = X + Y * XMAX;

    // save current value of errorEstimate -> its sn
    float sn = errorEstimate[tid];

    // step forward in time
    const float2 targetPos = convert_float2(pos)
          + velocity[tid] * dt / lcell;

    // s^ n
    const float sInp = bilerp(targetPos, sedimentInp1);

    // save the error estimate: sediment - sedimentN
    errorEstimate[tid] = sn - sInp;
  }
```

Listing 5.14: AdvectSedimentBackward

The Kernel *AdvectSedimentBackward* is again basically identical to *EulerAdvection*. The difference lies in the calculation of *targetPos* - instead of going backward we take a step forward in time. With the new computed value the error estimate can be calculated and saved for the next stage.

```
__Kernel void LimitAdvection( __global float * errorEstimate
                            , __global float * sedimentNp1
                            , __global float2 * sedimentClamp )
{
  const uint tid = X + Y * XMAX;

  const float errorEst = errorEstimate[tid];    // sn - s^ n
  const float sInp1 = sedimentNp1[tid];     // s^ n+1

  const float2 clampValues = sedimentClamp[tid];

  // set calculcated amount of suspended sediment in cell
  // limited by 1st order values
  sedimentNp1[tid] = clamp( sInp1 + M_ONEHALF*errorEst
                          , clampValues.x, clampValues.y );
}
```

Listing 5.15: LimitAdvection

LimitAdvection computes the final resulting sediment amount for the current cell. As a final step the 2nd order accuracy value is limited by 1st order accuracy min/max values computed from the original implicit euler advected field.

5.2.8 Thermal Weathering Soil Outflow Calculation

```
__Kernel void CalculateWeathering( __global float * terrainHeight
                                 , __global float8 * soilFlow
                                 , __global float4 * normals
                                 , __global float * hardness
                                 , __global float * soilWetness
                                 , float dt  , float lcell
                                 , float talusAngle, float Ka
                                 , float Kt, float cellScale
                                 , float wetnessScale )
{
  const uint tid = X + Y * XMAX;

  const float currentHeight = terrainHeight[tid];
  const float h = hardness[tid];

  // get the terrain height of all neighbours
  const float8 neighbourHeights = get_moore(currentHeight,
     terrainHeight);

  // currentHeigth - neighbourHeight is positive if neighbours heights
      are less then the current height
  // max returns 0.0f is neighbours height is bigger than
     currentHeight
  const float8 heightDiff = fmax( (float8) 0.0f, (float8)currentHeight
     -neighbourHeights );
```

```
    const float8 alpha = atan( heightDiff / (cellScale*(float8)((float4)
        lcell, (float4)(lcell*M_SQRT2))) );
    const float materialTalus = h * Ka + talusAngle + ((soilWetness[tid
        ]/30.0f)-20.0f);
    const int8 alphaGreaterThanTalus = isgreater( alpha*M_TO_DEGREE,
        materialTalus );

    // mask out the other heights which are not greater than the talus
        angle
    const float8 heightsUnderTalus = select(
              (float8)0.0f,
              (float8)1.0f,
              alphaGreaterThanTalus ) * heightDiff;

    // calculate the maximum volume to be moved
    const float dV = dt*Kt*h*lcell*lcell*M_ONEHALF*getMax8(heightDiff);

    // calc sum of heights
    const float sum = getSum8(heightsUnderTalus);

    // update soilFlow
    soilFlow[tid] = (sum > 0.0f) ? ( dV *  heightsUnderTalus / sum)
        : ((float8)0.0f);
}
```

Listing 5.16: CalculateWeathering

CalculateWeathering uses all eight neighbours of the current cell. At first the height differences between the current cell and the neighbours are calculated. By limiting *heightDiff* to 0.0f the neighbours which are higher than the current cell are masked out.
After this the local talus angle to the neighbour is computed. For the diagonal neighbours the adjusted cell size $l_{cell} * \sqrt{2}$ is used. Since the visual spacing of the grid gets scaled at load time, the cell size needs to be scaled here too or the slopes would look wrong in the visualization.
In the next step the computed terrain angle is then compared to the material's talus angle $materialTalus$. The talus angle is mainly influenced by the material constant as well as the local hardness coefficient. If the coeficient is low the angle is close to the material constant, otherwise the material angle gets higher which means that it withstands bigger angles.
As another addition *soilWetness* was added to the *materialTalus* computation. If a material e.g. gets in solution it cannot be piled up very high, on the other side if the material is only slightly wet it withstands bigger angles. Since *soilWetness* mimics this behavior by mapping zero or very big water values to a low percent value and slight wetness to a big percent value, it can be used to adjust the materialAngle accordingly.
By using the int8 mask *alphaGreaterThanTalus* the neighbours which should not get any material are masked out in the variable *heightsUnderTalus*. The maximum volume *dV* going to be moved is then calculated as shown in chapter 3.

5.2.9 Thermal Weathering Height Update

```
__Kernel void WeatheringTerrainHeightUpdate(
```

```
            __global float * terrainHeight
          , __global float8 * soilFlow )
{
  const uint tid = X + Y * XMAX;

  float currentHeight = terrainHeight[tid];

  float8 outgoingSoil = soilFlow[tid];
  float8 incomingSoil = get_input_moore(soilFlow);

  // new height = current height + incoming - outgoing
  terrainHeight[tid] = currentHeight + getSum8(incomingSoil) - getSum8
      (outgoingSoil);
}
```

Listing 5.17: WeatheringTerrainHeightUpdate

The Kernel *WeatheringTerrainHeightUpdate* simply cumulates the soilFlow for the current cell and adds it to the cell's height.

5.3 Visualization

Multiple shaders where written to visualize certain parts of the simulaton

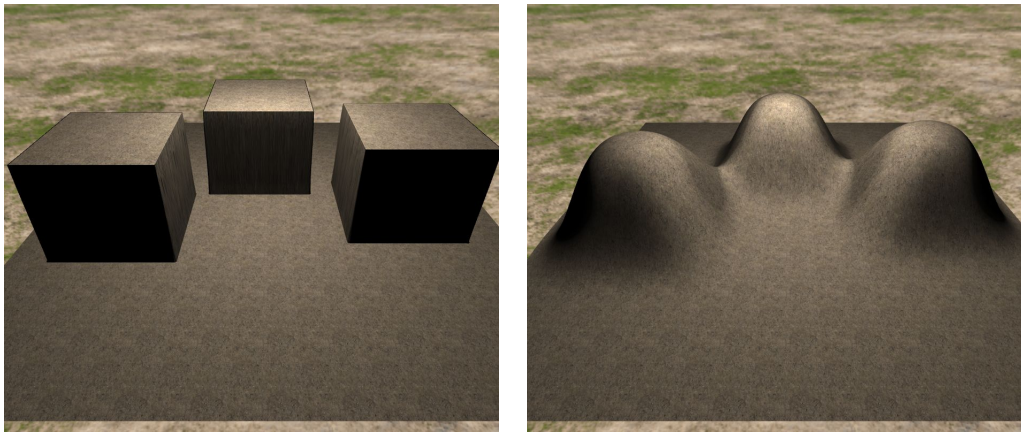
- SceneShader - phong shading and lighting and water reflection/refraction shading
- DebugShader - visualizes the calculated velocities and normals as color coded vectors in the scene
- ParamVisualizationShader - visualizes various simulation parameters by color coding using an orthogonal projection (see figure 5.4)
- SkyboxShader - draws a box from the cubemap textures
- ColorShader - uses only the material color as well as the distance to the light source

The scene rendering directly uses the updated values of the *TerrainPosition* buffer for the vertex positions. Multiple texture layers are added during the fragment shader pass of the *SceneShader* depending on the local accumulated soil and on the material wetness buffer as well as on the local water height. The used lighting model is the phong model. The reflection on the water uses a simple cubemap lookup, whereas the refraction is calculated by offsetting the texture coordinates according to the local normal and a fixed scale. Then the two values are mixed together using the fresnel term.

6 Results

This chapter presents the results achieved with the different variants of the erosion algorithm. At first the thermal weathering stage is tested in section 6.1. The next section 6.2 compares the different hydraulic erosion variants implemented, whereas in the following section 6.3 the results with the combined algorithm are presented, whereupon subsection 6.3.1 compares some natural phenomena to results which where achieved with the algorithm. At last the performance of the implementation is discussed in section 6.4.

6.1 Thermal Weathering



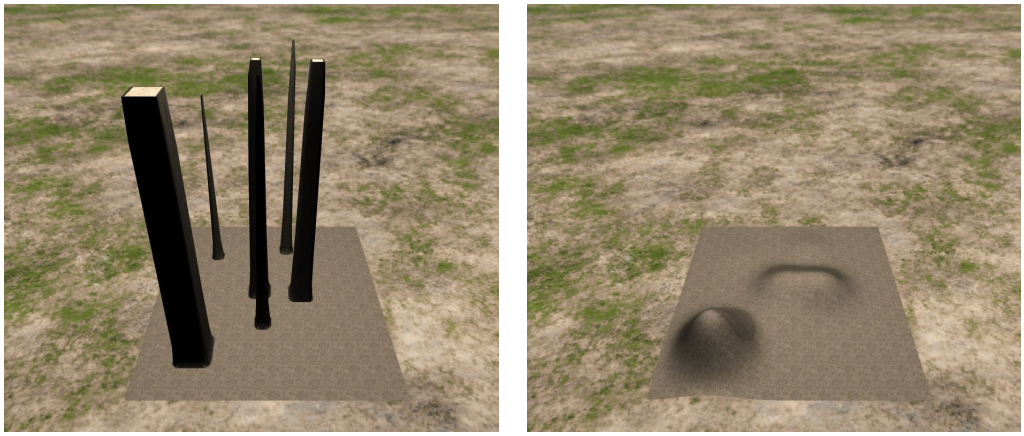
(a) Test Scene 1 (b) Test Scene 1 after 1000 iterations

Figure 6.1: Thermal Weathering Test 1

The results from the thermal erosion stage can be seen in figure 6.1 and figure 6.2. For both simulations an unusually high value for the thermal weathering constant K_t (100.0 & 50.0) was chosen to see the effects more clearly. As expected thermal weathering smoothes the edges of the terrain.

6.2 Hydraulic Erosion Neighbourhood and Advection

In figure 6.3 the erosion of a "PG" shaped mountain is shown. For this picture the water simulation was set to *FourPipes* with *BackwardEuler* advection. The parameters where set to mimic the behavior of the original testcase from Mei et. al [23]. The first picture depcits the initial terrain. The picture at the right shows the simulation a few iterations after turning on the random rain. In the lower left picture some deposition can already be seen. The picture at the right bottom



(a) Test Scene 2 (b) Test Scene 2 after 1000 iterations

Figure 6.2: Thermal Weathering Test 2

was taken after all the water has been evaporated. The cyan color indicates suspended sediment in the water, while red stands for deposited sediment.

Test Setup

To test the different implemented neighbourhoods as well as the different advection stages, two different scenes were simulated with random rain. All tests for a scene share the same test parameters, only the pipe model and the advection was changed. For each scene four test runs with the following settings where made:

- Neighbourhood: 4 Pipes - Advection: Euler
- Neighbourhood: 4 Pipes - Advection: Mac Cormack
- Neighbourhood: 8 Pipes - Advection: Euler
- Neighbourhood: 8 Pipes - Advection: Mac Cormack

In addition to the scene pictures the sediment images where also added for comparison. The results can be seen in figures 6.4, 6.6, 6.7 and 6.8.

The used algorithm parameters can be seen in table 6.1 (parameters which are not used by the tested erosion model are omitted).

Test Scene "PG"

The test scene "PG" uses a similar heightmap to the one Mei et al. used in their erosion simulation test. Figure 6.4 and 6.6 show the outcome of the simulation. Although the results look similar at first, the differences are in the details. Comparing figure 6.5 (a) to 6.5 (b) we can see, that (b) exposes more structure at the sedimentation sites, where (a) looks just flat - the same observation applies to (c) and respectively (d). When looking at the eroded ridges, we can see that in (a) and (b) the lines are rather straight, whereas in (c) and (d) they also take more diagonal paths. Comparing the suspended sediment maps it can be clearly seen, that the MacCormack scheme

Figure 6.3: “PG” shaped mountain example

produces much more detail and structure than the simple Euler Advection, which looks like a blurred version of the MacCormack scheme.

Test Scene “Pyramid”

The conclusions from the test scene “PG” also apply to the “Pyramid” test scene. Again it can be observed that straight lines towards the bottom begin to emerge with a four pipe neighbourhood, while an eight pipe neighbourhood allows more diagonal flow and therefore looks more natural. The comparison of the advection schemes also yield the same results as in test scene “PG”. The results can be seen in figure 6.7 and 6.8.

6.3 Full Algorithm Results

In figure 6.9 the combined results of the hydraulic erosion and the thermal weathering can be observed on a larger grid (1024x1024). While the hydraulic erosion stage alone produces deep ridges, the thermal erosion smoothes out these effects. However the whole simulation is still very sensitive to the chosen parameters and has to be adapted for the respective situation.

Parameter	Test Scene "PG"	Test Scene "Pyramid"
TimeStep	0.01	0.01
MinErosionAngle	10.0	10.0
MaxSpeedScale	50.0	max.
Kc	1.0	1.0
Ks	0.01	0.0001
Kd	0.001	0.1
Kdmax	4.0	1.0
Kh	0.01	0.01
Khmin	0.1	0.1
Khrestore	0.01	0.01
CellSize	1.0	1.0
PipeLength	1.0	1.0
OutflowDampening	0.995	0.995
AddSoilToWater	true	true
CorrectAdvection	true	true
RainIntensity	10.0	10.0
RainFrequency	2	2
RaindropSize	2	2
EvaporationRate	0.1	0.1

Table 6.1: Algorithm Test Parameters

6.3.1 Comparison to Natural Phenomena

To prove that the algorithm produces results which can be also observed in nature, several real scenes were taken and simulated with the algorithm as close as possible. The results can be seen in figure 6.10 and figure 6.11.

6.4 Performance Measurements

Test Setup

- CPU: Intel Core i5 750 @ 2.66 GHz, 4 Compute Units
- GPU: AMD Radeon HD 6870 @ 900 MHz, 14 Compute Units, 1GB Memory
- Memory: 8GB DDR3 RAM
- OS: Windows 7, 64 Bit with SP1
- Maps: bds_128, bds_256, bds_512, bds_1024, bds_2048

The performance measurements of the individual Kernels where conducted with the AMD APP Profiler. For the overall time the *StopWatch* class was used. Each test took place on the same map - only the grid size was scaled down for the respective case. All values are averages

[1] http://en.wikipedia.org/wiki/File:Devils_Tower_CROP.jpg
[2] http://www.flickr.com/people/dolgin/

measured in milliseconds (*ms*) over 1000 simulation iterations.

For grid sizes 128, 256 and 512 the simulation time only approximately doubles for an exponential increase of cells. Starting with grid size 1024 this factor increases to three whereas at size 2048 it is at approximately 3.5. From eight pipes to four pipes the computation times where lower but this did not have as much impact on the computation time as expected. Apparently there are other bottlenecks which limit the simulation time to an upper bound (e.g. the Weathering Kernels use a Moore neighbourhood in every test case).

Figures & result tables can be found in 6.12, 6.13 as well as 6.14.

For the CPU measurements the Intel OpenCL SDK was used and the OpenCL Device was simply switched to type CPU. Only totals where measured to have comparative overall results to the GPU variant.

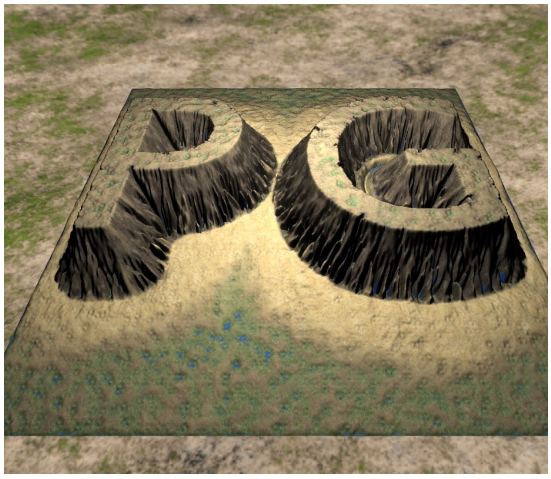
The achieved speedup to the compared "CPU only" measurement was between 5 and 23 times (see table 6.4). Considering Jákó [26] who reports a speedup of approximately 100 times compared to a "CPU only" solution these values seem very low. However the CPU solution measured by Jákó was not reported to be accelerated in any way whilst the CPU measurements (see figure 6.14) where parallelized and auto-vectorized automatically by using the Intel OpenCL SDK. Moreover simulation times measured with the AMD APP SDK and Device Type CPU took approximately twice the times achieved with the Intel SDK.

Grid Size	GPU	CPU	Speedup CPU/GPU
128x128	0.88	4.4	5
256x256	1.42	11	7,75
512x512	2.72	41	14,9
1024x1024	8.8	163	18.52
2048x2048	29.87	685	22.93

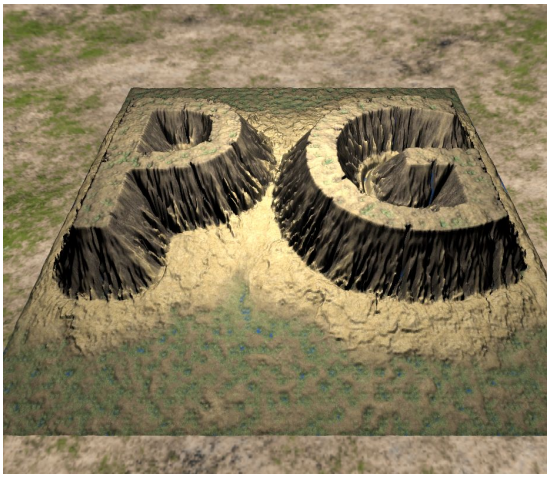
Table 6.2: Comparison CPU vs. GPU

(a) “PG”, 4 Pipes, Euler Advection

(b) “PG”, 4 Pipes, MacCormack Advection

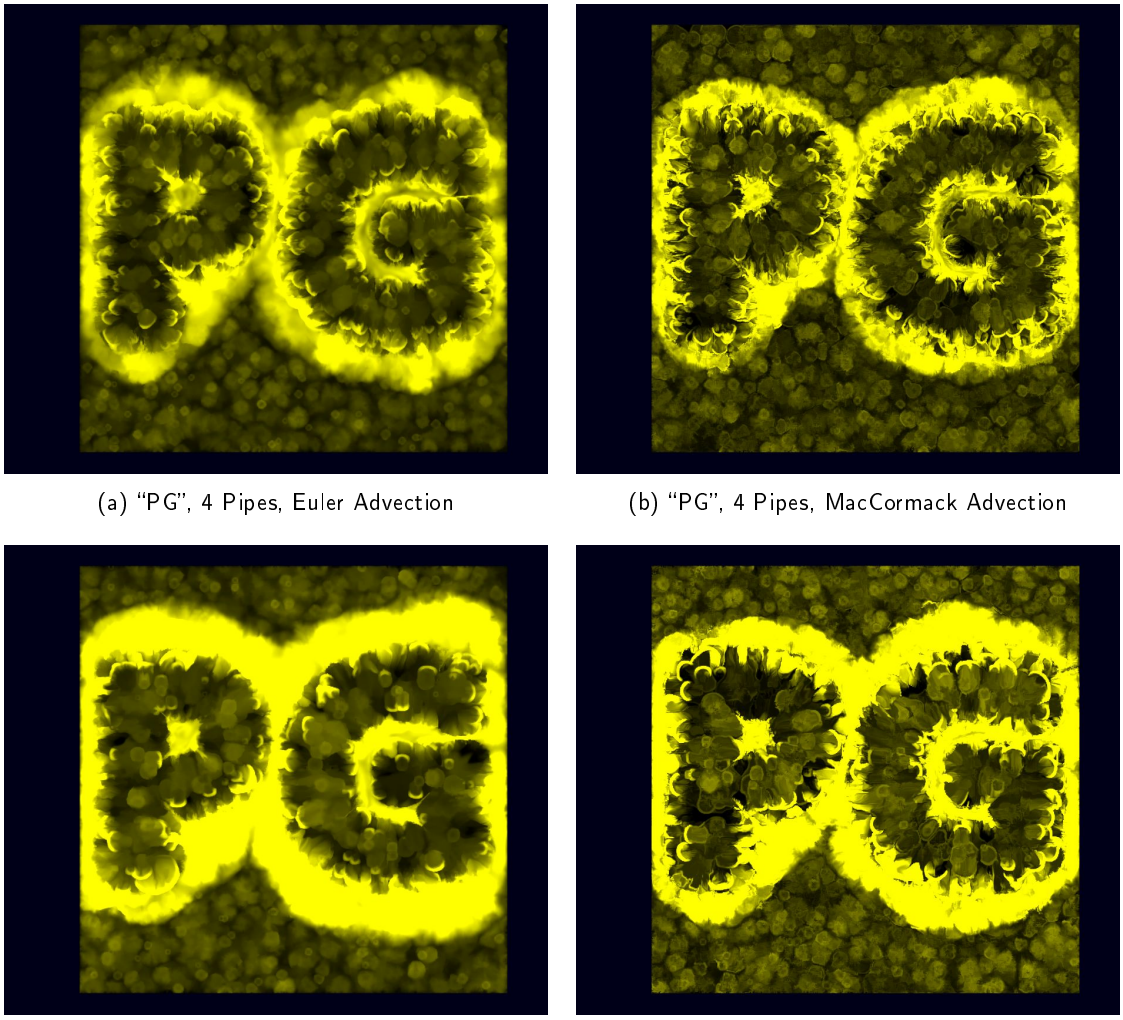


(c) “PG”, 8 Pipes, Euler Advection



(d) “PG”, 8 Pipes, MacCormack Advection

Figure 6.4: 4 vs. 8 Pipes - Euler vs. Mac Cormack Advection - “PG”



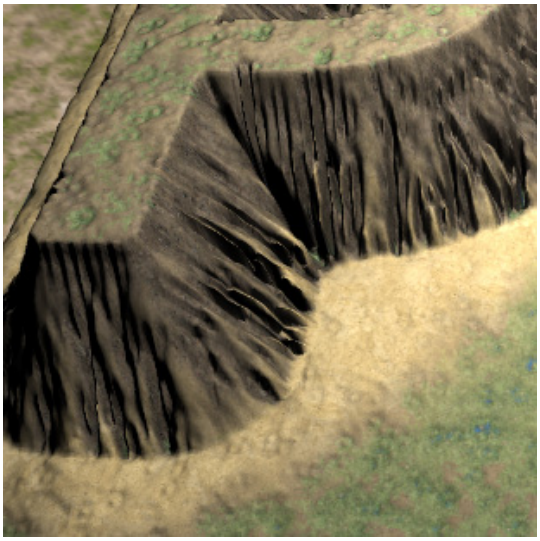
(a) “PG”, 4 Pipes, Euler Advection

(b) “PG”, 4 Pipes, MacCormack Advection

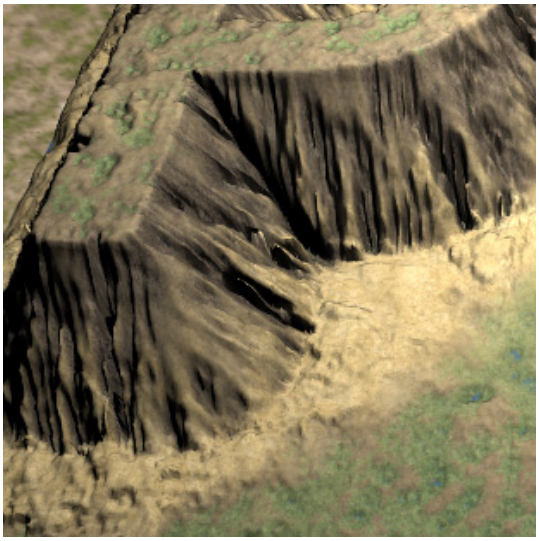
(c) “PG”, 8 Pipes, Euler Advection

(d) “PG”, 8 Pipes, MacCormack Advection

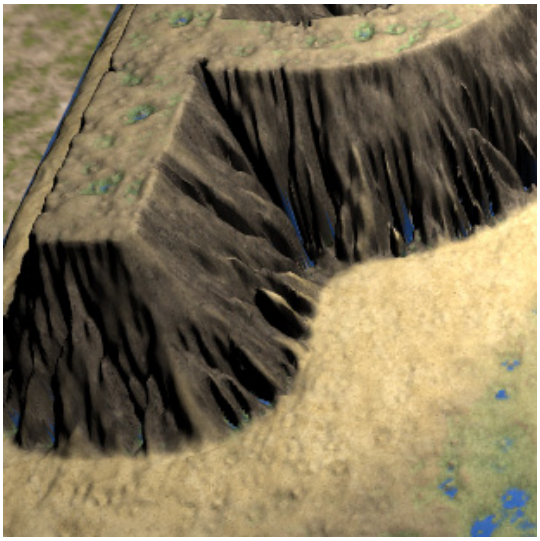
Figure 6.5: 4 vs. 8 Pipes - Euler vs. Mac Cormack Advection - “PG” Detail



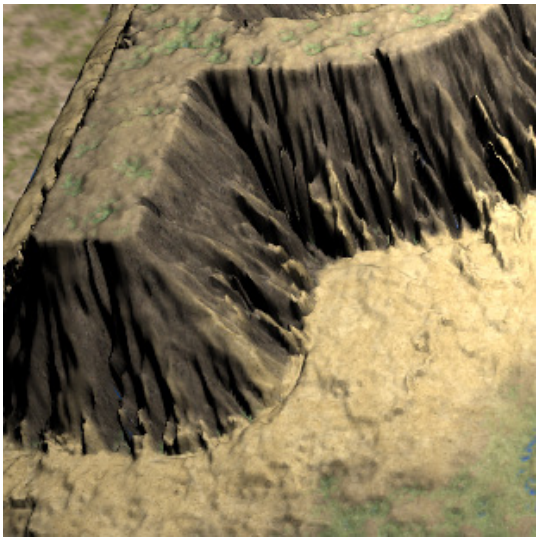
(a) “PG”, 4 Pipes, Euler Advection Detail



(b) “PG”, 4 Pipes, MacCormack Advection Detail

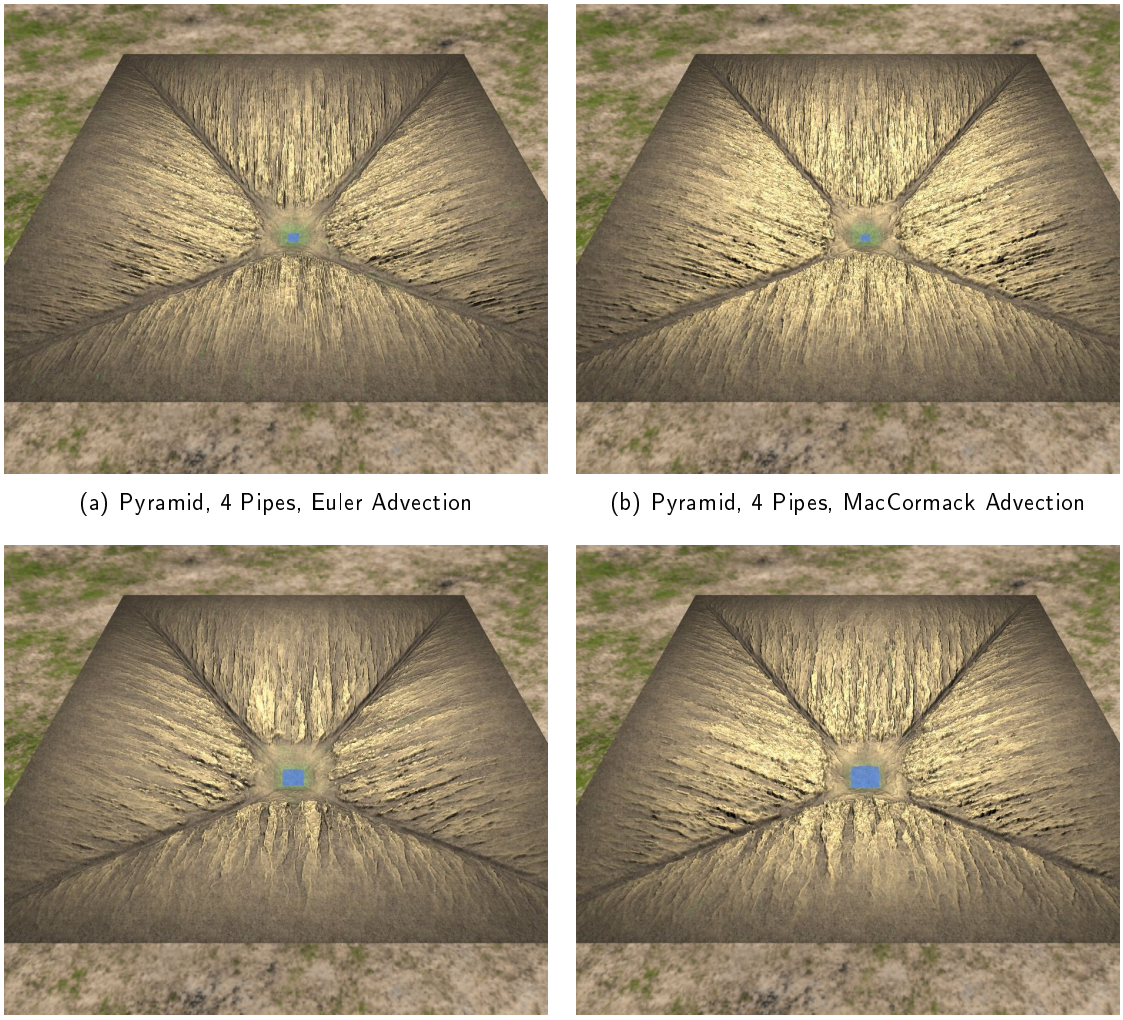


(c) “PG”, 8 Pipes, Euler Advection Detail



(d) “PG”, 8 Pipes, MacCormack Advection Detail

Figure 6.6: 4 vs. 8 Pipes - Euler vs. Mac Cormack Advection Sediment - “PG”



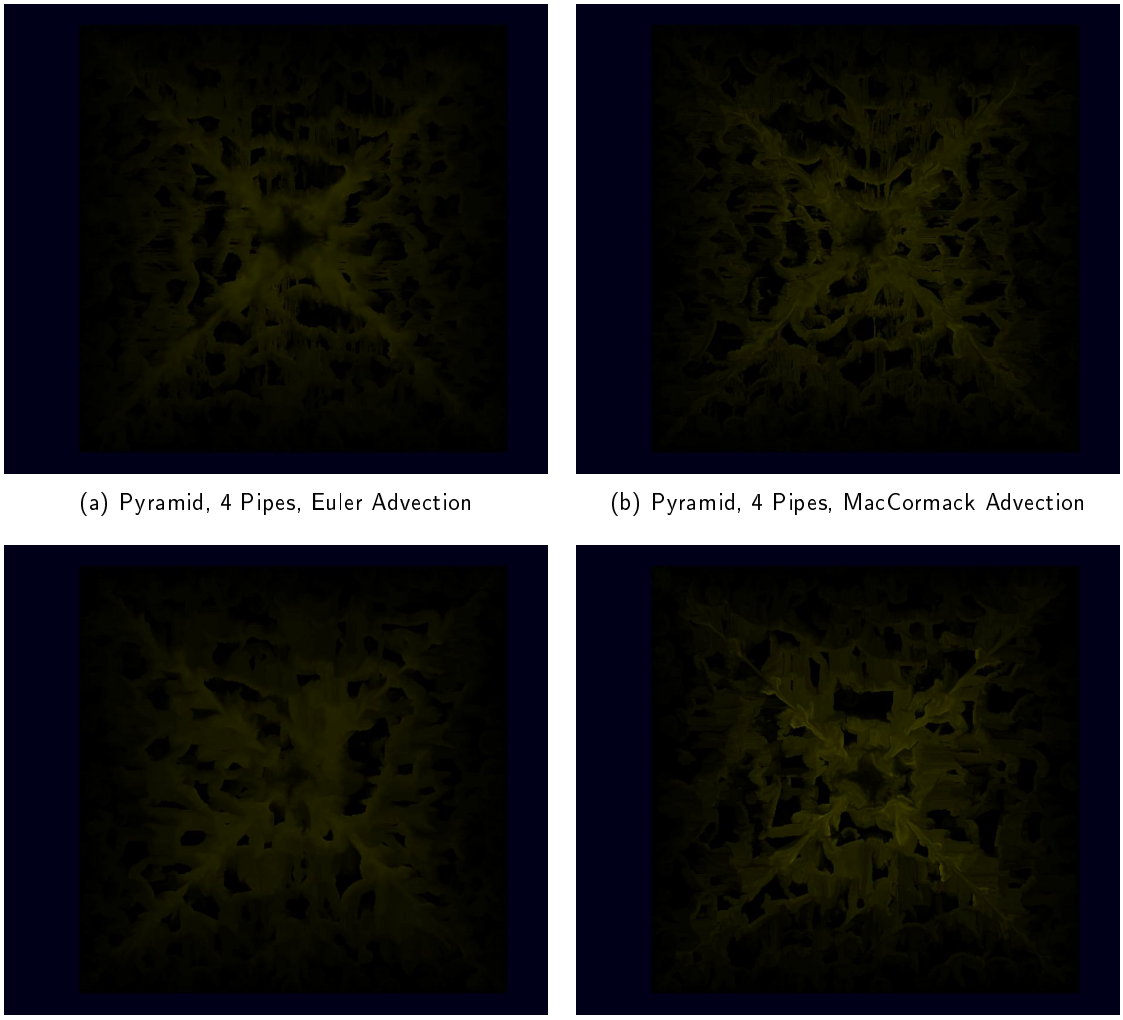
(a) Pyramid, 4 Pipes, Euler Advection

(b) Pyramid, 4 Pipes, MacCormack Advection

(c) Pyramid, 8 Pipes, Euler Advection

(d) Pyramid, 8 Pipes, MacCormack Advection

Figure 6.7: 4 vs. 8 Pipes - Euler vs. Mac Cormack Advection - Pyramid



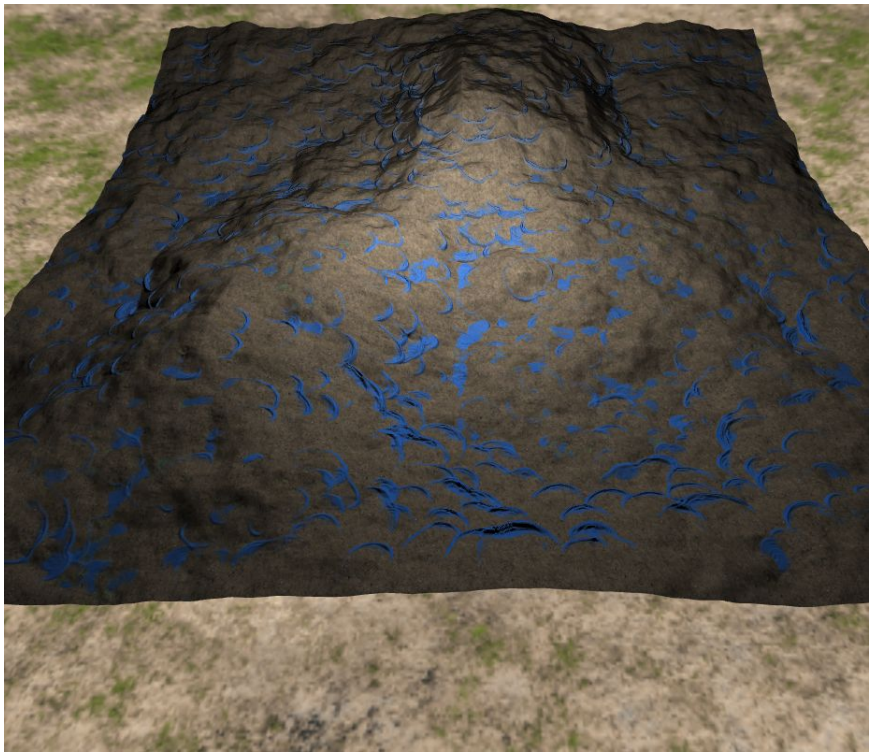
(a) Pyramid, 4 Pipes, Euler Advection

(b) Pyramid, 4 Pipes, MacCormack Advection

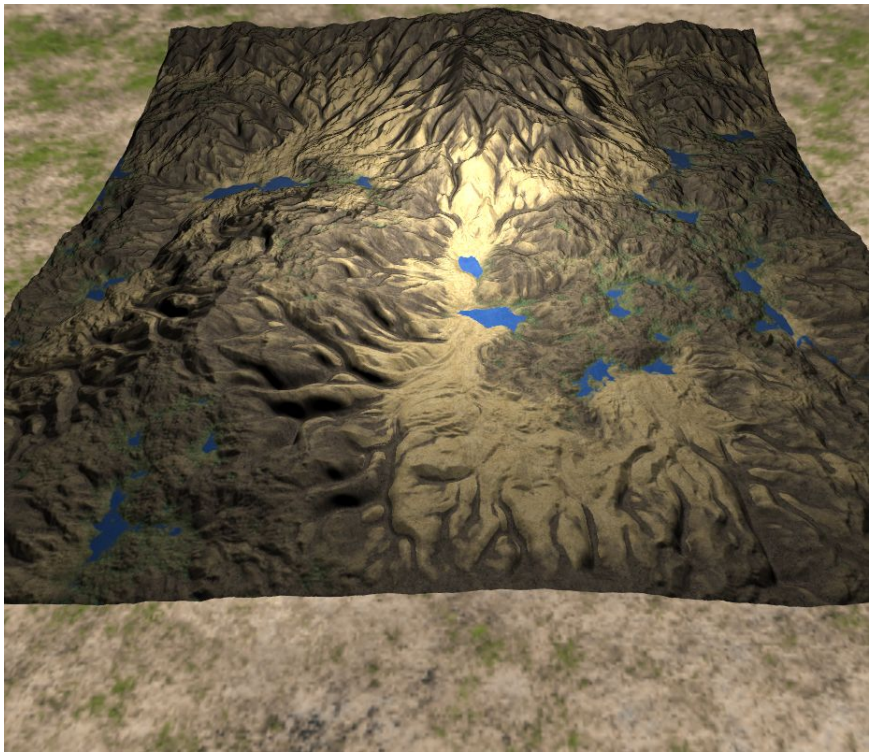
(c) Pyramid, 8 Pipes, Euler Advection

(d) Pyramid, 8 Pipes, MacCormack Advection

Figure 6.8: 4 vs. 8 Pipes - Euler vs. Mac Cormack Advection Sediment - Pyramid



(a) Initial



(b) After Erosion

Figure 6.9: Full Erosion Model Example

(a) Devils Tower

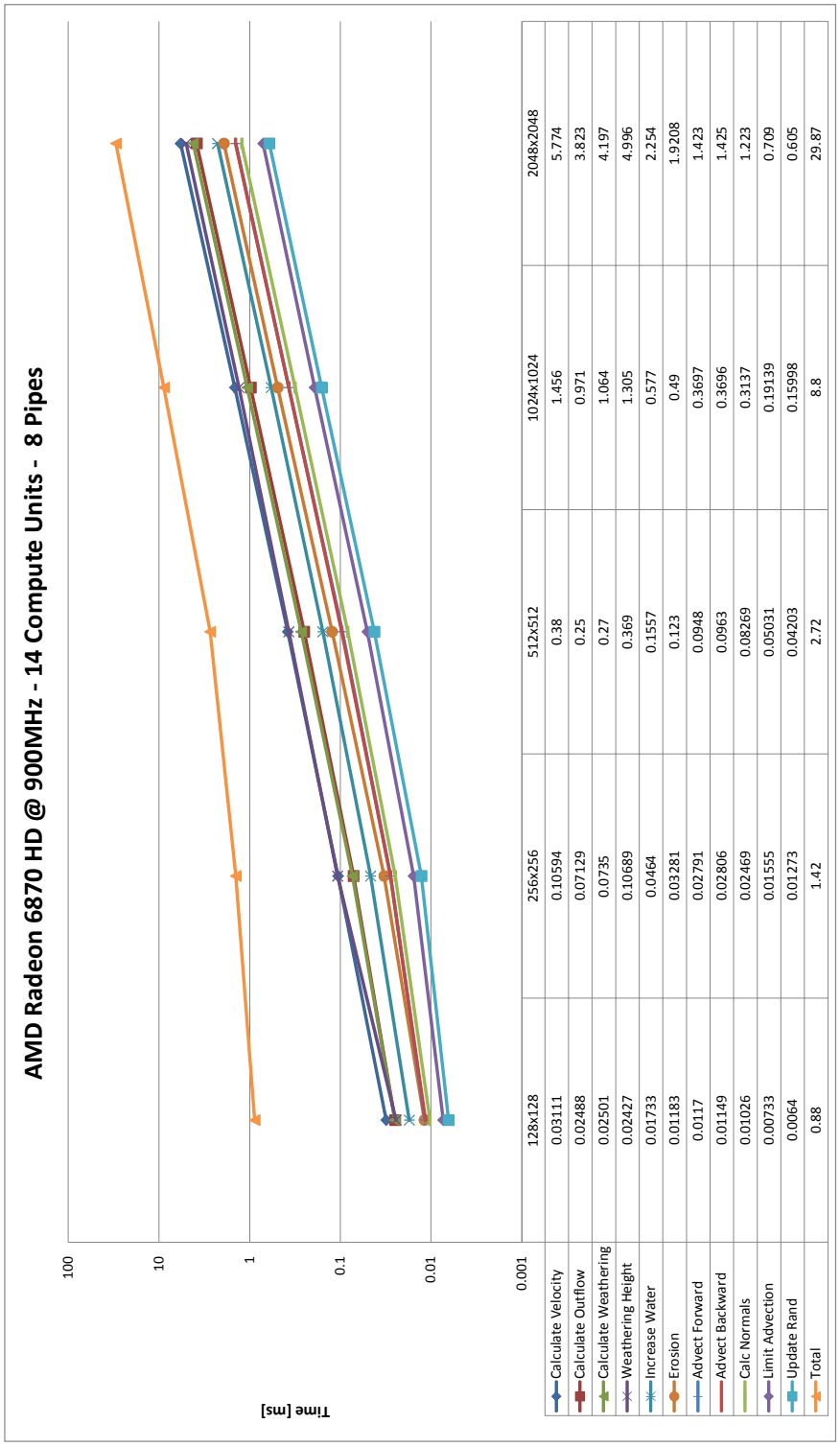
(b) Simulation

Figure 6.10: Devils Tower in Wyoming, (photographed by Colin Faulkingham [1])

(a) Grooves

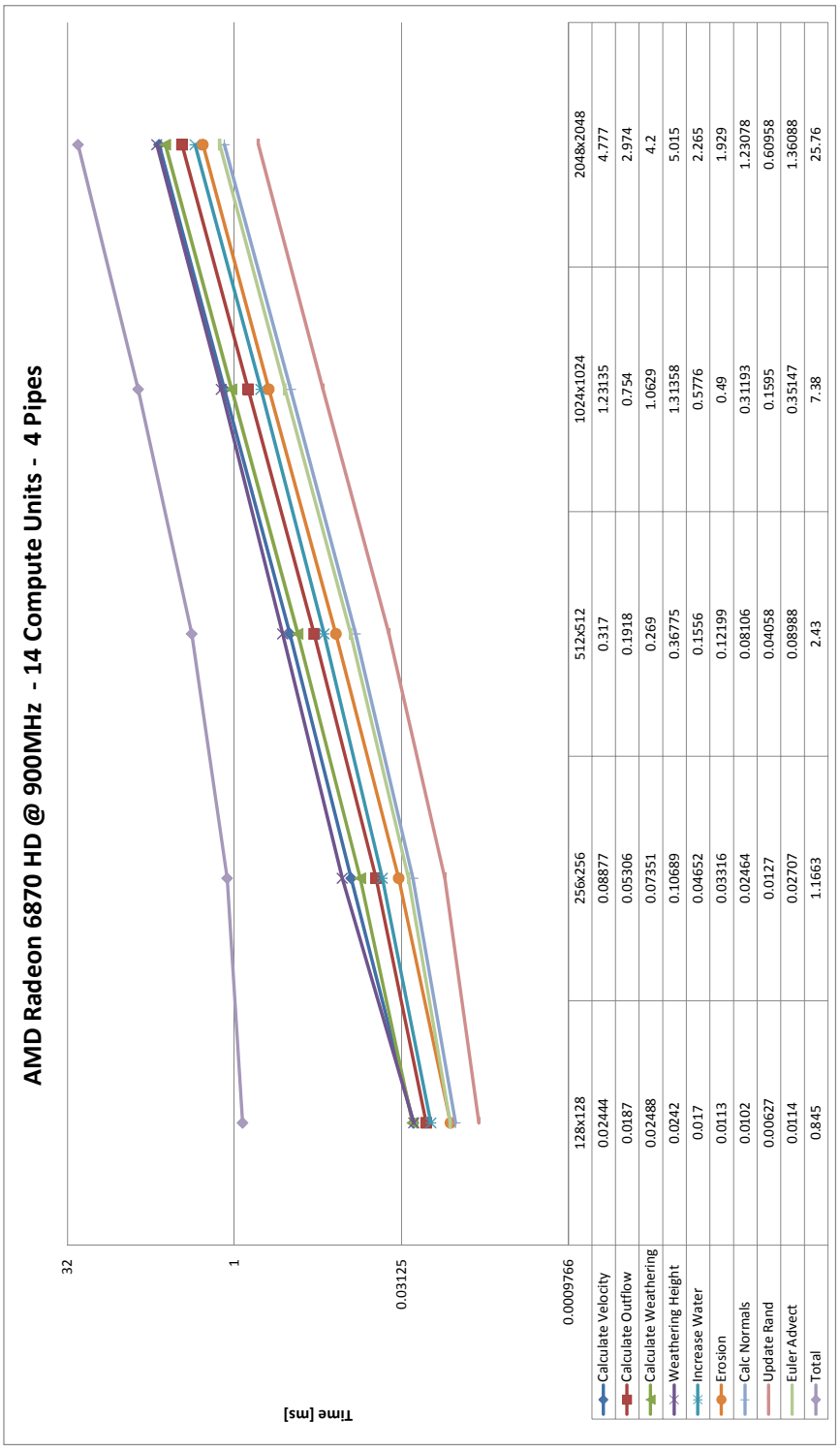
(b) Simulation

Figure 6.11: Grooves, (photographed by Avi Dolgin [2])



	128x128	256x256	512x512	1024x1024	2048x2048
Calculate Velocity	0.03111	0.10594	0.38	1.456	5.774
Calculate Outflow	0.02488	0.07129	0.25	0.971	3.823
Calculate Weathering	0.02501	0.0735	0.27	1.064	4.197
Weathering Height	0.02427	0.10689	0.369	1.305	4.996
Increase Water	0.01733	0.0464	0.1557	0.577	2.254
Erosion	0.01183	0.03281	0.123	0.49	1.9208
Advect Forward	0.0117	0.02791	0.0948	0.3697	1.423
Advect Backward	0.01149	0.02806	0.0963	0.3696	1.425
Calc Normals	0.01026	0.02469	0.08269	0.3137	1.223
Limit Advection	0.00733	0.01555	0.05031	0.19139	0.709
Update Rand	0.0064	0.01273	0.04203	0.15998	0.605
Total	0.88	1.42	2.72	8.8	29.87

Figure 6.12: Measurement 1, 8 Pipes & MacCormack Advection



	128x128	256x256	512x512	1024x1024	2048x2048
Calculate Velocity	0.02444	0.08877	0.317	1.23135	4.777
Calculate Outflow	0.0187	0.05306	0.1918	0.754	2.974
Calculate Weathering	0.02488	0.07351	0.269	1.0629	4.2
Weathering Height	0.0242	0.10689	0.36775	1.31358	5.015
Increase Water	0.017	0.04652	0.1556	0.5776	2.265
Erosion	0.0113	0.03316	0.12199	0.49	1.929
Calc Normals	0.0102	0.02464	0.08106	0.31193	1.23078
Update Rand	0.00627	0.0127	0.04058	0.1595	0.60958
Euler Advect	0.0114	0.02707	0.08988	0.35147	1.36088
Total	0.845	1.1663	2.43	7.38	25.76

Figure 6.13: Measurement 2, 4 Pipes & Euler Advection

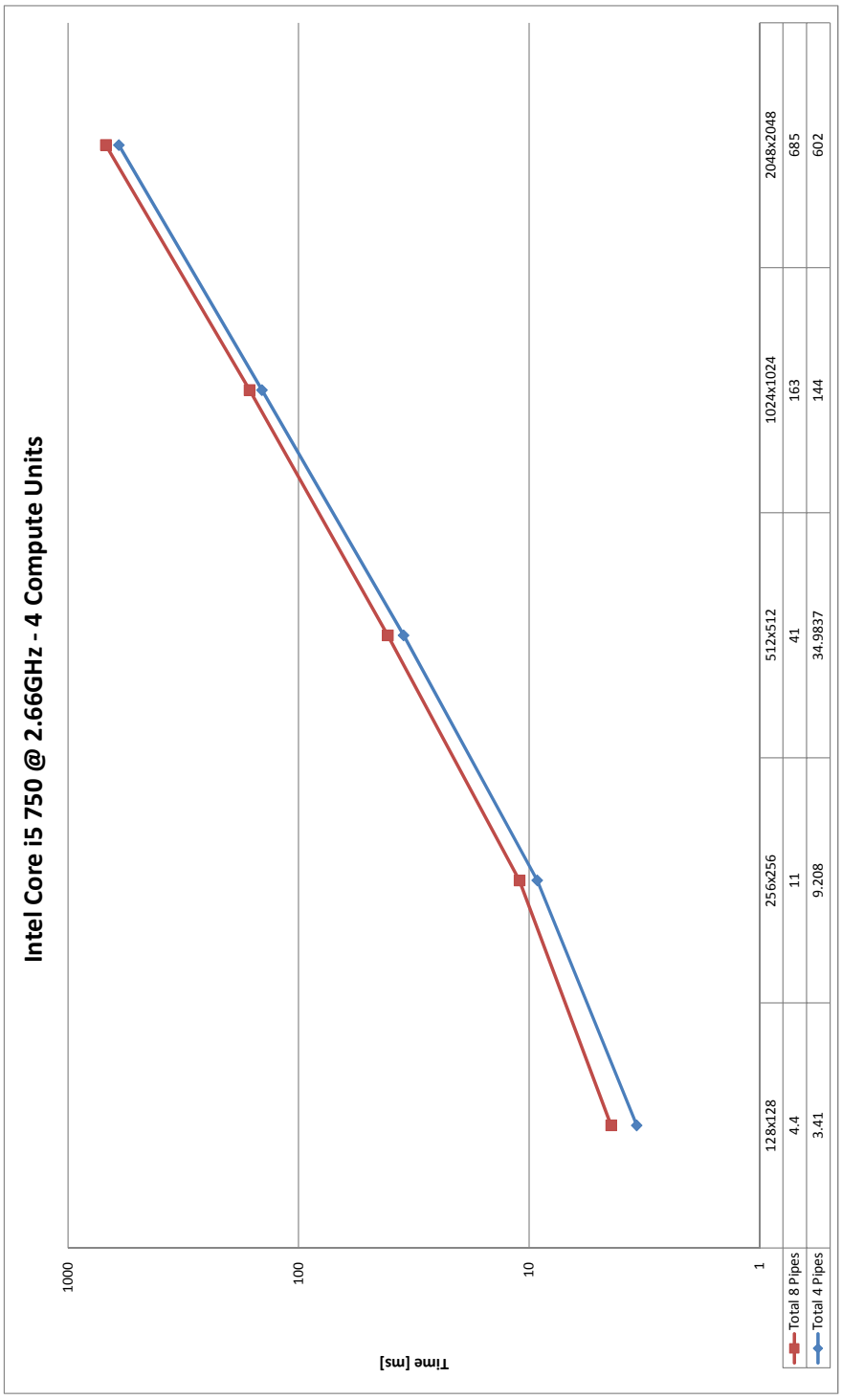


Figure 6.14: Measurement CPU, 4 Pipes & Euler Advection + 8 Pipes & MacCormack Advection

7 Conclusion

The initial purpose of this thesis was to implement an erosion algorithm with the new and universal OpenCL computing platform. For this the existing literature was examined leading to a series of papers which already explored the theme real-time erosion simulation but on different platforms and with a very specialized focus. With the knowledge of these algorithms a new mixture of the different strategies was proposed in chapter 3 “Erosion Model”.

To give an overview on the OpenCL platform which is new in comparison to already existing approaches like nVidia’s Cg Shading Language or CUDA, the very basics where explained in chapter 4 “A Brief Introduction to OpenCL”. References to current hardware were made regularly, to give a good insight on how the distinct components are implemented on actual hardware. At the end of this chapter a short example program was shown to give an idea on how to start an OpenCL application from scratch.

In the following chapter the implementation of the laid out model in a self-written test framework and an overview of the host application was presented, followed by an explanation of the memory layout and the execution path of the OpenCL parts. Later on the implemented Kernels’s were discussed at which more insight into the actual realization was provided.

Chapter 6 “Results” covered the different stages of the erosion algorithm which were explored in separation as well as combined. To prove the simulation results, some real world phenomena were taken and compared to simulated scenes. Finally the measured performance of the algorithm was presented on both CPU and GPU solutions, which was achieved by forcing the OpenCL runtime to run on a specific device type. Although the performance measurements from chapter “Results” were sufficient for this kind of flexible cover-it-all-approach, there is still much room for optimizations.

After all it can be concluded that hydraulic erosion alone can produce good resuts when the parameters are tuned right for the specific situation. However the achieved results and also the stability can be greatly improved in a long term manner if a thermal weathering process is added to the simulation. The problem with the hydraulic erosion alone is, that it produces pretty sharp edges on a regular grid and can carve very deep channels. The thermal weathering process is able to smooth out these effects if configured correctly.

The comparison of the four pipe model to an eight pipe model is a point of discussion with no clear answer. On the one hand it can be clearly stated that simulation results can be visually improved by using a “Moore” neighbourhood, on the other hand it can be less memory and computational intensive if using only a “Von Neumann” neighbourhood if optimized for this approach. The same applies to the advection step where the MacCormack scheme was compared to a simple-semi lagrangian backward Euler step. The MacCormack scheme produces much more detail and if such detail is desired a MacCormack scheme can greatly improve simulation results, whilst maintaining only a moderate overhead. Nevertheless this decision is problem dependant,

e.g. in case of a physics simulation for a game, it makes sense to abandon all overhead in favor of general execution speed.

The general advice is to include the thermal weathering step at any rate and if desireable the eight pipe solution and the MacCormack advection scheme.

At last this thesis suggests topics for further researches. For example the shallow water implementation has several disadvantages by nature and could be improved by using other approaches like multiple layers of pipes as described in the work "Efficient Animation of Water Flow on Irregular Terrains" by Maes et al. [33]. Also the hydraulic erosion stage could be improved by using multiple layers of material like Stava et al. [24]. For the sediment advection it is highly recommended to research possibilities on how to implement a mass conserving advection scheme in this algorithm or at least find a good solution to calculate deposition and dissolving constants automatically on a per cell basis. As mentioned before the application of this algorithm e.g. in games need further optimizations which also offers a direction for future work. A part which was completely out of focus of this work was the optimization of the visualization. There exist many proposals for efficiently rendering terrain geometry but the combination of these rendering techniques with the erosion algorithm needs further research.

Bibliography

[1] R. S. Harmon and W. W. D. III., *Landscape Erosion and Evolution Modeling*. Kluwer Academic/Plenum Publishers, New York, USA, 2001.

[2] G. Valette, S. Prévost, L. Lucas, and J. Léonard, "Soda project: a simulation of soil surface degradation by rainfall," *Computers & Graphics*, vol. 30, no. 4, pp. 494–506, 2006.

[3] A. D. Kelly, M. C. Malin, and G. M. Nielson, "Terrain simulation using a model of stream erosion," in *SIGGRAPH'88: Proceedings of the 15th annual conference on computer graphics and interactive techniques*, 1988, pp. 263–268.

[4] P. Prusinkiewicz and M. Hammel, "A fractal model of moutains with rivers," in *Proceedings of Graphics Interface '93*, 1993, pp. 128–137.

[5] F. K. Musgrave, C. E. Klob, and R. S. Mace, "The synthesis and rendering of eroded fractal terrains," in *SIGGRAPH'89: Proceedings of the 16th annual conference on computer graphics and interactive techniques*, 1989.

[6] P. Roudier, B. Peroche, and M. Perrin, "Landscapes synthesis achieved through erosion and deposition process simulation," *Computer Graphics Forum*, vol. 12, no. 3, pp. 375–383, 1993. [Online]. Available: http://dx.doi.org/10.1111/1467-8659.1230375

[7] K. Nagashima, "Computer generation of eroded valley and mountain terrains," *The Visual Computer*, vol. 13, pp. 456–464, 1998, 10.1007/s003710050117. [Online]. Available: http://dx.doi.org/10.1007/s003710050117

[8] B. Benes and R. Forsbach, "Visual simulation of hydraulic erosion," *Journal of WSCG*, vol. 10, pp. 79–86, 2002.

[9] N. Chiba, K. Muaroka, and K. Fujita, "An erosion model based on velocity fields for the visual simulation of mountain scenery," *Journal of Visualization and Computer Animation*, vol. 9, no. 4, pp. 185–194, 1998.

[10] B. Sutherland and J. Keyser, "Particle-based enhancement of terrain data," in *Siggraph'06 Research Poster*, 2006, p. 96.

[11] B. Benes, V. Tesinsky, J. Hornys, and S. K. Bhatia, "Hydraulic erosion," *Computer Animation and Virtual Worlds*, vol. 17, no. 2, pp. 99–108, 2006.

[12] J. Stam, "Stable fluids," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128. [Online]. Available: http://dx.doi.org/10.1145/311535.311548

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer Graphics Forum*, vol. 26, no. 1, 2007, pp. 80–113.

[14] M. Harris, "Fast fluid dynamics simulation on the gpu," in *GPU Gems*. Addison-Wesley Professional, 2004, ch. 38, pp. 637–666.

[15] E. Wu, Y. Liu, and X. Liu, "An improved study of real-time fluid simulation on gpu," *Journal of Computer Animation and Virtual World*, vol. 15, no. 3-4, pp. 139–146, 2004.

[16] Y. Liu, X. Liu, and E. Wu, "Real-time 3d fluid simulation on gpu with complex obstacles," in *Proceedings of Pacific Graphics'04*, 2004, pp. 247–256.

[17] W. Li, X. Wei, and A. Kaufman, "Implementing lattice boltzmann computation on graphics hardware," *The Visual Computer*, vol. 19, no. 7-8, pp. 444–456, 2003.

[18] A. Monitzer, "Fluid simulation on the gpu with complex obstacles using the lattice boltzmann method," Master's thesis, UT Vienna, Institute of Computer Graphics and Algorithms, 2008.

[19] D. A. Randall, "The shallow water equations," Department of Atmospheric Science Colorado State University, Fort Collins, Colorado 80523, Tech. Rep., 2006.

[20] M. Kass and G. Miller, "Rapid, stable fluid dynamics for computer graphics," in *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, 1990, pp. 49–57.

[21] B. Benes, "Real-time erosion using shallow water simulation," in *VRIPHYS'07: 4th Workshop in Virtual Reality Interactions and Physical Simulation*, 2007.

[22] J. O'Brien and J. K. Hodgins, "Dynamic simulation of splashing fluids," in *Proceedings of Computer Animation '95*, 1995, pp. 198–205.

[23] X. Mei, P. Decaudin, and B.-G. Hu, "Fast hydraulic erosion simulation and visualization on GPU," in *15th Pacific Conference on Computer Graphics and Applications, Pacific Graphics 2007, November, 2007*. Maui, Hawaii, Etats-Unis: IEEE, Nov. 2007, pp. 47–56.

[24] O. Stava, B. Benes, M. Brisbin, and J. Krivanek, "Interactive terrain modeling using hydraulic erosion," M. Gross and D. James, Eds. Dublin, Ireland: Eurographics Association, 2008, pp. 201–210. [Online]. Available: http://www2.tech.purdue.edu/cgt/Facstaff/bbenes/private/papers/Stava08SCA.zip

[25] A. Selle, R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac, "An unconditionally stable maccormack method." journal of scientific computing (in review). available online at," *J. Sci. Comput*, 2007.

[26] J. Jákó, "Fast hydraulic and thermal erosion on gpu," in *Proceedings of CESCG 2011: The 15th Central European Seminar on Computer Graphics*, 2011.

[27] P. Y. Julien and D. B. Simmons, "Sediment transport capacity of overland flow," in *Transactions of the ASAE, Vol. 28, No. 3*. St. Joseph, Michigan: American Society of Agricultural Engineers, 1985, pp. 755–762.

[28] J. Grotzinger, T. Jordan, F. Press, R. Siever, and V. Schweizer, *Press/Siever- Allgemeine Geologie*, ser. SAV Geowissenschaften. Spektrum Akademischer Verlag, 2007.

[29] H. L. Penman, "Natural Evaporation from Open Water, Bare Soil and Grass," *Royal Society of London Proceedings Series A*, vol. 193, pp. 120–145, April 1948.

[30] K. Group, *The OpenCL Specification - Version 1.2*, Khronos Group, 2011. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[31] B. Gaster, D. Kaeli, L. Howes, P. Mistry, and D. Schaa, *Heterogeneous Computing With Opencl*. Elsevier Science & Technology, 2011.

[32] AMD, *AMD Accelerated Parallel Processing OpenCL™ Programming Guide (v1.3f)*, Advanced Micro Devices Inc., 2011. [Online]. Available: http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[33] M. M. Maes, T. Fujimoto, and N. Chiba, "Efficient animation of water flow on irregular terrains," in *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, ser. GRAPHITE '06. New York, NY, USA: ACM, 2006, pp. 107–115. [Online]. Available: http://doi.acm.org/10.1145/1174429.1174447

[34] B. Neidhold, M. Wacker, and O. Deussen, "Iterative physically based fluid and erosion simulation," in *Eurographics Workshop on Natural Phenomena '05*, 2005, pp. 25–32.

[35] nVidia, *nVidia OpenCL Best Practices Guide*, NVIDIA Corporation, 2011.

List of Figures

List of Tables

List of Abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
DSP	Digital Signal Processor
OpenGL	Open Graphics Language
GLUT	GL Utility Toolkit
GLSL	GL Shading Language
DevIL	Developers Image Library
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
CUDA	Compute Unified Device Architecture
Cg	C for graphics
LBM	Lattice Boltzmann Method
SWE	Shallow Water Equations
BFECC	Back $ Forth Error Compensation and Correction
SIMD	Single Input Multiple Data
VLIW	Very Large Instruction Word
SSE	Streaming SIMD Extensions
IL	Intermediate Language
API	Application Programming Interface
SLI	Scalable Link Interface
FPS	Frames per Second
PSD	Photoshop Document
RNG	Random Number Generator
LKG	Linear Congruential Generator