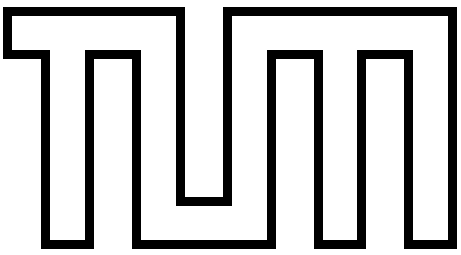


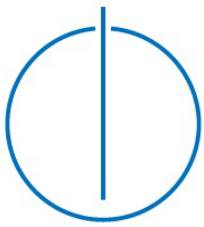
Technische Universität München

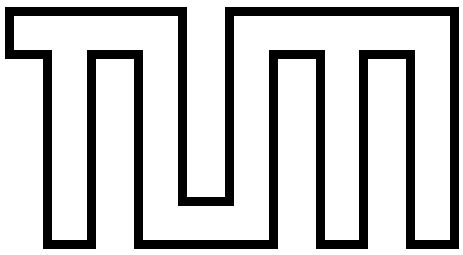
Fakultät für Informatik

Bachelor's Thesis in Informatics

Fast Voxel-Based Hydraulic Erosion

Sebastian Weiß





Technische Universität München

Fakultät für Informatik

Bachelor's Thesis in Informatics

Fast Voxel-Based Hydraulic Erosion

Schnelle Voxel-basierte hydraulische Erosion

Author: Sebastian Weiß

Supervisor: Prof. Dr. Rüdiger Westermann

Advisor: M.Sc. Florian Ferstl

Submission: 15.05.2016

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, 15.05.2016

(Sebastian Weiß)

Abstract

Simulating realistic looking hydraulic erosion can greatly increase the realism of terrains in computer graphics. Other areas of application include the fast evaluation of possible erosion scenarios which is needed in extreme weather conditions. The methods presented in this thesis are based on the SPH-method for simulating fluids using particles in 3D. The proposed extensions include the precise collision with a complex terrain represented as a sparsely stored level set. Furthermore, each particle carries its own amount of sediment which is exchanged with the terrain to simulate dissolving and deposition and with other particles to simulate sediment diffusion. By using particles and a level set instead of heightmaps, caves, overhangs and other complicated structures can be generated. The presented methods are fast enough to simulate and render up to 100,000 particles on a terrain with a resolution of 1024*1024*512.

Zusammenfassung

Die Simulation von realistisch aussehender Erosion durch Wasser kann den Realismus und Detailreichtum von virtuellem Terrain deutlich erhöhen. Andere Einsatzgebiete sind zum Beispiel eine schnelle Überprüfung möglicher Erosionsszenarien bei extremen Wetterbedingung. Die vorgestellten Methoden basieren auf dem SPH-Verfahren zur Simulation von Flüssigkeiten mittels Partikeln in 3D. Die Erweiterungen an dieser Methode umfassen unter anderem eine präzise Kollision der Partikel mit dem Terrain, welches als Level-Set gepeichert wird. Des Weiteren trägt jedes Partikel Sediment mit sich. Die Partikel tauschen Sediment mit dem Terrain aus um Ablösung und Ablagerung zu simulieren, und mit anderen Partikeln für Sedimentdiffusion. Durch die Benutzung von Partikeln und einem Level-Set statt Höhenfeldern können Höhlen, Überhänge und andere komplizierte Strukturen erzeugt werden. Die vorgestellten Methoden sind schnell genug um bis zu 100000 Partikel auf einem Terrain mit einer Auflösung von 1024*1024*512 zu simulieren.

Contents

1. Introduction

Many features of natural terrains are results of erosion. Water carves river beds and forms caves, chasms and waterfalls, wind grinds hard corners away and builds huge dunes, and plant roots can break up even the hardest rock. The simulation of erosion is a well studied field in computer graphics as it greatly increases the realism and richness of detail of virtual landscapes that would be very hard to achieve by hand.

In this thesis, the focus lies only on hydraulic erosion because this is the effect that is most noticeable in our daily life. It is important for farmers and even for gardeners when heavy rain washes the newly spread potting soil away.

The goal is to simulate the effects of hyraulic erosion fast and realistically so that the results can be visualized directly. Therefore, the proposed methods favor speed over physical correctness. Furthermore, we want to be able to simulate the creation of river beds, cliffs, overhangs, waterfalls and caves, etc., in a few seconds.

We propose a method for simulating erosion on a full 3D grid of voxels. In order to keep the simulation fast, design decisions are made so that the algorithms fit on the parallel architecture of modern GPUs. In our implementation, every computation, including the simulation and rendering, is done completely on the GPU. The CPU only fulfills organization tasks. This is made possible by the flexible programming model provided by OpenCL1.2 and interoperability between DirectX11 and OpenCL1.2.

The terrain is stored as a level set [7, 53, 12] using a two-layer regular grid hierarchy. This is a good tradeoff between random access speed and cache usage on the one side and sparsity on the other side. For the simulation, we couple a fluid solver based on smoothed particle hydrodynamics from [39] with an adoption of the erosion equations from [57]. In each simulation step, the particles interact with each other and collide with the terrain. At collisions, each particle can dissolve or deposit sediment that it carries around.

This thesis is structured in the following way: We first review related work in chapter 2. In chapter 3 we present our proposed methods, first the terrain representation in section 3.1, then the fluid simulation in section 3.2 which is split into the smoothed particle hydrodynamic method in section 3.2.1, the erosion equations in section 3.2.2 and in optimization techniques in section 3.2.3. Rendering methods are described in section 3.3. Results are presented in section 4. Finally, we discuss our work in section 5.

2. Related Work

The generation of naturally looking terrain is of great interest in the literature because of its importance for computer games.

Heightmap based algorithms are often used for real-time applications. For example, [23] generates heightmaps with various types of noise, [1] based on voronoi diagrams, [30] using software agents and [55] using genetic algorithms. Other algorithms focus on the interaction with the user as a designer [20, 14].

Less work has been done to include hydrology in the generation procedure. An interesting approach [25] is to generate rivers first and afterwards the terrain heights.

Attempts to describe the process of hydraulic erosion from a more physical point of view has been done in [34, 46, 35], to only name a few. Despite of their physical accuracy, they require many parameters that have to be measured or estimated and are computationally expensive.

In our thesis, we start with an existing terrain and simulate hydraulic erosion on it. Existing works [4, 57] perform this type of erosion, but are limited to heightmaps. Our methods remove the limitations of heightmaps and allows the generation of caves and overhangs.

Fluid simulations in 3D, which are needed in this thesis, can be separated in two groups: grid based and particle based approaches. Grid based approaches [32, 31, 9, 18] solve the Navier Stokes equation on a regular grid of voxels. Newer research improved the results by using an adaptive grid resolution based on so called tall cells [42] or octrees [16, 58]. All these method use a poison solver for solving the incompressibility of fluids. Lattice Boltzmann Methods (LBM) [28, 51, 18] on the other hand rely on the computation of in- and outflow of each voxel to ensure the incompressibility. Therefore, no global solver is needed. All grid based approaches, however, have the same problem in common: they do not preserve the volume of the fluid. It can happen that small splashes reach a size that is smaller than the grid resolution and then the water disappears. This is a critical problem for our simulation goal because sediment from water splashes would fly in the air.

The other group of algorithms uses particles. Smoothed particle hydrodynamics (SPH) is a way to model the interactions between fluid particles [39, 36, 54, 9]. These algorithms

do not suffer from loss of fluid volumes as grid based approaches, because each particle represents its own quantity of volume. The performance of these methods, however, scales with the number of particles. On recent hardware, it is hard to simulate more than 100,000 particles with interactive frame rates.

Recent work tried to couple grid based and particle based approaches. Flip methods and coupling methods use grid solvers within the fluid and particles on the boundary [15, 43]. These methods combine the advantages of both grid based and particle based methods, but are algorithmically complex.

In our scenarios, the water flow over the terrain is usually quite thin. Therefore, we use a simple SPH technique to simulate the water flow.

3. Voxel-based Hydraulic Erosion

In the following sections, the details of the proposed methods are presented. First, we describe how the terrain is represented and stored. Then we present the algorithms for the fluid and erosion simulation. Third, we briefly explain simple rendering techniques for the terrain and fluid.

3.1. Terrain representation

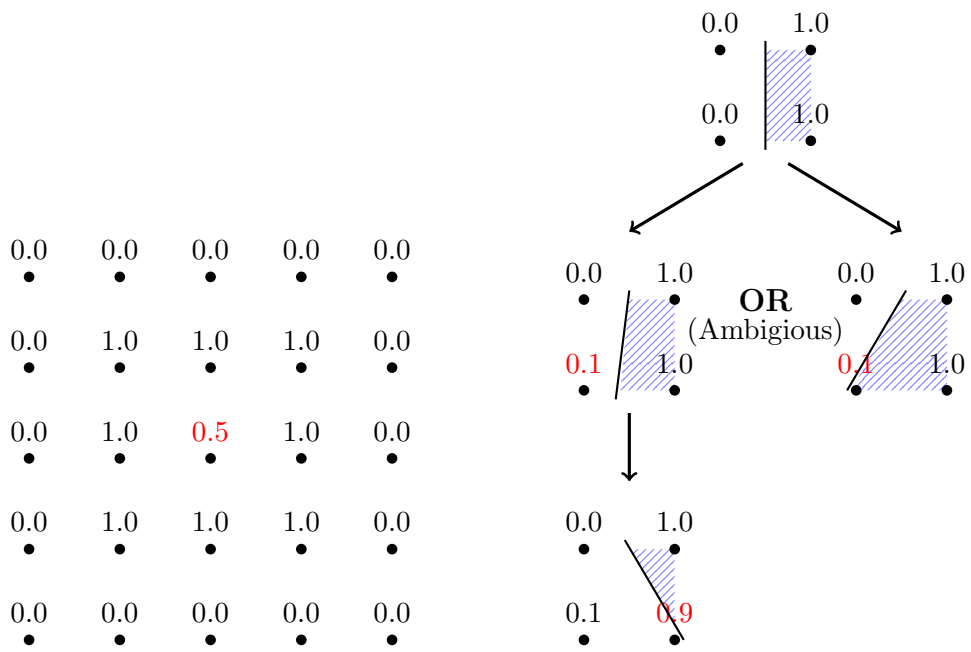
To represent a three-dimensional terrain, several methods have been developed so far:

- Height map: This is the most common terrain representation in computer games. A two-dimensional gray scale texture is used to specify the height of each point on the ground plane [13, 44, 57, 1, 4, 6, 23]. Although this method is very efficient in storage and rendering, caves or overhangs can't be represented by it. Therefore, this method is not useful for achieving our goals.

- Solid voxels: A three dimensional grid of voxels is used where each voxel is either completely filled or completely empty. This leads to a "Minecraft"-style look of the terrain. Efficient render techniques have been developed for this type of terrain using octrees or meshing [41, 24, 50, 10]. But the lack of sub-voxel accuracy or expensive modifications makes this unsuitable for erosion simulations.

- Density volumes: This technique might be the most popular one for rendering volumetric datasets such as clouds, smoke or medical data like CT scans. Each voxel stores a float ranging from 0 (empty) to 1 (full) [47, 27, 8, 37]. This representation eases the formulation of modification operations. Raytracing then is a way to render these transparent scenes. However, when a hard surface boundary is needed, which is required for collision detection, this method shows some serious ambiguity (see fig. 3.1). That makes it also unsuitable for our purpose.

- Level sets and signed distance fields: The surface is represented implicitly by storing the signed distance from the current position to the nearest surface boundary [7, 53, 12]. This method provides a very elegant way to detect whether a point lies within the terrain or not and directly provides the surface normal by evaluating



(a) legal configuration, but no physical meaning as a terrain surface

(b) Ambiguity and jumps in the surface interpolation

Figure 3.1.: Problems with density volumes (in 2D)

the gradient of the signed distance field. The main disadvantage is that surface modifications become quite tricky.

Based on the discussion of these four ways for terrain representation, we decided to use level set methods.

3.1.1. Introduction to level set methods

Level set methods have been used for a long time and are frequently described in literature [7, 53, 12]. They provide an elegant way of formulating many problems in physics or mathematics.

Let $\Omega \subset \mathbb{R}^n$ be the volume to be represented with the boundary $\partial\Omega$. In our case, Ω is the terrain with the surface $\partial\Omega$. Let $\mathbf{x} = (x_1, ..., x_n)^T \in \mathbb{R}^n$ be a point in space. The function

$\varphi : \mathbb{R}^n \to \mathbb{R}$ is called a level set function when the following holds:

$$\begin{array}{lcll} \varphi(\mathbf{x}) & > & 0 & \text{for} \quad x \in \Omega \\ \varphi(\mathbf{x}) & = & 0 & \text{for} \quad x \in \partial\Omega \\ \varphi(\mathbf{x}) & < & 0 & \text{for} \quad x \notin \bar{\Omega} \end{array} \tag{3.1}$$

The region $\Gamma := \partial\Omega = \{\mathbf{x} \in \mathbb{R}^n : \varphi(\mathbf{x}) = 0\}$ is the surface of the volume and is also often called the zero-level.

Fig. 3.2 visualizes a simple level set. The left side shows the represented object (in this case, a 2D object). The level set function takes positive values inside and negative outside. On the right side, φ is shown as a 3D plot with the height indicating the value of φ. The zero level is represented by the blue plane.

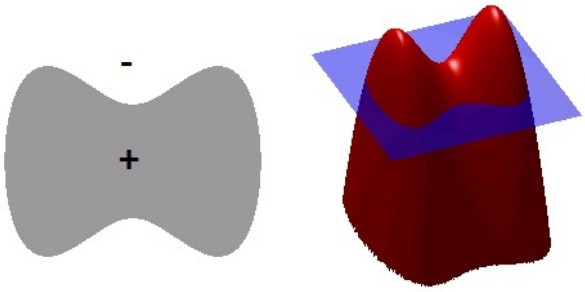


Figure 3.2.: The level set function of the 2D object on the left is plotted in 3D on the right [1]

If the additional property

$$||\nabla\varphi|| = 1 \tag{3.2}$$

holds, then φ defines a signed distance field. For every point $\mathbf{x} \in \mathbb{R}^n$, $|\varphi(\mathbf{x})|$ denotes the shortest distance in the euclidean norm to the volume boundary Γ, and is positive inside and negative outside.

The surface normal $\mathbf{n}(\mathbf{x})$ at position $\mathbf{x}$ is defined as:

$$\mathbf{n}(\mathbf{x}) = -\frac{\nabla\varphi(\mathbf{x})}{||\nabla\varphi(\mathbf{x})||} \tag{3.3}$$

With these tools at hand, we can now analytically describe any surface and any terrain with this level set function. In order to use this function in practice, we have to discretize it.

[1] source (edited): https://en.wikipedia.org/wiki/File:Level_set_method.jpg

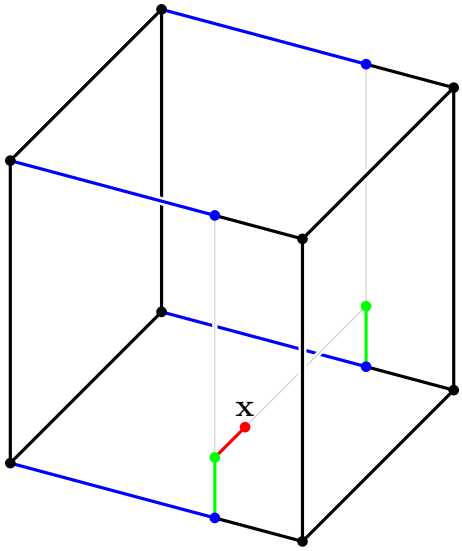


Figure 3.3.: Trilinear interpolation of the level set

3.1.2. Interpolation and normal computation

From now on, we always assume that we work in the three-dimensional space $\mathbb{R}^3$. We store the value of φ at every voxel in a three-dimensional grid.

To query the value of φ at any given point in the space, we perform a trilinear interpolation of the eight surrounding voxels (see 3.3). We denote this function by $L(\mathbf{x})$. It is used in section 3.2.1.4 and its implementation is shown in program 2 (see appendix). OpenCL allows a short formulation of this operation using vector swizzling and the build-in mix-function.

To evaluate the gradient, we compute the gradient of the trilinear interpolation. This leads to the function $\mathbf{N}(\mathbf{x})$ and the implementation is presented in program 3 (see appendix).

3.1.3. Narrow band and renormalization

If the level set φ is a signed distance field (i.e. equation (3.2) holds), then the vector returned by $\mathbf{N}(\mathbf{x})$ is always normalized. However, the terrain modifications that will be presented in 3.1.5 and 3.2.2.4 destroy this property. Therefore we have to renormalize or reinitialize the level set periodically.

Let φ^0 be the current level set. The level set φ, which fulfills eq. (3.2), can be expressed as the solution of the following Eikonal equation [7]

$$\begin{cases} ||\nabla\varphi|| = 1 \\ \text{sgn}(\varphi) = \text{sgn}(\varphi^0) \end{cases} \tag{3.4}$$

with the signum function sgn being either -1, 0 or 1. Solving this equation leads to a normalized level set function.

There are two classes of algorithms for solving this equation. Fast marching algorithms [26, 22] work like a Dijkstra algorithm. They start with a set of active cells at the surface border and march away from these cells. Fast sweeping algorithms [29, 7] are iterative algorithms that solve the Eikonal equation as an optimization problem. Because this class of algorithms can be parallelized on the GPU very straightforward, we chose this in our implementation.

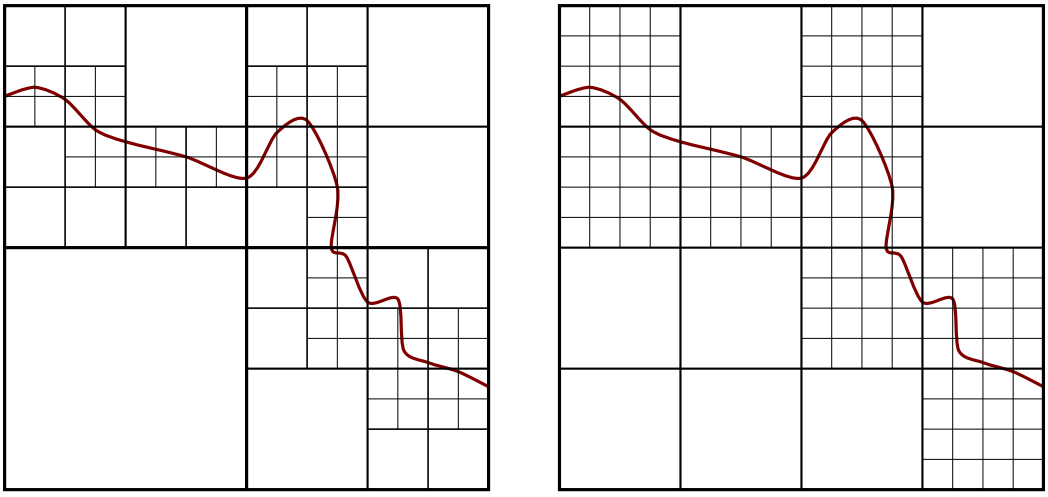
A formulation of the Eikonal equation as an optimization problem looks like [29]

$$\frac{\partial\varphi}{\partial t} = \text{sgn}(\varphi)(1 - ||\nabla\varphi||) \tag{3.5}$$

with a special continuous signum function $\text{sgn}(\varphi) := \frac{\varphi}{\sqrt{\varphi^2 + ||\nabla\varphi||}}$.

Experiments have shown that both the simple algorithm of [29] and the more advanced solver of [7] show the same problem: they move the surface a little bit. When the cells along the surface boundary are updated during a fast sweeping iteration, small changes in the fractions between two cells change the location of the boundary. This is a severe problem for the erosion simulation in 3.2.2.

We went a very drastic way to solve the problem: cells that lie on the boundary (i.e. at least one of its six neighbors has a different sign) are not allowed to change their value. Therefore, the surface interface does not change. But of course, the level set is not a signed distance function at the boundary anymore. In practice, this is not an issue because we only have to determine if a position is inside or not and to compute the gradient for the collision (see 3.2.1.4). The renormalization is only used to limit the error when the water carves its way through many layers of voxels.



(a) A sparse voxel octree **(b)** Two-layer grid hierarchy

Figure 3.4.: Two ways of storing the same surface

3.1.4. Sparse storage

The goal is to simulate a large terrain. A big area of the terrain, however, will be completely filled with air or with stone and only a small area will contain the terrain boundary. Therefore, a way to utilize this sparse structure in a storage technique has to be found.

An established approach is the use of a sparse voxel octree (SVO) [11, 49, 33], in which voxels are organized in a sparse octree where each leave is either a single voxel or a small block of voxels (see fig. 3.4a). Although this method is very efficient to render, it has a serious drawback: The strong relationship between neighbors is lost. One can't just access the neighbors of a given voxel with pointer arithmetic. Instead, the tree has to be traversed, which can be time-consuming. The access of neighboring voxels is needed very often in our algorithms, e.g. in 3.1.2 and 3.1.3. Therefore, a SVO is unsuitable for our purposes.

Instead we use a two-layer grid hierarchy (see fig. 3.4b). A coarse grid first subdivides the 3d space into smaller cells. If the terrain surface runs through such a coarse cell, it is subdivided into a fine grid. The latter then stores the voxel data.

As you can see in the comparison figure 3.4, the two-layer grid hierarchy stores more voxels. However, no costly index structure as for the SVO is needed and also most neighbor lookups can be resolved inside the same fine grid. In our experiments, we use fine grid sizes of 8^3 or 16^3.

The algorithms in the next sections need the following information per voxel, stored in two single-precision float entries:

- the value of the level set φ at the voxel.
- a per-voxel soil hardeness $K_{s,\mathbf{p}_i}$ (see 3.2.2.2).

Because the terrain is modified by the erosion simulation (see 3.2.2), a test that checks whether the terrain surface approaches a non-subdivided cell and create new fine cells if necessary, has to be executed periodically.

With this storage technique we are able to execute the whole simulation on a grid with a size of up to 1024*1024*512.

3.1.5. Generation from heightmaps

Unfortunately, it is difficult to find any volume data from real-world terrain. The only available sources are height maps from satellite scans. Such a height map can be seen in image 3.5. White pixels indicate higher areas, black pixels lower areas.

To generate the 3D grid from the heightmap, we first align the voxel grid with the height map so that the xy-plane of the grid is the image plane of the height map. For simplicity, the grid resolution in x and y direction is the same as the height map. The height is then mapped on the z coordinate of a voxel. The task is now to compute the shortest distance from a voxel in the 3D space to the height map. This value, with the correct sign, is then used for the level set function φ.

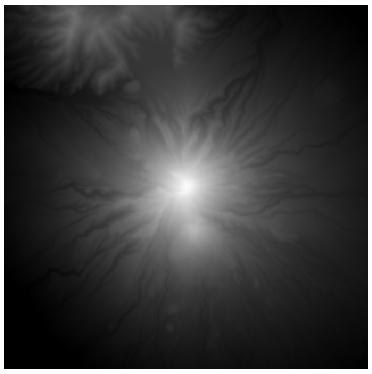


Figure 3.5.: A sample heightmap [2], white indicates a high area, black a low area

[2]Retrieved 04/20/2016 from http://johnflower.org/tutorial/finding-height-maps-web

Let w and d be the width and the depth of the grid (i.e. the size on the ground plane), and h be the height in grid units. Then the position of each voxel is $\mathbf{p} \in \mathbb{Z}^3$ with $0 \leq \mathbf{p}_x \leq w-1$, $0 \leq \mathbf{p}_y \leq d-1$, $0 \leq \mathbf{p}_z \leq h-1$. Furthermore, let $h(x,y), h : \mathbb{Z}^2 \to [0, h-1]$ represent the height map, with $0 \leq x \leq w-1$, $0 \leq y \leq d-1$.

We use a simple approach. The distance of a voxel at position $\mathbf{p}$ to the height map is defined as

$$\begin{aligned} d = \min(\quad & |h(\mathbf{p}_x, \mathbf{p}_y) - \mathbf{p}_z|, \\ & ||(1, h(\mathbf{p}_x + 1, \mathbf{p}_y) - \mathbf{p}_z)^T||, \\ & ||(-1, h(\mathbf{p}_x - 1, \mathbf{p}_y) - \mathbf{p}_z)^T||, \\ & ||(1, h(\mathbf{p}_x, \mathbf{p}_y + 1) - \mathbf{p}_z)^T||, \\ & ||(-1, h(\mathbf{p}_x, \mathbf{p}_y - 1) - \mathbf{p}_z)^T||, \\ & p((\mathbf{p}_x, h(\mathbf{p}_x, \mathbf{p}_y))^T, (\mathbf{p}_x - 1, h(\mathbf{p}_x - 1, \mathbf{p}_y))^T, (\mathbf{p}_x, \mathbf{p}_z)^T), \\ & p((\mathbf{p}_x, h(\mathbf{p}_x, \mathbf{p}_y))^T, (\mathbf{p}_x + 1, h(\mathbf{p}_x + 1, \mathbf{p}_y))^T, (\mathbf{p}_x, \mathbf{p}_z)^T), \\ & p((\mathbf{p}_y, h(\mathbf{p}_x, \mathbf{p}_y))^T, (\mathbf{p}_y - 1, h(\mathbf{p}_x, \mathbf{p}_y - 1))^T, (\mathbf{p}_y, \mathbf{p}_z)^T), \\ & p((\mathbf{p}_y, h(\mathbf{p}_x, \mathbf{p}_y))^T, (\mathbf{p}_y + 1, h(\mathbf{p}_x, \mathbf{p}_y + 1))^T, (\mathbf{p}_y, \mathbf{p}_z)^T) \;) \end{aligned} \tag{3.6}$$

with the distance between a line ($\mathbf{a}$ - $\mathbf{b}$) and a point $\mathbf{x}$ in 2D calculated as

$$p(\mathbf{a}, \mathbf{b}, \mathbf{x}) := \left| \frac{(\mathbf{b}_y - \mathbf{a}_y, \mathbf{a}_x - \mathbf{b}_x)^T}{||(\mathbf{b}_y - \mathbf{a}_y, \mathbf{a}_x - \mathbf{b}_x)^T||} \cdot (\mathbf{x} - \mathbf{a}) \right|. \tag{3.7}$$

The computed distances in the one-dimensional case are shown in fig. 3.6. Green is the interpolated height map, the dotted lines are the computed distances. The red dotted line is the shortest of these distances.

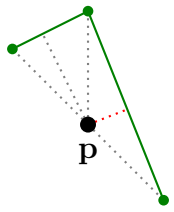


Figure 3.6.: simple point-heightmap distance

To determine if a voxel is inside (positive sign) or outside (negative sign) of the terrain, we set the sign to $sgn(h(\mathbf{p}_x, \mathbf{p}_y) - \mathbf{p}_z)$.

This algorithm only works along the border of the height map. Therefore, a renormalization step as described in 3.1.3 is needed.

3.2. Particle-based fluids

To simulate the flow of water, we are using a particle based method based on Smoothed Particle Hydrodynamics (SPH) [39].

SPH is a technique to approximate the Navier-Stokes equation [32, 39]

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v} \cdot \nabla)\mathbf{v} - \frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\Delta \mathbf{v} + \mathbf{f} \tag{3.8}$$

with respect to the incompressibility constraint

$$\nabla \cdot \mathbf{v} = 0 \tag{3.9}$$

using particles. Here, $\mathbf{v}$ is the velocity vector field, t the time step, p the pressure scalar field, ρ the mass density scalar field, $\mathbf{f}$ the external forces and μ the viscosity. We directly adopted the formulas presented in [39] as shown in the chapters 3.2.1, 3.2.1.2 and 3.2.1.3

For each particle with index i we define the following quantities: the position $\mathbf{p}_i$, the mass m_i, the density ρ_i, the pressure p_i, the velocity $\mathbf{v}_i$, the acceleration $\mathbf{a}$ and several forces $\mathbf{f}_i$. To support erosion the following quantities are also needed: the sediment that the particle carries s_i, the sediment capacity C_i and to support evaporation, the water mass per particle w_i.

3.2.1. Smoothed particle hydrodynamics

The key idea behind Smoothed Particle Hydrodynamics (SPH) is that a quantity can be evaluated at any point in the space by interpolating particles around it. Let A be the quantity that should be evaluated at point $\mathbf{r}$ and let A_j be the queried quantity at the position of the particle. Then A is computed as

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{p}_j, h) \tag{3.10}$$

In this equation, the function $W(\mathbf{r}, h)$ denotes the smoothing kernel with core radius h. This function specifies how individual particles contribute to the final quantity. These kernels are described in detail in the next section.

In the fluid simulation, we include the effects of pressure between particles, causing repulsion forces, and viscosity. The latter describes a kind of "thickness" of the fluid.

(a) **(b)**

Figure 3.7.: Comparison of a water fall with water pouring into a glass [3]

External forces are realized with gravity and terrain collisions. Furthermore, we include surface tension which tries to minimize the curvature of the surface, although its effect on fluids over a large time scale and world scale from meters to kilometers is not always noticeable. A waterfall (3.7a) shows almost no surface tension, a lot of spray can be seen. On a smaller scale, e.g. when water is poured into a glass (3.7b), surface tension is very important because it keeps the water jet together.

3.2.1.1. Particle mass

The particle mass is influenced by both the water it carries, to support evaporation, and the sediment, to support dissolving and deposition. Therefore, the particle mass is defined as

$$m_i = \omega w_i + \sigma s_i \tag{3.11}$$

with ω specifying the contribution of the water to the mass of each particle and σ the contribution of the sediment. Each particle starts with a water mass w_i of 1. This water mass is changed during evaporation 3.2.2.6

[3] waterfall: http://www.fanpop.com/clubs/mother-nature/images/27716536/title/waterfall-photo, accessed on 04/06/2016, water glass: http://www.ntptalk.com/articles/water-is-vital-for-good-health.php, accessed on 04/06/2016

3.2.1.2. Smoothing kernels

We use three different smoothing kernels in total. All kernels should be even (i.e. $W(-\mathbf{r}, h) = W(\mathbf{r}, h)$) and normalized (i.e. $\int W(\mathbf{r}, h)\,\mathrm{d}\mathbf{r} = 1$). Furthermore, kernels whose derivatives at the boundary vanish increase the stability of the algorithm.

All of the following three kernels [39] and their derivatives are defined in the region $||\mathbf{r}|| \leq h$ and zero otherwise. For simplification of the notation, we set $r := ||\mathbf{r}||$.

The W_{poly6} kernel is used for pressure computation and surface tension. Because r only appears squared, it is fast to compute.

$$W_{\text{poly6}}(\mathbf{r}, h) := \frac{315}{64\pi h^9}(h^2 - r^2)^3 \tag{3.12}$$

$$\nabla W_{\text{poly6}}(\mathbf{r}, h) := \frac{-945(h^2 - r^2)^2}{32\pi h^9} \begin{pmatrix} \mathbf{r}_x \\ \mathbf{r}_y \\ \mathbf{r}_z \end{pmatrix} \tag{3.13}$$

$$\Delta W_{\text{poly6}}(\mathbf{r}, h) := \frac{945(h^2 - r^2)r^2}{8\pi h^9} - \frac{2835(h^2 - r^2)^2}{32\pi h^9} \tag{3.14}$$

The next kernel, W_{spiky} is used for repulsion force computation. Its gradient does not vanish when two particles are getting close together, so they do not clot together.

$$W_{\text{spiky}}(\mathbf{r}, h) := \frac{15}{\pi h^3}(h - r)^3 \tag{3.15}$$

$$\nabla W_{\text{spiky}}(\mathbf{r}, h) := \frac{-45(h - r)^2}{\pi h^6 r} \begin{pmatrix} \mathbf{r}_x \\ \mathbf{r}_y \\ \mathbf{r}_z \end{pmatrix} \tag{3.16}$$

In the two kernels described above, the laplacian gets negative when $\mathbf{r}$ is small. This causes stability issues when computing the viscosity forces where the laplacian is needed. To solve this, a third kernel is used for that special purpose [39].

$$W_{\text{viscosity}}(\mathbf{r}, h) := \frac{15}{2\pi h^3}(-\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1) \tag{3.17}$$

$$\Delta W_{\text{viscosity}}(\mathbf{r}, h) := \frac{45}{\pi h^6}(h - r) \tag{3.18}$$

With the kernels in place, the formulas for pressure, viscosity and surface tension forces can now be described.

3.2.1.3. Density, pressure, repulsion, viscosity

The following equations are directly taken from [39].

First, the density for each particle i is computed:

$$\rho_i = \sum_j m_j W_{\text{poly6}}(\mathbf{p}_i - \mathbf{p}_j, h) \tag{3.19}$$

Second, from the density, the particle pressure is retrieved by

$$p = k(\rho - \rho_0) \tag{3.20}$$

with k being a gas constant. The higher k is, the stronger is the repulsion force and the particles are further away from each other. The rest density ρ_0 is a small value that does not change the gradient because it is constant, but increases the numerical stability.

Third, with the pressure, the repulsion or pressure forces can be computed per particle by

$$\mathbf{f}_i^{\text{pressure}} = -\sum_{j,j\neq i} m_j \frac{p_i + p_j}{2\rho j} \nabla W_{\text{spiky}}(\mathbf{p}_i - \mathbf{p}_j, h). \tag{3.21}$$

Here we have to exclude the current particle in the sum because the nabla-operator of the spiky kernel is not well defined for a zero $\mathbf{r}$.

Next, the viscosity term is evaluated as

$$\mathbf{f}_i^{\text{viscosity}} = \mu \sum_{j,j\neq i} m_j \frac{\mathbf{v}_j}{\rho_j} \Delta W_{\text{viscosity}}(\mathbf{p}_i - \mathbf{p}_j, h). \tag{3.22}$$

Finally, for the surface tension, an additional quantity called the "color field" is defined first which is 1 at particle positions and 0 otherwise.

$$c_s(\mathbf{p}) = \sum_j m_j \frac{1}{\rho_j} W_{\text{poly6}}(\mathbf{p} - \mathbf{p}_j) \tag{3.23}$$

The gradient of the color field

$$\mathbf{n} = \nabla c_s \tag{3.24}$$

gives the surface normal pointing inside the fluid. Inside and outside the fluid, $\mathbf{n}$ vanishes. Furthermore, the laplacian measures the curvature of the fluid

$$\kappa = \frac{-\Delta c_s}{||n||}. \tag{3.25}$$

Putting all this parts together results in the surface tension force

$$\mathbf{f}_i^{\text{surface}} = -\sigma \Delta c_s \frac{\mathbf{n}}{||\mathbf{n}||} \tag{3.26}$$

with the constant σ specifying the strength of the force. Note that for numerical stability, this equation is only evaluated if $||\mathbf{n}||$ exceeds a small threshold.

3.2.1.4. Advection and collision

In order to move each particle, all forces (pressure, viscosity and gravity) are summed and the particle acceleration $\mathbf{a}$ is computed from it.

$$\mathbf{a}_i = (\mathbf{f}_i^{\text{pressure}} + \mathbf{f}_i^{\text{viscosity}} + \mathbf{f}_i^{\text{surface}} + \mathbf{f}^{\text{gravity}})/m_i' \tag{3.27}$$

Because this equation is derived from $F = m\,a$, lighter particles are accelerated more quickly. This results in stability issues when the water mass w_i vanishes due to evaporation 3.2.2.6. Therefore, we use a different equation for the particle mass here:

$$m_i' = \omega + \sigma s_i \tag{3.28}$$

The difference to eq. (3.11) is that we assume w_i to be 1 during particle advection.

Now the velocity of the particle is updated by

$$\mathbf{v}_i \leftarrow \mathbf{v}_i + t\mathbf{a}_i \tag{3.29}$$

and the new position becomes

$$\mathbf{p}_i \leftarrow \mathbf{p}_i + t\mathbf{v}_i. \tag{3.30}$$

The final goal is to simulate erosion. To do so, the water particles have to collide with the terrain first. Section 3.1.2 provides us with two functions for accessing the terrain. $L(\mathbf{x})$

returns the value of the level set at this point and $\mathbf{N}(\mathbf{x})$ its normal. The task is now to find the time when a particle hits the terrain and to reflect the particle on the surface.

A collision happens when $L(\mathbf{x})$ becomes zero. Because we cannot solve for the exact time of this event analytically, since no implicit terrain representation is available for the whole terrain, we have to move the particle step by step and test when $T(\mathbf{x})$ changes from negative (outside) to positive (inside).

Algorithm 1 (see appendix) shows the collision resolution, it replaces equation 3.30. It is called for every particle, therefore, the index i on every particle property is dropped for simplicity. The algorithm is visualized in figure (3.8).

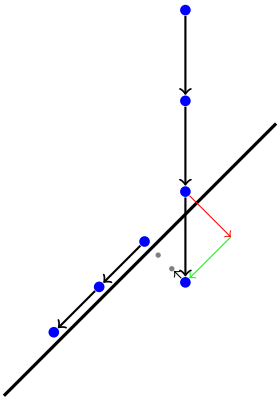


Figure 3.8.: Collision resolution

In this algorithm, the travel distance is computed first (line 1). Because we can only test if a position is inside or outside the terrain, we have to proceed in small steps, leading to the loop starting in line 5. Since we want the moving step size to be independent from the speed of the movement, which ensures that all particles collide with the same precision, we normalize the direction (line 3) and store the remaining time in t_{max} (line 2). The step size is defined by a constant t_{step} (line 6). After we move the particle by one step (line 6-7), we test if the particle is now inside the terrain (line 9). In that case, we first move the particle out along the surface normal (line 11-13). In order to simulate a complete inelastic collisions where the terrain has infinite mass, the normal component of the particle velocity is set to zero in line 14. Finally, we scale t_{max} to fit the new velocity.

In image (3.9), the collision algorithm can be seen in action as the water flows over the terrain.

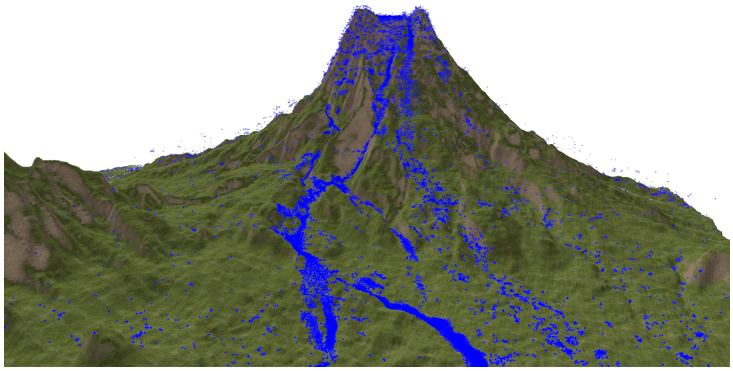


Figure 3.9.: Water particles flow over the terrain

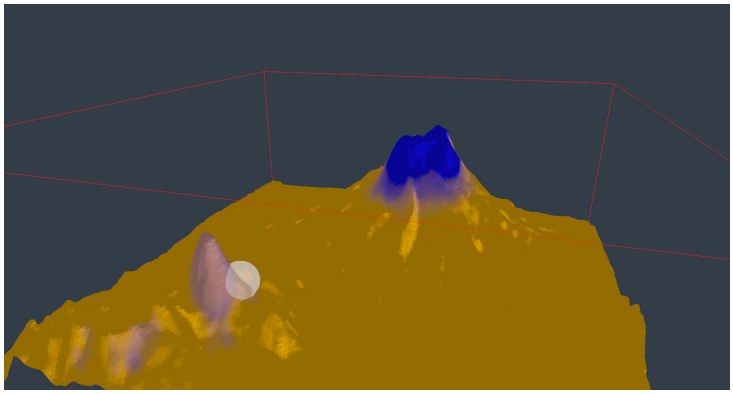


Figure 3.10.: Rain density texture: blue indicates a high density and more water particles are sampled there

3.2.1.5. Rain sampling

We provide two ways to define the water sources. The first method is a rain density texture that the user can modify with a brush (fig. 3.10). In each frame, a specific number of particles is sampled using an OpenCL kernel that is outlined in algorithm 2 (see appendix). This kernel is executed with one thread per particle that should be created. It requires a buffer of seeds for the random number generation that is passed from call to call.

The second method is to specify a water source explicitly 3.11. The user can create sources in the shape of a box, with customizable position, size, initial velocity, particles per frame and initial sediment.

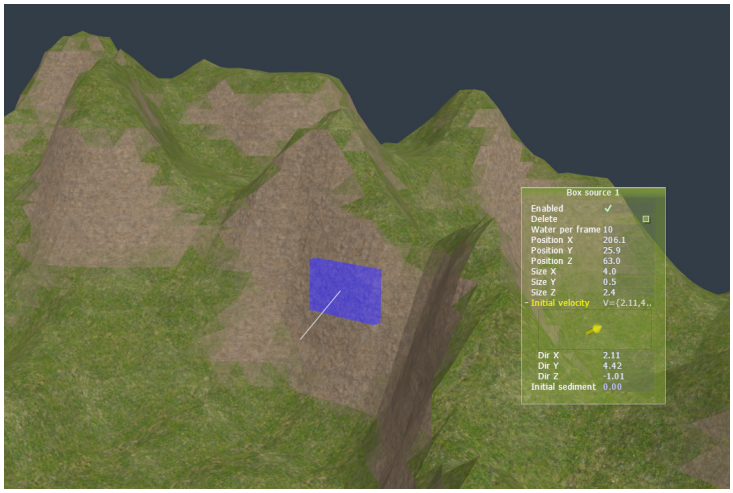


Figure 3.11.: A box as a water source

3.2.2. Hydraulic erosion

To simulate the process of erosion, we adopt and modify the equations from [57]. These equations were used initially to simulate erosion on a 2D height map using the shallow water equation.

Hydraulic erosion consists mainly of these phenomena: dissolving, transportation and deposition of sediment [57]. We propose an algorithm that is based on the assumption that a water particle picks up terrain (dissolving) if it is under-saturated with sediment and drops terrain (deposition) if it is over-saturated.

Figure 3.12 shows the influence of the particle mass. Particles that carry a lot of sediment (left water source) are heavier and therefore accelerate more slowly than the lighter particles without sediment.

3.2.2.1. Sediment capacity

The sediment capacity C_i of a particle i is calculated as:

$$C_i = K_c||\mathbf{v}_i|| \tag{3.31}$$

where K_c is a user-defined sediment capacity constant. This formula is proportional to the particle's speed which follows the intuition that fast flowing water can transport more sediment than slow moving water.

Figure 3.12.: Heavy particles on the left accelerate slower than light particles on the right

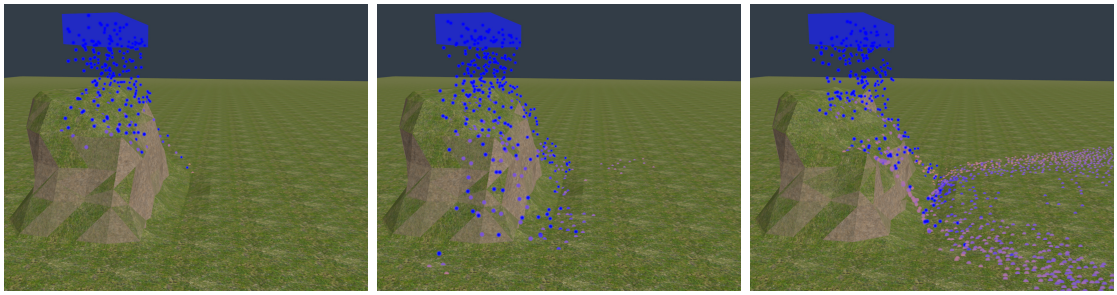
3.2.2.2. Dissolving

The sediment capacity is then compared with the sediment volume s_i that the particle carries. If $C_i > s_i$, the particle dissolves some sediment. The amount of dissolved sediment Δs is computed as:

$$\Delta s = (K_{s,n}||\mathbf{w}_i|| + K_{s,t}||\mathbf{v_i} - \mathbf{w}_i||)(C_i - s_i)K_{s,\mathbf{p}_i} \tag{3.32}$$

where $\mathbf{w}_i = \mathbf{n}(\mathbf{v}_i \cdot \mathbf{n})$ is the component of the particle velocity normal to the terrain surface (with normal $\mathbf{n}$), $K_{s,n}$ is the user-defined dissolving constant for normal movement, $K_{s,n}$ is the user-defined dissolving constant for tangential movement and $K_{s,\mathbf{p}_i}$ is the terrain hardness at the position of the particle.
Because we can specify the influence of the movement that is normal and tangential to the surface separately, we can e.g. say that only normal movement erodes terrain, resulting in a fast creation of holes. Experiments have shown that the ratio $K_{s,n}/K_{s,t} = 1$ results in visually pleasing and plausible results.
The terrain hardness $K_{s,\mathbf{p}_i}$ is stored per voxel and interpolated to compute this value at the position of the particle (see 3.1.2). With this factor, we can model different types of soil. For example, we assign a low value of $K_{s,\mathbf{p}_i}$ to stone, and a high value to sand. Therefore, sand is eroded much faster than stone.

Fig. 3.13 shows the effect of the dissolving on a tiny hill.



(a) It starts raining **(b)** The first particles reach the bottom **(c)** After a few simulation steps

Figure 3.13.: Effects of the dissolving simulation. The color indicates the amount of sediment a water particle carries.

3.2.2.3. Deposition

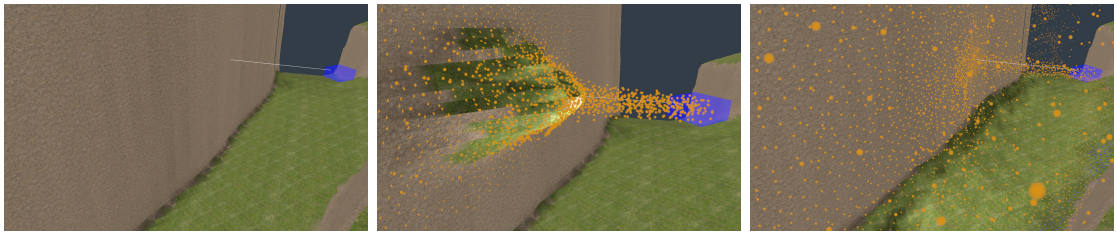
In addition to dissolving, sediment is deposited if $C_i < s_i$. However, we have to include the direction of the surface normal towards gravity. Sediment is only deposited if the angle of the surface is not too steep. Otherwise, the sediment won't stick. Including this intuition into the formula also prevents the creation of "noses": Without this additional factor, highly saturated water that crushes against a vertical wall would build unrealistic bulks (see fig. 3.14). In addition, we clamp the deposition to a small lower bound ϵ. This is necessary because during evaporation, the water mass and therefore C_i vanishes. Without this minimal deposition speed, the sediment mass would get lower and lower, but would never reach zero. The equation then becomes:

$$\Delta s = \min\left(s_i,\, \text{smoothstep}(\alpha_1, \alpha_2, \mathbf{n} \cdot -\mathbf{g})\max(C_i - s_i, \epsilon)K_d\right) \tag{3.33}$$

where K_d is a user-defined deposition constant and $\mathbf{g}$ is the normalized direction of the gravity. Smoothstep smoothly interpolates in the provided interval and is defined as:

$$\begin{aligned} \text{smoothstep}(a, b, x) &:= t^2(3 - 2t) \\ \text{with } t &= \text{clamp}(\frac{x - a}{b - a}, 0, 1) \end{aligned} \tag{3.34}$$

α_1 and α_2 specify the minimal and maximal angles. If the angle between the surface normal and the gravity is steeper than $\cos^{-1}(\alpha_1)$, no sediment is deposited. If the angle is smaller than $\cos^{-1}(\alpha_2)$, this factor is 1 and the deposition is only limited by K_d. In our experiments, we set $\alpha_1 = 0.6$, $\alpha_2 = 0.8$.



(a) The initial scene

(b) Deposition without the correction term based on the surface normal

(c) Deposition with the correction term

Figure 3.14.: Deposition error and the solution. Sediment is only deposited if the terrain is flat enough

3.2.2.4. Particle and terrain update

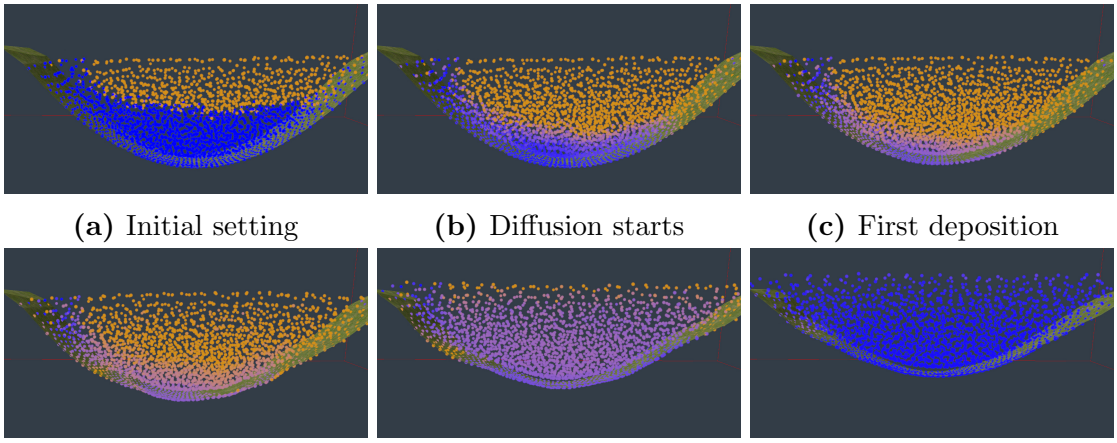
With the amount of change in the sediment Δs, positive for dissolving, negative for deposition, we first update the sediment volume of the particle i by:

$$s_i \mathrel{+}= \Delta s \tag{3.35}$$

The terrain is specified by a level set, see chapter 3.1. A local modification of the terrain at position $\mathbf{p}_i$ is simply a modification of the level set values of the eight surrounding voxels (similar to the interpolation from chapter 3.1.2). This method can be seen in algorithm 3 (see appendix).

Although this algorithm preserves the property of a signed distance field in the neighborhood around the surface, it does not so in the voxels a few steps away from the surface. Therefore, a periodic renormalization step as described in 3.1.3 is necessary. This ensures that the level set of the terrain stays close to a signed distance field even after carving through multiple layers of voxels.

An additional detail has to be kept in mind: All particles are processed in parallel on the GPU. The level set can be modified by many particles at the same time and at the same place, therefore, the access must be synchronized. Since OpenCL 1.2 does not provide an AtomicAdd operation for floats, a workaround must be used (taken from [21]) (see program 4 in the appendix).



(a) Initial setting **(b)** Diffusion starts **(c)** First deposition

(d) **(e)** **(f)** Equilibrium is reached

Figure 3.15.: Sediment diffusion: sediment from the top sinks down and is deposited

3.2.2.5. Sediment diffusion

The dissolving-deposition process described so far has still one unsolved issue, since the exchange of sediment between particles and sediment is only evaluated on collisions: Assume that a lake is filled with water particles that carry no sediment. On the top is a small layer with particles saturated with sediment. Fig. 3.15a shows this scenario, clipping planes are used to show only a cross-section of the lake. Until now, an equilibrium would be reached in that case because the water particles do not change positions and the sediment remains at the top.

This behavior is unrealistic because one key feature is missing: Sediment diffuses through the water and sinks to the bottom because of gravity. We model this sediment diffusion similar to how light is diffused through volumetric material using directed scattering.

A phase function is used to specify the asymmetry of the diffusion [38]. The phase function used here is defined as in [45]

$$p_k(\cos\theta) = \frac{1}{4\pi}\frac{1-k^2}{(1-k\cos\theta)^2}. \tag{3.36}$$

In this equation, $k \in (-1, 1)$ denotes the asymmetry constant. A perfect symmetric diffusion is achieved by $k = 0$, forward-diffusion by $k > 0$ and backward diffusion by $k < 0$ (see 3.16). In our experiments, we found $k = 0.7$ to be a good choice. This value leads to a strong directional diffusion while still allowing diffusion to the sides needed because the particles are never directly located under each other.

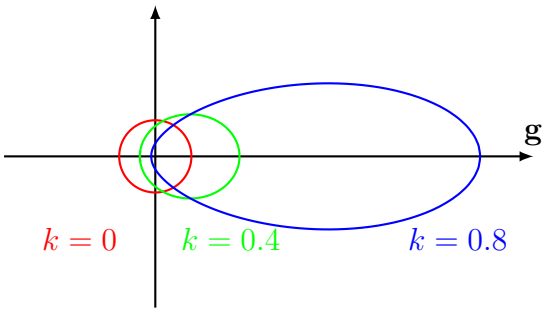


Figure 3.16.: Different choices for the asymmetry parameter

Using the phase function, the diffusion from particle i to particle j is calculated in the framework of SPH as

$$\Delta s_{i,j} = \beta \, p_k(\mathbf{g} \cdot \frac{\mathbf{p}_j - \mathbf{p}_i}{||\mathbf{p}_j - \mathbf{p}_i||}) \, W_{\text{poly6}}(||\mathbf{p}_j - \mathbf{p}_i||, h) \, s_i \tag{3.37}$$

with β specifying the strength of the diffusion. $\Delta s_{i,j}$ is proportional to s_i, following the intuition, that the diffusion is faster the higher the difference in the sediment transportation is.

Summing the sediment exchanges between two particles together, the sediment at particle i is updated as

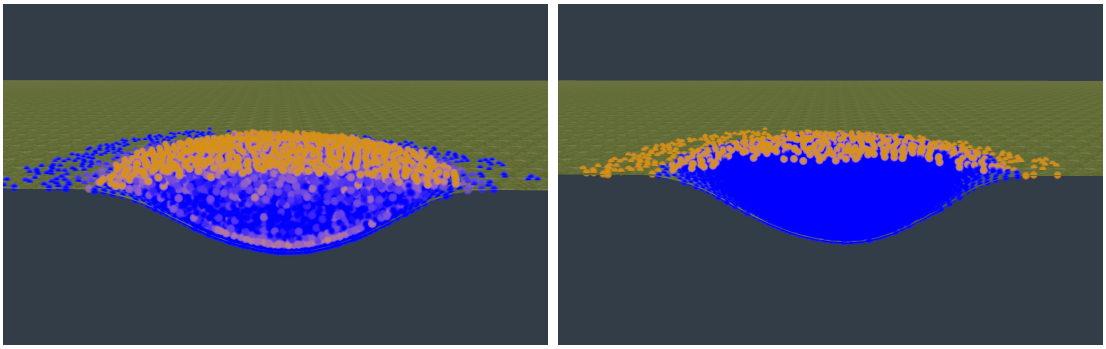
$$\begin{aligned} \Delta s_i &= \sum_{j \neq i} (\Delta s_{j,i} - \Delta s_{i,j}) \\ s_i \; &+= \Delta s_i. \end{aligned} \tag{3.38}$$

Equation (3.38) preserves the sediment mass. No sediment is lost or gained, it is only redistributed (as long as it is not deposited).

The whole diffusion process is shown in fig. 3.15. The sediment sinks down to the bottom of the lake where it is deposited ((b) to (e)). In picture (f), an equilibrium has been reached again where (almost) all the sediment is deposited on the ground. Note that the lake is much less deep than in the initial setting (a).

3.2.2.6. Evaporation

Research in the field of hydrology [3, 19] has shown that the evaporation rate per surface area depends on properties like the temperature and wind speed. By assuming these



(a) Using the z-coordinate of the surface tension

(b) Checking for particles above

Figure 3.17.: Two approaches for finding the surface particles. Orange indicates surface, blue inside.

factors to be constant during the simulation, we can model the evaporation as a constant decrement of the water mass of surface particles.

The problem now becomes finding the surface particles.
The first trial, seen in fig. 3.17a, uses the z-coordinate of normal computed during the surface tension. This normal points inside, therefore, surface particles should have a negative value of $\mathbf{n}_z$. However, this does not work out because this normal can only be computed accurately if the fluid is multiple particles thick. As seen in the figure, particles at the sides have no particles beneath them and are therefore not detected as surface particles.
The second approach can be seen in fig. 3.17b. In this method, a particle is labeled as surface if no other particle lies inside a cone of a certain angle above the current particle. Formally, a particle i is a surface particle if the following holds:

$$\text{Surface}(i) = \bigwedge_{j,j\neq i} (\mathbf{f}^{\text{gravity}} \angle(\mathbf{p}_j - \mathbf{p}_i)) < \alpha \tag{3.39}$$

We choose for the angle α e.g. 70°.

The water mass is now updated using

$$\begin{aligned} \Delta w_i &= -K_e \text{ if Surface}(i) \text{ otherwise } 0 \\ w_i &= \max(0, w_i - \Delta w_i). \end{aligned} \tag{3.40}$$

We delete a particle if w_i becomes zero and the carried sediment s_i falls below a certain small threshold.

3.2.2.7. Alternative approach

Prior to the pure particle based approach, we tried a different approach. One can compute the erosion per voxel: Every particle stores a boolean indicating if it has collided or not. Then for each voxel, all particles that are in the influence radius of that voxel (see 3.2.3) and have collided are tested. For each of these particles, the erosion equations (3.2.2.1 to 3.2.2.3) are evaluated, the changes written to the particles and the changes to the terrain are accumulated. This method works but has some serious disadvantages. First, it requires a kernel thread for each voxel. Because of the sparse structure of the terrain (3.1.4), a lower bound for that is $\mathcal{O}(wh)$ with $w = h = 256$ for a small terrain. With a voxel depth of e.g. 8 this is already much bigger than the number of particles simulated in normal cases. Furthermore, the particles have to be tested as well. Therefore, the run-time of this method also scales with the number of particles. In our experiments, this method is much slower than the particle based approach (10x-100x). Second, the erosion simulation is more inexact. Since the erosion is evaluated at the end positions of a particle, scenarios like an early collision and then a long travel without collisions within one time step are not captured. These arguments have motivated us to concentrate on the particle based approach as described in the previous sections.

3.2.3. Particle storage and acceleration

For each particle, we have to store the following properties:

- the position $\mathbf{p}_i$, three floats
- the velocity $\mathbf{v}_i$, three floats
- the force (pressure and viscosity) $\mathbf{f}_i^{\text{pressure}} + \mathbf{f}_i^{\text{viscosity}}$, three floats
- the density ρ_i, one float
- the sediment s_i, one float

We allocate three buffers of type float4 and two of type float on the GPU and use the following buffer layout:

- buffer 1: xyz = $\mathbf{p}_i$, w = s_i
- buffer 2: xyz = $\mathbf{v}_i$, w = w_i

- buffer 3: xyz = $\mathbf{f}_i$, w = unused
- buffer 4: ρ_i
- buffer 5: Δs_i

For the computation of the density (eq. 3.19), pressure force (eq. 3.21) and viscosity force (eq. 3.22), we have to sum over all particles. Without optimization, this would be a $\mathcal{O}(n^2)$ operation with n being the number of particles. However, because only particles within the core radius h (see 3.2.1) contribute to the sum, we can bring down the computation cost to $\mathcal{O}(nr)$, where r is the maximal number of particles inside the core radius. r can often be expressed as a constant depending on the core radius h, the gas constant k (see eq. 3.20) and the minimal and maximal value of the particle mass m_i.

We achieve this by sorting the particles based on their position. The position of each particle in the grid coordinates is rounded to the next integer representing the voxel that contains this particle. These three integers are then assembled to a single integer key. All particles are sorted according to this key. We use a radixsort implementation from [17]. After that, we build an offset table that maps each key to the index of the first particle with that key, or -1 if there are no particles at that key (see program 5 in the appendix). With the particles sorted by their position, a sum over all particles can be formulated as a loop over only the neighboring particles obtained by the offset table (shown in program 6 in the appendix).

This simple method assumes that the core radius does not exceed the size of one voxel. Particles that are further away are not captured in the 27-neighborhood and are therefore ignored. If one wants to include those as well and support a bigger core radius, it is either possible to increase the search area of the neighborhood, or to downscale the searching grid by e.g. dividing each particle position by 2 (then eight voxels are mapped to one sorting index). The latter approach can sometimes result in a faster simulation since fewer different keys have to be sorted. Therefore, it is used in some test cases.

In summary, the following three kernel calls are needed to perform the whole fluid and erosion simulation:

1. Compute densities ρ_i (eq. 3.19)
2. Compute pressure force $\mathbf{f}_i^{\text{pressure}}$ (eq. 3.21), viscosity force $\mathbf{f}_i^{\text{viscosity}}$ (eq. 3.22); perform sediment diffusion, stored in Δs_i (eq. 3.38); test if a particle is at the surface (eq. 3.39)

3. Apply sediment delta (eq. 3.38), evaporate water (eq. 3.40); advect and collide particles (see 3.2.1.4), perform dissolving (see 3.2.2.2) and deposition (see 3.2.2.3)

3.3. Visualization

The final step of a simulation is to visualize the results. We first describe how to display the terrain and how to deal with the terrain changes introduced by the erosion simulation. Second, rendering techniques for the water particles are presented.

3.3.1. Terrain rendering

We implemented two simple approaches of terrain rendering: marching cubes and a direct raytracer.

3.3.1.1. Marching Cubes

Marching cubes is a method to generate a polygonal mesh from a level set function [48]. It looks at eight neighboring voxels (like it is done in 3.1.2) and builds the triangles from one of the 256 cases created from rotated versions of the 16 base cases (see fig. 3.18). Because the output is a list of triangles, it fits into any existing render engines, both online and offline.

We use the algorithm presented in [48]. It generates the marching cubes surface on-the-fly in a pipeline of shader programs from a volumetric texture. However, we can't directly use this way because our terrain is not available as a 3D texture, but as many fine cells (see 3.1.4). Therefore, we use an OpenCL program that generates the marching cubes mesh for each fine cell and stores them in a vertex buffer. The program consists of three stages:

1. For each voxel in the fine cell, the triangle count is computed.

2. A prefix sum is performed. The list of triangle counts becomes a list of indices.

3. For each voxel, the triangle data is written into a vertex buffer large enough (with DirectX - OpenCL - interop) at the index computed in the previous stage.

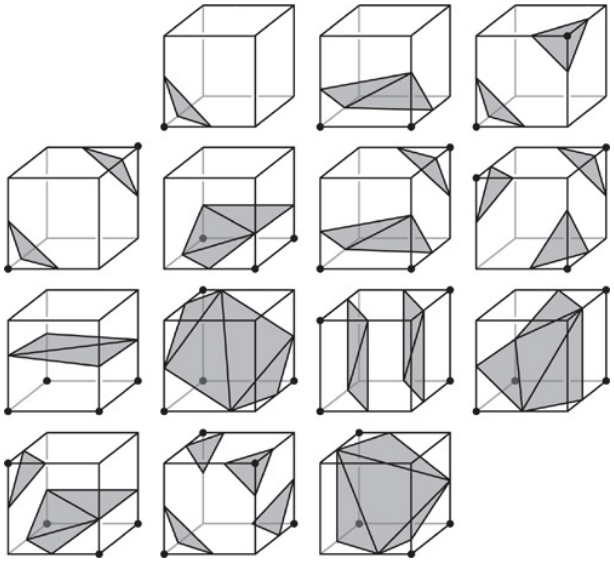


Figure 3.18.: 16 base cases of the marching cubes algorithm (the empty and full case is not shown) [48]

The result is a vertex buffer for each fine cell that is rendered in each cell. Because the creation of these vertex buffers may become a bottleneck, we don't update them every frame, but periodically (e.g. 10 cells every frame).

3.3.1.2. Ray tracing

In the ray tracing technique, we shoot a ray from the camera through every pixel on the screen and test whether and when it hits the terrain [2].

For every step on the ray tracing, the value of the level set function φ is queried at this point (see 3.1.2). If the value is positive, it is inside the terrain and the algorithm terminates. If the value is negative, the position is still outside and the algorithm has to advance along the ray.

The level set function provides an approximation to the shortest distance to the terrain surface. This does not provide a direct solution to the distance needed to travel, but it gives us a lower bound on the step size. To increase stability, we clamp the step size to be in the interval of [0.1,10] for example. If φ becomes positive, the terrain is reached. We then perform some binary search steps (e.g. 5) to increase the precision of the collision position. Although this method results in a speedup, our tracing algorithm is still quite naïve. Better algorithms could also utilize the sparsity of the terrain storage.

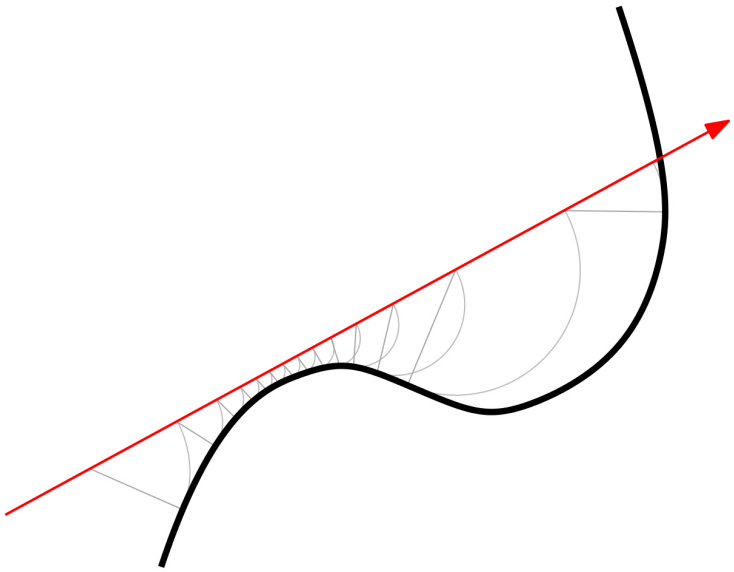


Figure 3.19.: Raytrace the level set

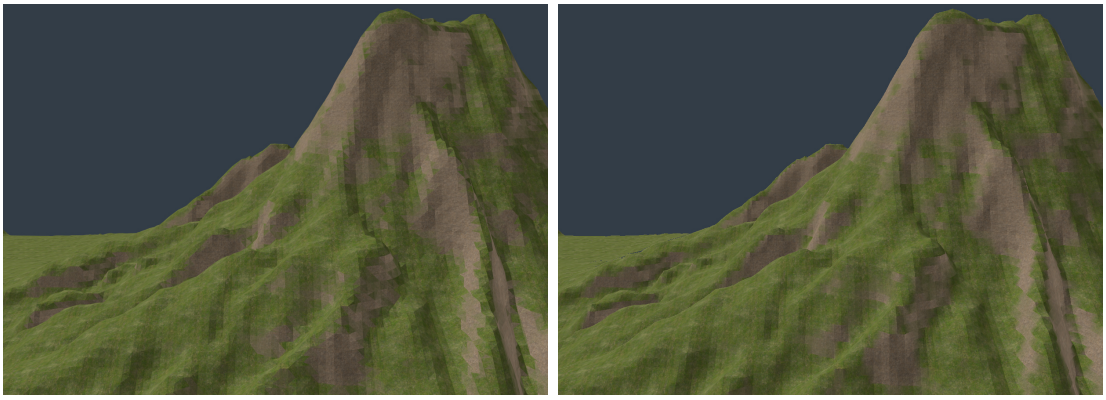
Figure 3.19 shows the ray tracing algorithm. The terrain surface is drawn in black, the ray in red. On each step, the level set gives the shortest distance to the surface (dark gray lines). This distance is then projected on the ray to specify the new step size (gray circular arcs).

In figure 3.20 a comparison between marching cubes and ray tracing is shown. The differences a very subtle, ray tracing produces slightly smoother images at steep corners. The marching cubes approach, however, has a severe disadvantage: the periodic update of the meshes. This leads to gaps between cells and does not scale well with the grid size. Therefore, ray tracing is used most often, although it is slower in most cases.

3.3.1.3. Shading

Shading is the method of assigning colors to pixels. We use the same method for both the marching cubes algorithm and the raytracer. The input to the shading pixel shader is the position in world space $\mathbf{p}$ and the surface normal $\mathbf{n}$.

We use tri-planar texturing [5] to get the texture color at the position $\mathbf{p}$ and blend between a grass texture and a stone texture based on the z-value of the surface normal. The implementation can be seen in program 7 (see appendix).



(a) marching cubes **(b)** ray tracing

Figure 3.20.: Comparison of marching cubes and ray tracing

After the color from the textures is fetched, a simple phong shading is performed to include a single directional light source. Since we can't always rely on the correct orientation of the normal vector, the light source is treated as coming from both sides.

3.3.2. Water rendering

Many sophisticated render techniques have been developed to render a smooth surface based on particles [40, 39, 32, 42, 36].

With respect to the scope of this work, we implemented two water rendering techniques. First, a simple sprite based approach is used. Each particle is rendered as a point sprite (3.21) with the size decreasing with the distance from the camera. The particles are colored based on how much sediment they carry (3.22). To enhance the appearance, the particles can be sorted based on their screen space depth value. This allows us to render the transparent particles correctly and the location of the sediment within the flow is visualized more clearly. The drawback is an increased rendering time. Furthermore, a simple shadow mapping is used to enhance the perception of water particles flying through the air. As a second approach, the water particles are rendered with a technique called "Spreen Space Fluid Rendering with Curvature Flow" (SSF) ([56, 52]). The water particles are rendered into the depth buffer as spheres. The depth information is then blurred to minimize the curvature using a bilateral gaussian filter. Finally, the water surface is reconstructed from the depth information and shaded. A comparison can be seen in fig. 3.23.

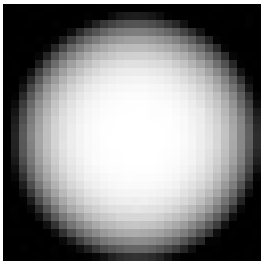
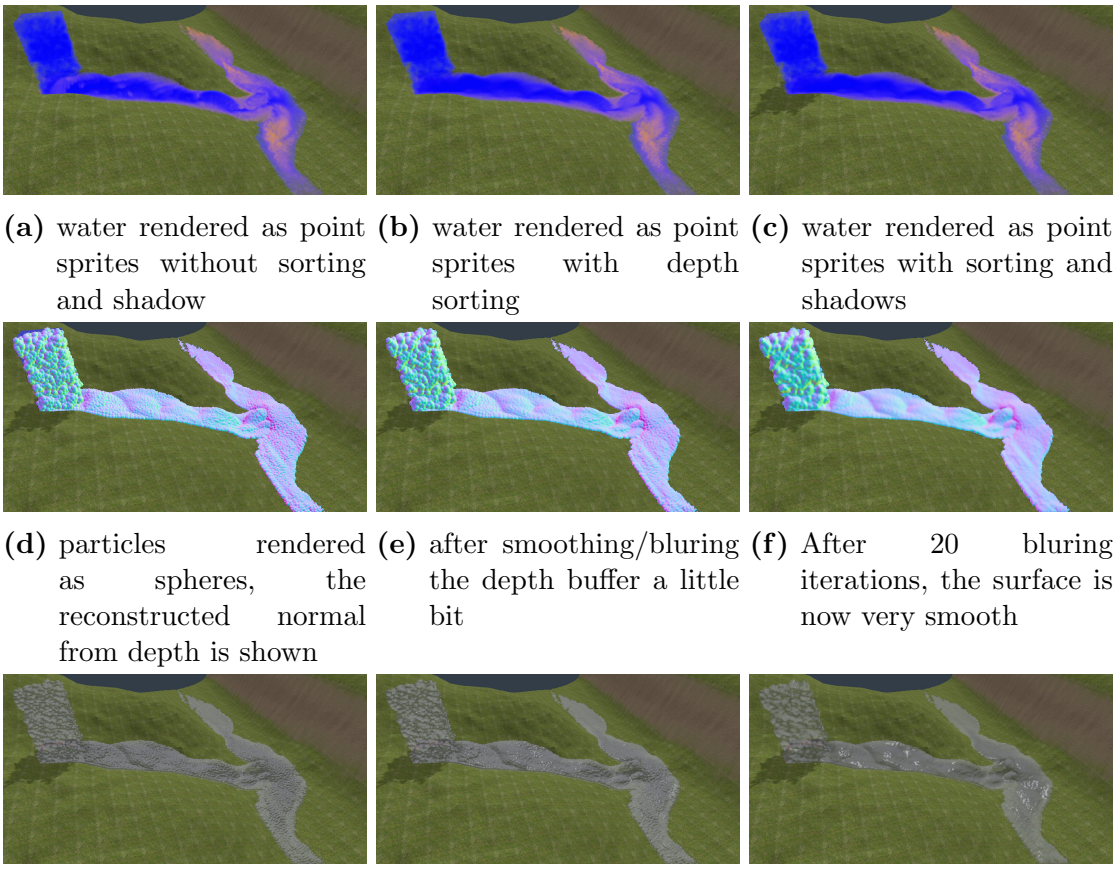


Figure 3.21.: Water sprite texture

Figure 3.22.: Color ramp for shading the particle based on the amount of transported sediment



(a) water rendered as point sprites without sorting and shadow

(b) water rendered as point sprites with depth sorting

(c) water rendered as point sprites with sorting and shadows

(d) particles rendered as spheres, the reconstructed normal from depth is shown

(e) after smoothing/bluring the depth buffer a little bit

(f) After 20 bluring iterations, the surface is now very smooth

(g) **(h)** **(i)**
the above surfaces rendered with Phong shading

Figure 3.23.: Different techniques for rendering the water particles

4. Results

Some test cases are presented in this section that show the effects of the simulation and its parameters. The images can be seen all together on the next pages.

Fig. 4.1 shows the effects of the per-voxel dissolving constant $K_{s,\mathbf{p}_i}$ (see 3.2.2.2). In this test case, water from a square source rains on a slope and erodes the terrain there. The per-voxel dissolving constant varies from 0.01 to 1.0 based on a perlin noise. You can see areas with a strong erosion (high $K_{s,\mathbf{p}_i}$) and almost no erosion (low $K_{s,\mathbf{p}_i}$) in the figure.

The next series of images in 4.2 evaluates the influence of the ratio $r = K_{s,n}/K_{s,t}$ (see 3.2.2.2). The original terrain is shown as a white shape. If only normal movement erodes the terrain 4.2a, the first sediment is dissolved at the bottom where the water crushed frontally against the terrain. The silhouette of the slope, however, is not changed. In 4.2 only tangential movement dissolves sediment instead. As you can see, the shape of the slope is changed while at the bottom, sediment is deposited again.

We now analyse the performance of the proposed algorithms. The measurements are shown in table 4.1. The simulation was executed on a desktop computer with an Intel Xeon CPU E5-1650 v2 3.50GHz (6 cores), 32GB RAM and an AMD FirePro W9100 GPU with 16VRAM running Windows 8.1 Enterprise. The screen resolution was set to 1440x900. Each time specification shows the time needed for that operation per frame or per time step. To get accurate results, we enclosed every part of the simulation with a call to `clFlush`.

The first test case is a slope with pillars sticking out 4.3. Then we analyze the influence of the grid resolution in 4.4. The grid resolution mainly influences the rendering time and particle sorting time, the force computations are mostly dependent on the particle count. The next test is a smooth wall 4.5. In the close look of the river, you can see that the particles at the bottom carry sediment while the top ones don't.

It can be seen from the table 4.1 that the whole fluid simulation step only takes between 10 to 15 ms in normal cases, longer only in scenarios with many particles. The most time per frame is spent in the rendering because of the terrain ray tracing or marching cubes updates. Therefore we execute multiple simulation steps per rendered frames, typically between 5 and 10. In total we achieve between 5 to 20 frames per second. The speed is highly dependent on the terrain size and particle count.

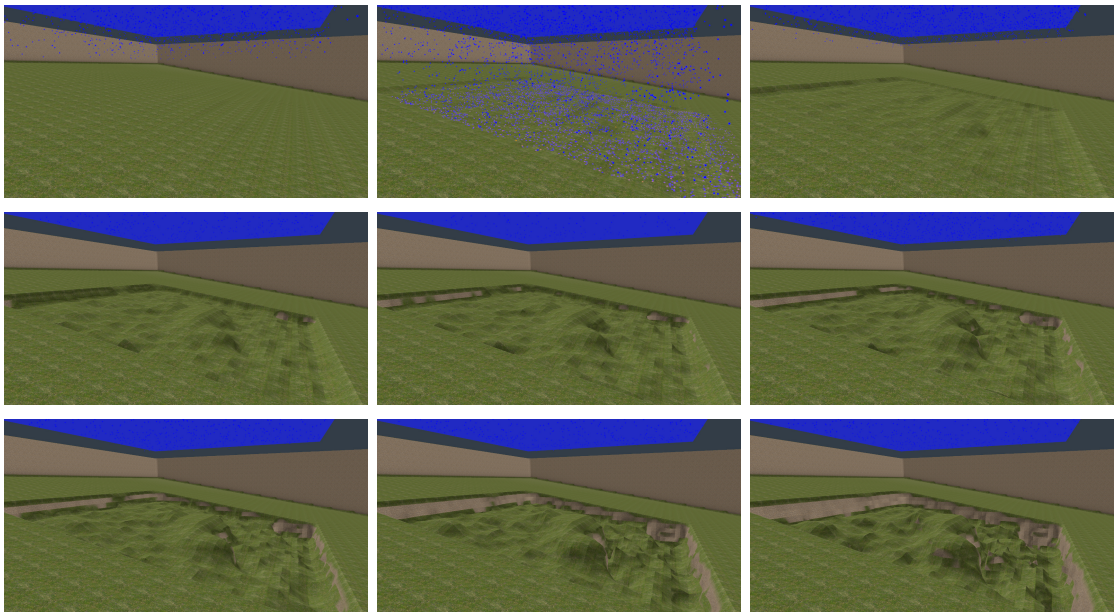
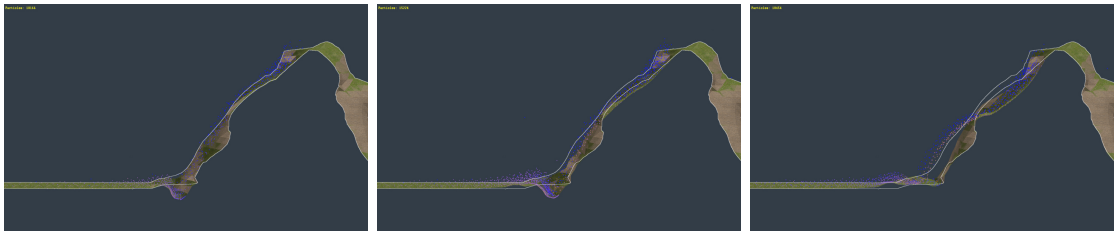


Figure 4.1.: The effects of $K_{s,\mathbf{p}_i}$ (see 3.2.2.2) as the time evolves



(a) $K_{s,n} = 0.1$, $K_{s_t} = 0$ **(b)** $K_{s,n} = 0.05$, $K_{s_t} = 0.05$ **(c)** $K_{s,n} = 0$, $K_{s_t} = 0.1$

Figure 4.2.: Different settings for the ratio $r = K_{s,n}/K_{s,t}$ (see 3.2.2.2)

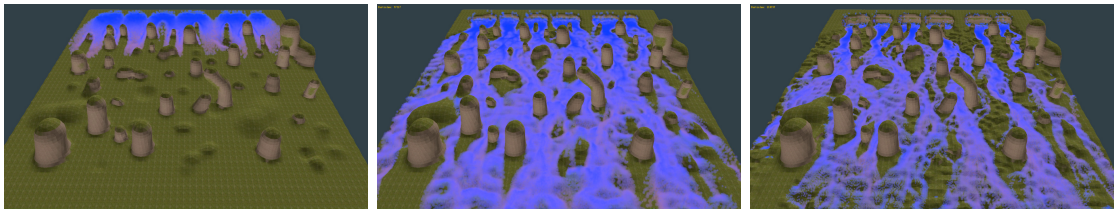
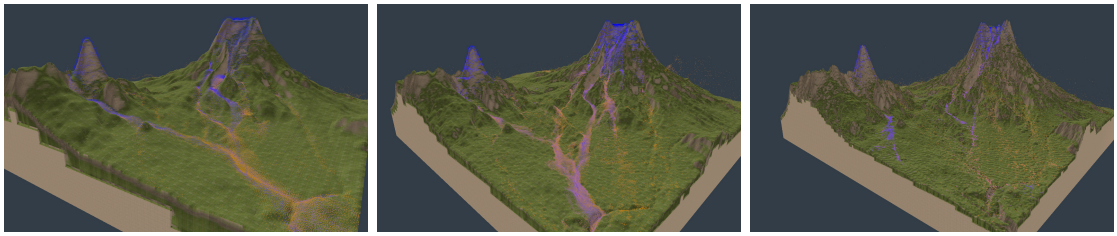
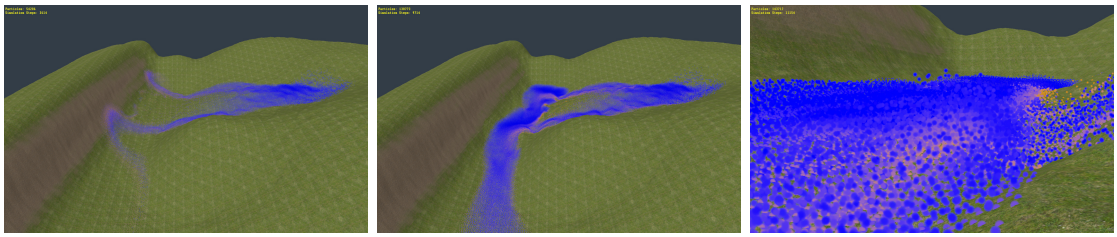


Figure 4.3.: A slope with stone pillars, from the beginning to the end of the simulation



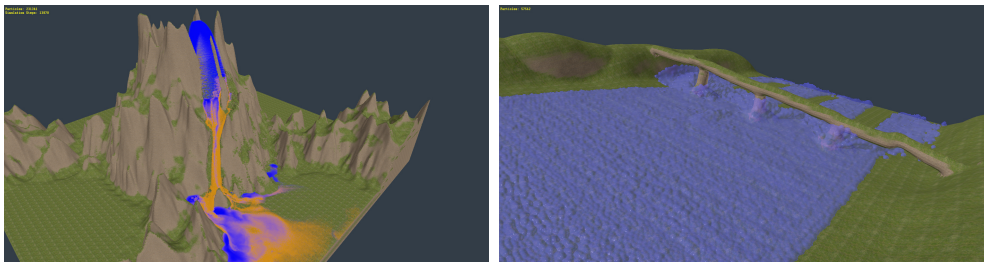
(a) 256*256*128 **(b)** 512*512*256 **(c)** 1024*1024*1024

Figure 4.4.: Two hills with different resolutions



(a) At the beginning **(b)** After a few minutes **(c)** A close look

Figure 4.5.: A smooth wall



(a) Volcano **(b)** Bridge over a slope

Figure 4.6.: Other test cases

Test case			Rendering			Fluid simulation				
Name	Grid size	# particles	Marching Cubes	Raytracing	particles	Density	Forces	Collision+Erosion	Sorting	Raining
Slope with pillars 4.3	128^3	30000	40 ms	31.5 ms	2.2 ms	0.5 ms	1.4 ms	1.2 ms	2 ms	1.9 ms
Two Hills 4.4a	$256 * 256 * 128$	50000	17 ms	45 ms	1 ms	1.3 ms	4.3 ms	0.8 ms	3 ms	1.3 ms
Two Hills 4.4b	$512 * 512 * 256$	100000	72 ms	73 ms	1.2 ms	1.4 ms	4.9 ms	1.5 ms	4 ms	1.9 ms
Two Hills 4.4c	$1024 * 1024 * 512$	200000	- (too big)	134 ms	1.3 ms	4.1 ms	15 ms	14.5 ms	17.2 ms	3.3 ms
Smooth Wall 4.5	$256 * 256 * 128$	140000	17.5 ms	38 ms	1.2 ms	1.2 ms	4.8 ms	0.8 ms	2.0 ms	1.3 ms
Volcano 4.6a	$512 * 512 * 256$	150000	143 ms	140 ms	9 ms (with sorting)	1.7	6.9	1.9 ms	4.1 ms	0.9 ms
Bridge 4.6b	$256 * 256 * 256$	50000	20 ms	50 ms	3 ms (with SSF, 3.3.2)	0.3 ms	1.5 ms	1.3 ms	5 ms	1.5 ms

Table 4.1.: Performance analysis of several test cases

5. Conclusion and future work

In this thesis we have presented methods for an interactive simulation of hydraulic erosion. We used a level set to represent the terrain and store it in a sparse two-layer hierarchy of grids. The fluid simulation was based on Smoothed Particle Hydrodynamics with a special collision handling with the terrain. For the erosion simulation, equations from erosion on heightmaps and light diffusion were used and adopted. To render the outcomes, raytracing, marching cubes and particle sprite techniques were presented.

The simulation is able to produce realistic looking results within seconds. We are quite content with the physical accuracy of the fluid model, but future work has to include the validation of the proposed erosion equations for their accuracy. Furthermore, forces within the terrain are not yet simulated. This extension would allow us to detect e.g. when the connection of an overhang becomes too weak and it crushes down, or when a landslip is triggered. Also the coupling of solid objects with the fluid simulation would enable us to simulate stones and tree branches that are carried with the water and modify the terrain. These topics are part of future work.

A. Algorithms

Program 1 Helper function that queries the values of the eight surrounding voxels, see 3.1.2

```
float8 getNeighbors(int3 pos) {
    return (float8) (
        getLevelset(pos),
        getLevelset(pos + (int3)(1,0,0)),
        getLevelset(pos + (int3)(0,1,0)),
        getLevelset(pos + (int3)(1,1,0)),
        getLevelset(pos + (int3)(0,0,1)),
        getLevelset(pos + (int3)(1,0,1)),
        getLevelset(pos + (int3)(0,1,1)),
        getLevelset(pos + (int3)(1,1,1))
    );
}
```

Program 2 Performs trilinear interpolation to query φ at any point in space, see 3.1.2

```
float L(float3 pos) {
    int3 voxel = convert_int3(pos); //get integral and fractional part
    float3 frac = pos - convert_float3(voxel);
    float8 neighbors = getNeighbors(voxel); //query neighbors
    float4 ix = mix(neighbors.s0246, neighbors.s1357, frac.x);
    float2 ixy = mix(ix.s02, ix.s13, frac.y);
    return mix(ixy.s0, ixy.s1, frac.z);
}
```

Program 3 Computes the normal **n** at a given point in space, see 3.1.2

```
float3 N(float3 pos) {
    int3 voxel = convert_int3(pos); //get integral and fractional part
    float3 frac = pos - convert_float3(voxel);
    float8 neighbors = getNeighbors(voxel); //query neighbors
    float4 ix = mix(neighbors.s0246, neighbors.s1357, frac.x);
    float4 iy = mix(neighbors.s0145, neighbors.s2367, frac.y);
    float2 ixy = mix(ix.s02, ix.s13, frac.y);
    float2 ixz = mix(ix.s01, ix.s23, frac.z);
    float2 iyz = mix(iy.s01, iy.s23, frac.z);
    float dx = iyz.s1 - iyz.s0;
    float dy = ixz.s1 - ixz.s0;
    float dz = ixy.s1 - ixy.s0;
    return (float3)(dx, dy, dz);
}
```

Algorithm 1 Particle advection and collision with the terrain, see 3.2.1.4

1: $\mathbf{d} = t\mathbf{v}$ // the distance the particle travels
2: $t_{\max} = ||\mathbf{d}||$ // normalize the direction
3: $\mathbf{d}' = \mathbf{d}/t_{\max}$ // so that the step size is independent of the speed
4: $\hat{\mathbf{p}} = \mathbf{p}$
5: **while** $t_{\max} > \epsilon$ **do** // e.g. $\epsilon = 0.00001$
6: $t' = \min\{t_{\max}, t_{\text{step1}}\}$ // e.g. $t_{\text{step2}} = 0.01$
7: $\hat{\mathbf{p}} \mathrel{+}= t'\mathbf{d}'$ // move a small step forward
8: $t_{\max} \mathrel{-}= t'$
9: **if** $L(\hat{\mathbf{p}}) > 0$ **then** // collision detected
10: $\mathbf{n} = \mathbf{N}(\hat{\mathbf{p}})$ // query the normal
11: **repeat**
12: $\hat{\mathbf{p}} \mathrel{+}= t_{\text{step2}}\,\mathbf{n}$ // move outside, e.g. $t_{\text{step2}} = 0.025$
13: **until** $L(\hat{\mathbf{p}}) \leq 0$
14: $\mathbf{v} = \mathbf{v} - \mathbf{n}(\mathbf{v} \cdot \mathbf{normal})$ // adjust velocity, normal component is set to zero
15: $\hat{\mathbf{d}} = t\mathbf{v}$
16: $t_{\max} \mathrel{*}= ||\hat{\mathbf{d}}||/||\mathbf{d}||$ // adjust $t_{\max}$
17: $\mathbf{d}' = \hat{\mathbf{d}}/||\hat{\mathbf{d}}||$
18: **end if**
19: **end while**

Algorithm 2 Rain sampling from a water density texture, see 3.2.1.5

```
1: function RandFloat(__global ulong* seed)
2:     // Taken from java.util.Random of Java 7
3:     *seed = (*seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)
4:     return (int)(*seed >> 24) / ((float)(1 ¡¡ 24));
5: end function
6: function SampleRain
7:     // Kernel call, one thread per particle
8:     Input: rain density texture d, the random seed
9:     Output: particle position p
10:    repeat
11:        x = RandFloat(seed)
12:        y = RandFloat(seed)
13:        τ = d(x, y)
14:        if τ > RandFloat(seed) then // attempt to sample particle at this position
15:            find height of the terrain at (x, y), store in z
16:            store particle position p = (x, y, z)
17:            return
18:        end if
19:    until max trials reached
20:    unable to sample particle, place particle outside the grid so it gets deleted
21:    return
22: end function
```

Algorithm 3 Terrain update of the eight surrounding voxels, see 3.2.2.4

```
1: Input: the change of sediment Δs
2: Input: the position of the particle p_i
3: x = floor(p_i)                          // round p_i down to the next integer
4: φ(x) −= Δs
5: φ(x + (1, 0, 0)^T) −= Δs
6: φ(x + (0, 1, 0)^T) −= Δs
7: φ(x + (1, 1, 0)^T) −= Δs
8: φ(x + (0, 0, 1)^T) −= Δs
9: φ(x + (1, 0, 1)^T) −= Δs
10: φ(x + (0, 1, 1)^T) −= Δs
11: φ(x + (1, 1, 1)^T) −= Δs
```

Program 4 AtomicAdd on floats for OpenCL 1.2, see 3.2.2.4

```
void AtomicAdd(volatile __global float *source, const float operand)
{
    if (source == 0) return;
    union {
        unsigned int intVal;
        float floatVal;
    } newVal;
    union {
        unsigned int intVal;
        float floatVal;
    } prevVal;
    do {
        prevVal.floatVal = *source;
        newVal.floatVal = prevVal.floatVal + operand;
    } while (atomic_cmpxchg(
            (volatile __global unsigned int *)source,
            prevVal.intVal, newVal.intVal) != prevVal.intVal);
}
```

Program 5 Build the offset table that maps keys to particle indices, see 3.2.3

```
__kernel void FillOffsets(__global uint* keys, __global uint* offsets)
{
    //One thread per particle
    //The offset buffer is prefilled with -1 (or 0xffffffff)
    const int idx = get_global_id(0);
    if (idx == 0) {
        uint key = keys[idx];
        offsets[key] = idx;
    }
    else {
        uint key = keys[idx];
        if (key != keys[idx - 1]) {
            offsets[key] = idx;
        }
    }
}
```

Program 6 Summing over all particles in the neighborhood using the offset table from the particle sorting, see 3.2.3

```
//posi contains the position of the current particle.
//NEIGHBORS is an array of int3 with the
//  relative positions of the 27-neighborhood.
for (int i = 0; i < NEIGHBOR_COUNT; ++i) {
    int3 posi2 = posi + NEIGHBORS[i].xyz;
    if (!VOXEL_VALID(posi2)) continue; //test if we are outside
    uint key = getVoxelID(posi2);
    uint offset = offsets[key];
    if (offset == 0xffffffff) continue; //no particle in that voxel
    for (int j = offset; j < particleCount; ++j) {
        float4 posj = positions[j];
        if (getVoxelID(convert_int3(posj.xyz)) != key) break;
        //Do something with the current particle with index j
    }
}
```

Program 7 Texturing and shading of the terrain, see 3.3.1.3

```
//compute the blending parameter for the tri-planar texturing
float3 blending = abs(n);
float b = blending.x + blending.y + blending.z;
blending /= b;
//compute the blending parameter between the two textures
float blending2 = 1-smoothstep(0.4f, 0.6f, abs(n.z));
//Fetch and blend texture values
float3 xaxis = lerp(
    gGrassTexture.Sample(samLinear, p.yz * gTextureScale).rgb,
    gStoneTexture.Sample(samLinear, p.yz * gTextureScale).rgb,
    blending2);
float3 yaxis = lerp(
    gGrassTexture.Sample(samLinear, p.xz * gTextureScale).rgb,
    gStoneTexture.Sample(samLinear, p.xz * gTextureScale).rgb,
    blending2);
float3 zaxis = lerp(
    gGrassTexture.Sample(samLinear, p.xy * gTextureScale).rgb,
    gStoneTexture.Sample(samLinear, p.xy * gTextureScale).rgb,
    blending2);
//assemble final texture color
color.rgb = xaxis * blending.x + yaxis * blending.y + zaxis * blending.z;
//perform simple lighting
color.rgb *= gAmbientLight
          + gDiffuseLight * abs(dot(n, -gLightDir));
```

Bibliography

[1] Amit Patel. Polygonal map generation for games, 2010. Retrieved 11/25/2015, from `http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/`.

[2] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[3] Audrey Maheu, Daniel Caissie, André St-Hilaire, Nassir El-Jabi. River evaporation and corresponding heat fluxes in forested catchments. *Hydrological Processes*, 28(23): 5725–5738, 2014. ISSN 08856087. doi: 10.1002/hyp.10071.

[4] Bedrich Beneš. Real-time erosion using shallow water simulation. *4th Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS"*, 2007.

[5] Brent Owens. Use tri-planar texture mapping for better terrain, 2014. Retrieved 04/01/2016, from `http://gamedevelopment.tutsplus.com/articles/use-tri-planar-texture-mapping-for-better-terrain--gamedev-13821`.

[6] Carsten Dachsbacher. *Interactive Terrain Rendering*. dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2006.

[7] Chohong Min. On reinitializing level set functions. *Journal of Computational Physics*, 229(8):2764–2772, 2010. ISSN 00219991. doi: 10.1016/j.jcp.2009.12.032.

[8] Christof Rezk Salama. Volume rendering techniques for general purpose graphics hardware. *den Nominierungsausschuss*, page 123, 2001.

[9] Colin Braley, Adrian Sandu. Fluid simulation for computer graphics: A tutorial in grid based and particle based methods. *Virginia Tech, Blacksburg*, 2010.

[10] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22, 2009.

[11] Cyril Crassin, Simon Green. Octree-based sparse voxelization using the gpu hardware rasterizer. In *OpenGL Insights*. CRC Press, Patrick Cozzi and Christophe Riccio, 2012.

[12] Danping Peng, Barry Merriman, Stanley Osher, Hongkai Zhao, Myungjoo Kang. A pde-based fast local level set method. *Journal of Computational Physics*, 155(2): 410–438, 1999. ISSN 00219991. doi: 10.1006/jcph.1999.6345.

[13] Filip Strugar. Continuous distance-dependent level of detail for rendering heightmaps. *Journal of Graphics, GPU, and Game Tools*, 14(4):57–74, 2009. ISSN 2151-237X. doi: 10.1080/2151237X.2009.10129287.

[14] Flora Ponjou Tasse, Arnaud Emilien, Marie-Paule Cani, Stefanie Hahmann, Adrien Bernhardt. First person sketch-based terrain editing. In *Proceedings of the 2014 Graphics Interface Conference*, pages 217–224, 2014.

[15] Florian Ferstl, Ryoichi Ando, Chris Wojtan, Rüdiger Westermann, Nils Thuerey. Narrow band FLIP for liquid simulations. *Computer Graphics Forum (Eurographics)*, 35(2):to appear, 2016.

[16] Frank Losasso, Frédéric Gibou, Ron Fedkiw. Simulating water and smoke with an octree data structure. In *ACM Transactions on Graphics (TOG)*, volume 23 (3), pages 457–462, 2004.

[17] Geist Software Labs. libcl: Parallel algorithm library. Retrieved 03/28/2016, from `http://www.libcl.org/`.

[18] Gonçalo Nuno Paiva Amador. *Real-time 3D rendering of water using CUDA*. PhD thesis, Univeristy of Beira Interior, Covilha, Portugal, 2009.

[19] H. L. Penman. Natural evaporation from open water, bare soil and grass. In *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, volume 193 (1032), pages 102–145, 1948.

[20] Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie, Eric Galin. Feature based terrain generation using diffusion equation. *Computer Graphics Forum*, 29(7): 2179–2186, 2010. ISSN 01677055. doi: 10.1111/j.1467-8659.2010.01806.x.

[21] Igor Suhorukov. Opencl 1.1: Atomic operations on floating point values, 2011. Retrieved 03/28/2016, from `http://suhorukov.blogspot.de/2011/12/opencl-11-atomic-operations-on-floating.html`.

[22] J.A. Sethian. A fast marching level set method for monotonically advancing fronts. volume 93 (4), pages 1591–1595, 1996.

[23] Jacob Olsen. Realtime procedural terrain generation, 10/31/2004.

[24] Jakob Andreas Bærentzen. Octree-based volume sculpting. *Vis98-IEEE Visualization 1998*, 1998.

[25] Jean-David Genevaux, Eric Galin, Eric Guerin, Adrien Peytavie, Bedrich Benes. Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics (TOG)*, 32(4):143, 2013.

[26] Jeff Dicker. *Fast Marching Methods and Level Set Methods: An Implementation.* dissertation, University of British Columbia Okanagan Campus, 2006.

[27] Jens Kruger, Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, 2003.

[28] JG Zhou. A lattice boltzmann model for the shallow water equations. In *Computer methods in applied mechanics and engineering*, volume 191 (32), pages 3527–3539. Elsevier, 2002.

[29] Jonas Larsen, Thomas Greve Kristensen. An overview of the implementation of level set methods, including the use of the narrow band method, 2005. Retrieved 03/28/2016, from `http://cs.au.dk/~tgk/courses/LevelSets/`.

[30] Jonathon Doran, Ian Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, 2010. ISSN 1943-068X. doi: 10.1109/TCIAIG.2010.2049020.

[31] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, 1999.

[32] Keenan Crane, Ignacio Llamas, Sarah Tariq. Real-time simulation and rendering of 3d fluids. In Hubert Nguyen, editor, *GPU gems 3*, pages 633–675. Addison-Wesley, Upper Saddle River, NJ, 2008. ISBN 0321515269.

[33] Kristof Römisch, Peter Møller-Nielsen. *Sparse Voxel Octree Ray Tracing on the GPU.* PhD thesis, Aarhus Universitet, Datalogisk Institut, 2009.

[34] M Ali, G Sterk, M Seeger, MP Boersema, P Peters et al. Effect of hydraulic parameters on sediment transport capacity in overland flow over erodible beds. In *Hydrology and Earth System Sciences Discussions*, volume 8 (4), pages 6939–6965, 2011.

[35] MA Nearing, GR Foster, LJ Lane, SC Finkner. A process-based soil erosion model for usda-water erosion prediction project technology. *Transactions of the ASAE*, 32 (5):1587–1593, 1989.

[36] Marcus Vesterlund. Simulation and rendering of a viscous fluid using smoothed particle hydrodynamics. *Master's thesis, Umea University, Sweden*, 2004.

[37] Mark J Harris, Anselmo Lastra. Real-time cloud rendering. In *Computer Graphics Forum*, volume 20 (3), pages 76–85, 2001.

[38] Matt Pharr, Greg Humphreys. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2010.

[39] Matthias Müller, David Charypar, Marku Gross. Particle-based fluid simulation for interactive applications. *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159, 2003.

[40] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, Markus Gross. Surface splatting. In Lynn Pocock, editor, *the 28th annual conference*, pages 371–378, 2001. doi: 10.1145/383259.383300.

[41] Mikola Lysenko. Meshing in a minecraft game, 2012. Retrieved 04/03/2016, from `http://0fps.net/2012/06/30/meshing-in-a-minecraft-game/`.

[42] Nuttapong Chentanez, Matthias Müller. Real-time eulerian water simulation using a restricted tall cell grid. In *ACM Transactions on Graphics (TOG)*, volume 30 (4), page 82, 2011.

[43] Nuttapong Chentanez, Matthias Muller. Coupling 3d eulerian, heightfield and particle methods for interactive simulation of large scale liquid phenomena. *Visualization and Computer Graphics, IEEE Transactions on*, 21(10):1116–1128, 2015.

[44] Paul Lemke, Helene Roht. Rendering von Landschaften: Seminar Echtzeit-Rendering SS 2008, Universität Koblenz Landau. 2008.

[45] Philippe Blasi, Bertrand Saec, Christophe Schlick. A rendering algorithm for discrete volume density objects. In *Computer Graphics Forum*, volume 12 (3), pages 201–210, 1993.

[46] PY Julien, DB Simons. Sediment transport capacity of overland flow. *Transactions of the ASAE*, 28(3):755–0762, 1985.

[47] Robert A Drebin, Loren Carpenter, Pat Hanrahan. Volume rendering. In *ACM Siggraph Computer Graphics*, volume 22 (4), pages 65–74, 1988.

[48] Ryan Geiss. Generating complex procedural terrains using the gpu. In Hubert Nguyen, editor, *GPU gems 3*, pages 7–38. Addison-Wesley, Upper Saddle River, NJ, 2008. ISBN 0321515269.

[49] S. Laine, T. Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011. ISSN 1077-2626. doi: 10.1109/TVCG.2010.240.

[50] Samuli Laine, Tero Karras. Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1048–1059, 2011.

[51] Shiyi Chen, Gary D Doolen. Lattice boltzmann method for fluid flows. In *Annual review of fluid mechanics*, volume 30 (1), pages 329–364. Annual Reviews 4139 El Camino Way, PO Box 10139, Palo Alto, CA 94303-0139, USA, 1998.

[52] Simon Green. Screen space fluid rendering for games. In *Proceedings for the Game Developers Conference*, 2010.

[53] Stanley Osher, Ronald P Fedkiw. Level set methods: an overview and some recent results. *Journal of Computational Physics*, 169(2):463–502, 2001. ISSN 00219991.

[54] Stefan Auer. Realtime particle-based fluid simulation. *Computer Graphics and Visualization*, 2009.

[55] Teong Joo Ong, Ryan Saunders, John Keyser, John J. Leggett. Terrain generation using genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1463–1470, 2005.

[56] Wladimir J van der Laan, Simon Green, Miguel Sainz. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98, 2009.

[57] Xing Mei, Philippe Decaudin, Bao-Gang Hu. Fast hydraulic erosion simulation and visualization on gpu. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pages 47–56, 2007. doi: 10.1109/PG.2007.15.

[58] Y Yu, L Shi. Visual smoke simulation with adaptive octree refinement. In *Proceedings of IASTED International Conference on Computer Graphics and Imaging, Kauai, Hawaii, USA*, pages 13–19, 2004.