CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

Interactive Erosion

Intuitive User Interface for Construction and Simulation of Virtual Terrains

David Hrusa

May 2018

Supervisor: Bedrich Benes PhD.

Acknowledgement / Declaration

I would like to thank professor Benes for proposing this engaging topic and for his willingness to supervise my thesis remotely. I would also like to thank Ing. Sloup for supervising the first section of the implementation and providing council even beyond his obligations.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date 18. 5. 2018

.....

Abstrakt / Abstract

Představujeme uživatelskou aplikaci pro simulaci interaktivní eroze, jež aplikuje její tři základní typy: termální, kinetickou a hydraulickou. Je postavena na principu vrstvové reprezentace a umožňuje definovat a spravovat vícero materiálů a ukládání terénů do souboru.

Klíčová slova: úprava, nástroj, terén, eroze, simulace, interaktivní, kinetická, hydarulická;

Překlad titulu: Interaktivní eroze (Intuitivní uživatelské rozhraní pro konstrukci a simulaci virtuálních terénů)

We present a user oriented interactive erosion application employing three basic erosion types: thermal, kinetic and hydraulic. It operates on a layered terrain representation and allows for definition and management of multiple materials and saving and loading of data via image files.

Keywords: editing, tool, terrain, erosion, simulation, interactive, kinetic, hydarulic;

Contents /

Tables / Figures

Chapter 1
Introduction

This application primarily sets out to design a user-friendly interface for people not intricately familiarized with the study of erosion simulation. While none of the erosion techniques are unique, it is notable that certain standardized GPU data exchange processes have been slightly modified.

1.1 Goals

The goal of this project was to implement an application that would deliver an intuitive way of handling terrain height maps and applying erosion algorithms to them. Such an application should consist of a user interface that renders the current state of the terrain, provides the user with a selection tool and lets them apply a chosen form of erosion to the world. A natural requirement is also the inclusion of some sort of camera navigation within the 3D space.

1.2 Design Overview

The core design principle employed in our solution is the strict differentiation between frontend and backend of the application. With increasing terrain sizes the time conducting simulation can quickly reach unresponsive levels as the complexity of each step rises with $O(n^2)$. A user unfamiliar with the internal workings of the application may for instance get the impression that they are handling it inappropriately. We opted to not include any research references to a multi-threaded frontend-backend application design as we understand it to be a relatively well understood principle.

The inclusion of GPU parallelism in this scenario is well justified as the vast majority of erosion algorithms may be broken down into per-cell computations. A user oriented application requires a convenient access to data manipulation. We will include a mix of custom UI systems and a third-party UI library to deliver a smooth experience.

1.3 Paper Overview

In the firs part of this thesis we explore the data representation possibilities and established techniques for simulating terrain erosion on a theoretical level. Namely we explore thermal weathering, a water simulation model and two water based erosion techniques utilizing it. Afterwards, we list some popular existing tools and finally discuss the principle of the 3D rendering pipeline. In the next section we cover the layout of our application, discuss rendering, simulation structure and finally the implementation of the aforementioned erosion techniques. In the last chapter we conduct a user testing and evaluate the feedback.

Chapter 2
State of the Art

Before constructing a simulation system of our own it is necessary to take a step back and take note of the key elements which current entertainment and research implementations focus on. The simulation of complex natural phenomena is presented with a number of obstacles when striving for accuracy, however modern parallel computing capabilities and pipelines seem almost crafted for the task once the formulation of the problem is processed and adjusted. Erosion simulation models generally recognize two main families of erosion effects which applied together provide the most satisfactory results. Each of the techniques emulates a different source of displacement within the natural environment.

2.1 Discretization

The strongest boundary for realism is the necessary discretization of space and it's properties. The current hardware lacks both the memory and computational speeds to accurately simulate every single particle of material.

Voxel based discretization is a popular technique in systems which focus on intricate rock formations involving complex phenomena such as cave systems and overhangs in general. [1] These generally fall into two major categories, so called euclidean voxel grids with a fixed size cubic voxel as their base unit. Certain implementations also experiment with irregular grid systems. A potential benefit to this might be the reduction of aliasing and other emergent properties caused by a regular data representation, while the counter effect is the increased complexity of any algorithm attempting to traverse such non-uniform space. [2] presents and application employing such a system. The obvious drawback of any voxel based technique is that the data set remains relatively massive with a complexity rising with $O(N^3)$.

An alternative approach to simulating terrain, especially at larger scale is utilized by [3] involving terrain layers. Instead of encoding a three dimensional grid with large amounts of homogeneous cells on top of one another, we may use a collection of flat two dimensional grids which store the amount of material in each of them. This representation forces us to assume that the sediment has homogeneous properties all the way through, which in most land masses will generally be correct.

2.2 Material Based Erosion

The first major group of erosion algorithms originates from the internal processes within the studied material. Prime examples of this would be thermal erosion[4], slippage and mass wasting. These methods generally have a straightforward implementation depending on how the terrain data is being represented and expand towards realism by increasing the defining characteristics of each cell, such as calculating ground water levels to better determine stability. [5] The common denominator in these approaches is the effect of gravity being the main driver behind displacement.

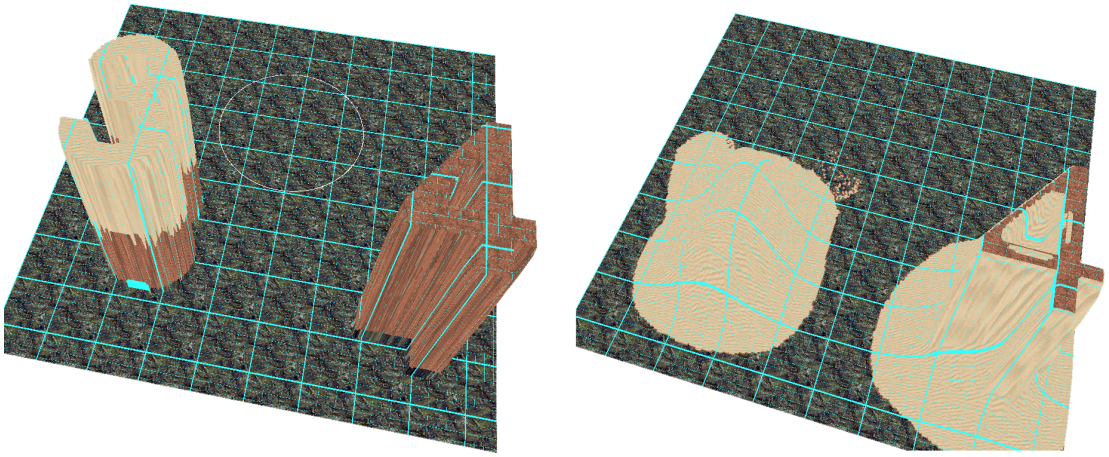

Figure 2.1. Eroding two materials with different thermal rate.

2.2.1 Thermal Erosion

The general task of computing erosion under any model constitutes computing the volumes of entering and leaving material on each simulation step. This may be expressed through a straightforward differential equation[6]:

$$Q_{in} - Q_{out} = \frac{\delta S}{\delta t}$$

where Q_{in} and Q_{out} respectively are the entering and leaving volumes. S being the volume inside said cell and t the time interval of a step. Depending on the hardness of the material the rate of this process may vary which is denoted through a thermal erosion constant which the user is allowed to modify. Additionally each material has it's own talus angle, which is the angle of a slope under which the material will become internally stable and no longer slip. Musgrave proposes that even material properties such as erosion rate or the talus angle could be non-uniformly calculated in a map to increase simulation precision. [7]

We will utilize the thermal erosion algorithm as described in [6]. Denoting the height of the eroded element by h and all the surrounding neighbor elements through indexes $h_i = \{1, 2, ..., 8\}$. A significant value is the maximal difference of heights between h and any lower h_i denoted as H. With the area of the element being a and m_t being the mask determining the strength of the operation the resultant volume to be moved $\Delta S = a * m_t * H/2$. The division by two ensures stability of the entire algorithm. If more volume were to be transferred the mass could begin to oscillate with each step.

Once the transported volume is determined it is subsequently proportionally distributed among all lower lying neighbors fulfilling the talus angle limiting condition. Let's denote the collection as $A = \{h_i, h - h_i > 0 \wedge (h - h_i)/d > tg^{-1}(\alpha), i = 1, ..., 8\}$ where d denotes the distance between the two cells and α is the aforementioned talus angle. A is essentially the list of neighboring cells without the cells which are not low enough to receive any material. Then the expression for finding the amount of material entering a neighbor cell with the index i is as follows:

$$\Delta S_i = \Delta S \frac{h_i}{\sum_{\forall h_k \in A} h_k}$$

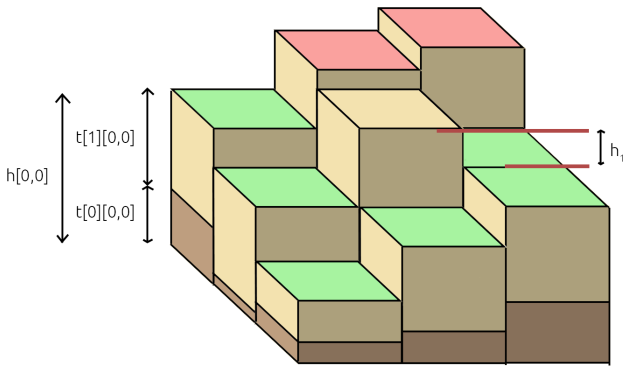

Figure 2.2. Thermal erosion - green cells will receive material, red will not.

The update itself proceeds by subtracting the ΔS from each respective cell and adding up ΔS_i values from each direction to each $i-th$ element. Coordinates outside of the boundaries are ignored. They do not receive any material and are not listed in A. We can clearly see that for each cell ΔS_i add up to ΔS and therefore the total mass of the material in the world is preserved with each step.

2.3 Medium Based Erosion

The second group of erosion algorithms is centered around a medium influencing our material from the outside. Any water or air based erosion model would belong into this category. Executions of these systems vary greatly between various projects, as there exists a plethora of ways to approach both the simulation of fluid movement and a number of ways the material may become suspended in it. We will be describing two hydraulic methods working in parallel.

The most general principle under which the logic of medium based material transportation operates is as follows. A medium based algorithm consists of an erosion and a deposition function. In each simulation step some amount of the material is either dissolved in the medium or deposited as is governed through these relationships. Subsequently, the dissolved material is transported with the flow of the medium. This is how medium based erosion fits into the the previously described general erosion task.

2.3.1 Fluid Discretization

Due to the nature of computational hardware certain degree of discretization is necessary when working with fluids. On an atomic level the flow of liquid media is the result of kinetic impacts between individual particles. At macroscopic levels this behavior has been redefined as a scalar value know as pressure. Due to how hard it is to compress liquids under normal circumstances simulations often operate with an ideal uncompressible fluid instead. This assumption means that we may transfer any pressure generated at one end of some container of liquid to the other immediately. The discretization logic we employ is that we subdivide space into a collection of containers and treat water in each one of them as being homogeneous. Any hydro-static pressure or force effect is applied to the cell as a whole. There are multiple ways to separate the space. In the case of a regular orthogonal grid we refer to the system as

euclidean [1]. An irregular grid is also a plausible approach despite presenting certain algorithm traversal complications even allowing for a very similar use of multiple step semi-Lagrangian algorithms. [2]

2.3.2 Fluid Simulation

The proper generalized way to express the fluid transport model is through the use of a partial differential equation: [8]

$$\frac{\delta \vec{u}}{\delta t} = -(\vec{u} \cdot \bigtriangledown)\vec{u} - \frac{1}{\rho} \bigtriangledown p + \vec{f}$$

where p is pressure, ρ is the material density and f stands for any external force. Our main goal is simulating and estimating the velocity vector $\vec{u}$.

For our levels of accuracy and complexity we will utilize a relatively intuitive hydraulic pipe model.[1] This model belongs into the category of shallow water models, which are ideal for our scenario as we mostly want to simulate river flows in active motion anyway. Essentially any water simulation solution is plausible here and its accuracy will impact the resulting accuracy of the material transport model. The basic working principle of this method uses space subdivided into Euclidean constant area square cells. Each cell is connected to it's four perpendicular neighbors only via virtual pipes. The pipe is effectively the entire cross section over which the neighbors touch each other, but the significance of this distinction is having a representation of such connection within our model.

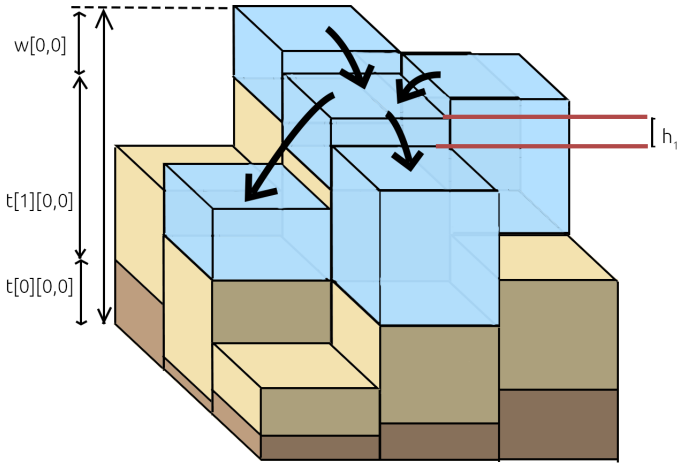

Figure 2.3. Four neighboring cells accelerating water based on surface height.

Unlike the thermal erosion scenario where each step was independent from the preceding ones, the pipe model preserves velocity information for each pipe in between steps. The updates of this model happen at discrete time steps. During each update the hydro-static pressure within each cell is computed and appropriate acceleration on the neighboring pipes is applied dictated by the formula:

$$\vec{a}_i j = \frac{\Delta P_{i,j}}{\rho l}$$

where $\Delta P_{i,j}$ is the difference in pressure between cells i and j, ρ is the density of the fluid and l is the length of the pipe in question – within a discretized scenario it is the

distance between the centers of cells i and j. Once the accelerations are computed the pipes draw upon their neighboring cells transferring water in the appropriate direction with flow equal to:

$$\vec{f}_{i,j}^{t+\Delta t} = \vec{f}_{i,j}^{t} + \Delta t C \vec{a}_{i,j}$$

where C is the cross section area of the pipe – In this model equal to $C = l^2$. As seen from this account such solution is very convenient for a GPU implementation and in practice resulted in a very pleasant water behavior, although the relatively limited count of connecting pipes does result in some visible emergent behavior when studied closely. Sometimes the material prefers to travel in the orthogonal directions.

To find the pressure at cell i, P_i we simply use the standard hydrostatic equation. Then the pressure difference between that cell and a neighboring cell j would be:

$$\Delta P_{i,j} = \rho g(h_i - h_j)$$

where ρ is the density of water, g the gravitational constant and h_a marks the heights of cell a. Note that $P_{i,j} = -P_{j,i}$ holds for any two cells i, j. This will prove useful during implementation. The height of a cell is considered at the water surface level, therefore it is equal to:

$$h_a = w_a + \sum_n t_{a,n}$$

where w_a refers to the current water level and $t_{a,i}$ is the amount of material at the i-th layer on cell a. We assume that the water does not enter in between the terrain levels and remains on the surface similarly to how we do not allow the material layers to slide under one another.

2.3.3 Kinetic Erosion

The first actual medium based erosion algorithm we are going to analyze is kinetic erosion, also know as force-based erosion.[4] Water moving across the terrain surface loosens sediment particles and moves them along. A key value for simulation is the sediment transport capacity of the flow $S_{i,k}^m$ which is defined by:

$$S_{i,k}^m = ||\vec{v}_i|| C_k sin(\alpha_i)$$

where $\vec{v}_i$ is the velocity vector at cell i, C_k stands for the sediment capacity constant unique to each layer material k and α_i is the maximal slope of the local terrain at i. [4] proposes that a hard rocky material would have $C_k = 0.0001$ while sand about $C_k = 0.1$.

Since the water transport model we are using consists of water pipes, we actually need to solve for $\vec{v}_i$ before proceeding with the computation. Since our pipes run in two perpendicular directions vertically and horizontally we may solve these directions separately for their respective velocities and then use those values as elements of the vector $\vec{v}_i$. Let us denote the four neighboring cell indexes n,e,s,w based on the names of the cardinal directions which should make this section easier to follow. Next we need to define where the *positive* direction will be – in our case we start numbering cells with $[0, 0]$ in the north-west so the positive flow will be headed in the south-eastern direction.

$$v_x = \frac{W_{i,x}}{l\bar{w}_i}$$

where $W_{i,x}$ is the amount of water passing through i per unit of time horizontally and $\bar{w}_i$ the average water level at the cell found as:

$$\bar{w}_i = \frac{w_i^t + w_i^{t-\Delta t}}{2}$$

.

We solve for the passing amounts $W_{i,x}$, $W_{i,y}$ in the following way:

$$W_{i,x} = \frac{f_{w,i} + f_{i,e}}{2}, W_{i,y} = \frac{f_{n,i} + f_{i,s}}{2}$$

$$\vec{v} = \begin{pmatrix} \vec{W_{i,x}} \\ W_{i,y} \end{pmatrix}$$

With the knowledge of the sediment transport capacity we may proceed to the erosion step itself. During each update the current dissolved sediment volume S'_i is compared to the capacity. If the sediment level is lower than the capacity, material is dissolved under some material-specific ratio. If the sediment level exceeds the capacity then the excess is deposited. After this the transportation step is executed2.3.5. As the material is shifted around and water levels change the landscape will begin to shift.

2.3.4 Hydraulic Erosion

Hydraulic erosion – also known as dissolution-based erosion – describes a very similar effect to kinetic erosion governed by a slightly different capacity. The phenomenon in question here is water penetrating the land mass and forming a heavy soaked regolith on the bed. As long as the area is covered by water this layer remains malleable and will be dragged around by the currents. Once the water level drops the capacity decreases and sedimentation occurs.

Unlike the force-based erosion, the capacity to form regolith for material k on cell i equals:

$$R_{i,k} = min(w_i, c_k)$$

where c_k is a material-specific constant dictating maximal thickness of regolith. [4] the amount of dissolved regolith is always maximal. This means that there is no rate of being dissolved.

2.3.5 Material Transportation

Material transportation surprisingly isn't nearly as straightforward as the pipe model used in our water simulation. The common approach borrows the same method used in rendering applications when scaling and transforming raster images – advection. The transport of material in general terms is expressed as:

$$\frac{\delta S'}{\delta t} = -(\vec{v} \cdot \bigtriangledown) S'$$

In our case we will be applying a semi-Lagrangian backward advection method. This means that in essence we are taking the velocity vector $\vec{v}$ in the given cell and instead of shifting the material off somewhere, we look backwards in the direction of $-\vec{v}$. We set the amount of material in our cell at t^{n+1} to that of the advected cell at t^n. Most of the time this will not land on a specific cell so a simple bilinear weight interpolation on the four nearest cell centers is taken instead.

The first obvious problem with this method is that it does not guarantee preservation of material. [9] describes a semi-Lagrangian method which does conserve mass between steps. The second potential issue is that single step semi-Lagrangian methods are not as accurate even when increasing the state space resolution massively. [8] If our simulation were to be very accuracy dependent then the best approach would be applying a higher order semi-Lagrangian MacCormack equations. [4] In a nutshell these methods take

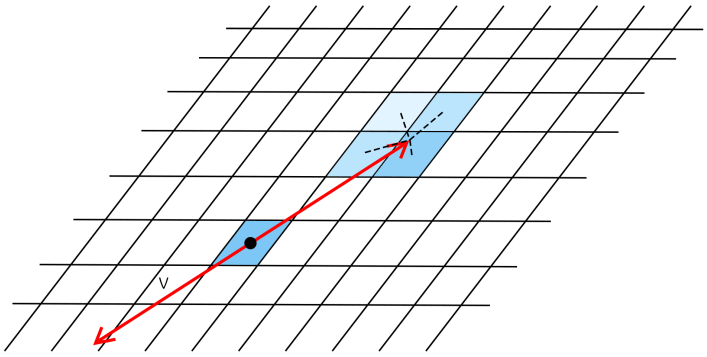

Figure 2.4. Semi-Lagrangian backward advection

one step backwards, center at the nearest rounded vertex and then take another step backwards from there resulting in a much more detailed flow of particles. We haven't used either one of these techniques in our implementation, since accuracy wasn't the priority, however, we provide reference to them to show the proper way for expansion.

2.3.6 Application within the Layered Model

So far we have been talking about the material involved in the simulation in general terms, however an issue arises once we start looking into the step by step application of the aforementioned equations. What happens once we start shifting and exposed piece of rock over some soil and dust layered on top of it in the neighboring cells? The material cannot realistically be deposited back into the rock layer. One potentially plausible way of addressing such scenario would be adding a new rock layer on top. However, such an approach might soon result in memory overflow shall a scenario arise where two different materials are fighting to be placed in the cell resulting in new layers being placed with each update.

The most usual interpretation which circumvents this issue is assuming that an eroded hardened rock [or any other harder material] would not have the capacity to immediately reshape into it's original sturdiness.[4] While the erosion step removes material as expected using the properties of the respective materials, deposition activities place everything into the topmost layer only alleviating representation complications. An extension of such system adds realism by keeping track of the time for which each cell remained unaffected by dissolving effects and slowly shifting some portion of the material down to the lower more hardened layers.

2.3.7 Order of Application

So far we have described a number of independent models. The way they are put together, first come water inputs and removal. If we designate some cells to be springs or drains this is where they apply their changes. Secondly, water pipes compute accelerations and shift water. Then velocity fields are calculated and both kinetic and hydraulic erosion perform their deposition / erosion step. In the end, thermal weathering is applied, moving any unstable overhangs back into balanced position.

2.4 Interaction Methods

Even though the field of terrain simulation has experienced substantial development over the last several decades, there haven't been any notably distinct selection or ma-

nipulation tools introduced yet. Most modern terraining approaches for commercial use focus on node-based terrain synthesis using 2D heightmap layers connected via operators similar to modern shader designs in tools such as Blender or 3dsMax. These tools are not as much interested in terrain simulation itself, but rather in the synthesis of entirely new terrains for use in games or movies. Applications dedicated to simulation of particle systems more akin to our hydraulic erosion processes usually come bundled in as a subsystem of an overarching engine. For this reason I also included game level editor and image editing software features in my list of references.

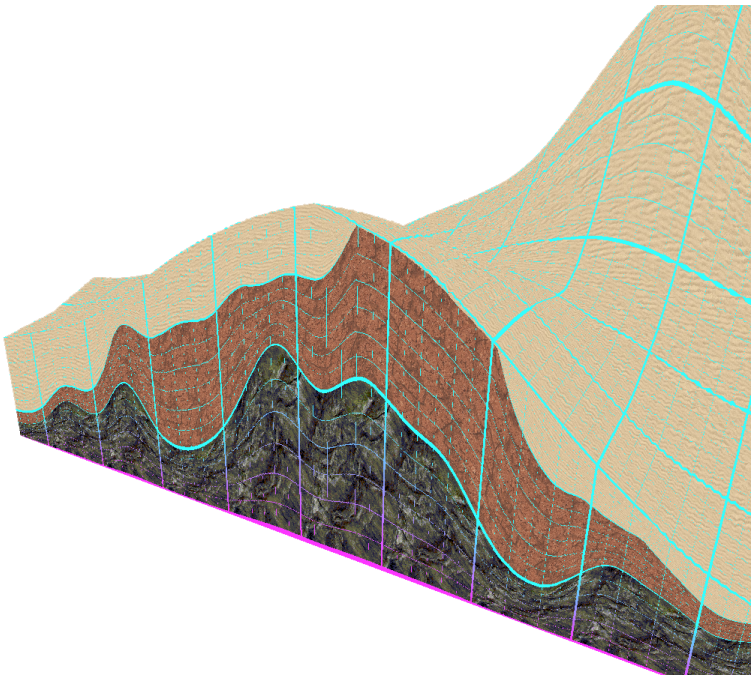

Figure 2.5. Layered terrain representation.

The first natural inclusion is a projected mouse cursor. For terrain shaping, creation of entities, masking or any other localized activity a cursor is an essential system to have. The logic behind projecting cursors is explored further in 2.7.3. Out of the other presented tools, the ability to mask out the influence of our erosion effects seems very useful. A Gaussian falloff brush lends itself as it will result in smoother transitions than a plain circular selection with no gradient. A so called polygonal lasso tool is also a popular choice for selection among digital artists. Generally, due to the nature of our layered terrain representation we may perceive any raster editing tool as a convenient accessory. We don't want to delve too deep into painting tools, but during the implementation it turned out that being able to step in and add material in places while simulating makes the working process much more interesting. In a similar way we also allow the user to manually paint in bodies of water.

Application Name	Tool Description
Earth Sculptor [1]	grab tool for terrain
Adobe Photoshop [2]	Magic wand
	Toggling continuous selection
	Growing and shrinking selection
Fractal Terrains 3 [3]	Deterrace, fill basin
Lithosphere Terrain Generator [4]	Mathematical operators on terrain layers
PnP TerrainCreator [5]	Support for pressure sensitive pen tablets
	Splatted and texturemap terrain texturing
Terragen 4 [6]	Georeferencing
VUE xStream 2016 [7]	Heightfield Terrains
	Spline Terrain Effect
	LOD drawing
World Creator 2 [8]	Curve Editor, Smart Scribbles
World Machine [9]	Terrain synthesis from shapes.

Table 2.1. Table of some noteworthy features found in popular editing applications.

An entity placement and deletion system enables the implementation of sprinklers – locations within the map that output water with each update step. In many scenarios these can simulate natural springs, rivers and much more.

2.5 Polygon Lasso Mathematics

To implement the polygon lasso tool on a discrete grid we need an algorithm to tell us whether we are inside or outside of the selection. We are using a raycasting algorithm counting the number of polygon side intersections. This way even non-convex polygons are accounted for. The algorithm is casting the ray in the direction of one of the axis to simplify the computations. If the ray's y is too high or too low we continue, otherwise we determine on which side of the line the point x resides. By aligning A to always have greater y coordinate we will resolve this for each segment with the same frame of reference. In the end we check whether the number of intersections is odd or even. The algorithm is as follows.2.6

2.6 Spacial Navigation Methods

With regards to the spacial navigation we utilize a semi-locked camera with two degrees of freedom. This means that the user is able to change the yaw and pitch, but the roll remains fixed. Since we are assembling a terrain editing tool, there is never a need to

1 http://www.earthsculptor.com/manual/index.htm
2 https://www.adobe.com/products/photoshop.html
3 https://secure.profantasy.com/products/ft_features.asp
4 http://lithosphere.codeflow.org/
5 http://pnp-terraincreator.com/user.php
6 https://planetside.co.uk/
7 https://info.e-onsoftware.com/vue
8 www.world-creator.com
9 http://www.world-machine.com/features.php

```
for( each segment AB in the polygon path ) do:
   A = end point with greater y value;
   B = end point with lower y value;
   intersects = 0;
   if( y > A.y & y > B.y ) || ( y < A.y & y < B.y ) then
      continue;
   end
   //checks whether x of the studied point is on one side of the line.
   intersects += x > [ B.x + (y - B.y) * (A.x - B.x)/(A.y - B.y) ]?0:1;
end
```

Figure 2.6. Pseudocode describing how the raycasting algorithm

rotate the environment sideways. As a result the lack of roll control does not have a constraining feel to it.

In fact, a common issue that 3D manipulation tools for modeling and sculpting have to deal with is conveying the navigational point of view to the user clearly. While it is possible to descend beneath the plane and make a 180 vertical flip we do not run the risk of confusing the sense of up and down as the user is never incentified to focus their attention to elements that would guide them that way.

The camera movement was initially facilitated by the classical first-person spectator view with AWSD and looking around by holding down the right mouse button. However, testing has shown that most of the time users find themselves struggling to simultaneously use keyboard and mouse inputs for navigation. We ultimately reserved the left mouse button for tools and let the right mouse button look around, wheel zoom and wheel press drag the camera. This way, all degrees of freedom are facilitated through the mouse and keys are reserved to mode switching operations.

Smart scribbles seem to be a very intuitive input method for unfamiliar users to quickly generate terrain. They allow the user to lay down splines atop the terrain and model either ridges or valleys along their path. Since our application is compiled for Windows we have access to much tighter controls with mouse and keyboard, so we primarily focused on mouse sculpting, however, smart scribbles would be high on our list of features to add in future expansions.

An absolute necessity is the ability to load and store terrain data in a file. For that reason we include interface for loading and saving individual layers encoded as color intensities into raster images. Operating systems natively support options for viewing common image types, such as bmp, jpg and png and users are generally familiar with them. Furthermore the encoded data will display the terrain map in a comprehensible way and even provide a thumbnail preview in the folder resulting in convenient handling.

2.7 Display Computation

In order to construct an efficient 3D application it is necessary to gain some insight into the mathematical background of how spacial projection operates. The processes discussed in this section are general and are applied in every 3D rendering engine.

2.7.1 Data Representation

The most common data representation technique in 3D computer graphics is the polygonal envelope. This means that only the surface of the rendered body is stored in memory. Additionally the surface of the body is represented by faces – polygons stretched

out between vertexes. Ultimately a shape is fully defined as an ordered collection of vertexes and a collection of lists of indexes pointing to them, each index list forming one face of the shape. Triangular polygons are the most common as they yield a number of conveniences during implementation such as each face always being flat. For each set of 3 vertexes there always exists a plane going through them in a 3 dimensional space.

Polygonal representation suits our situation very well. Even though our data is semi-voxel represented, due to the assumption of it being homogeneous throughout a layer the only really valuable information is the height in each column. It should be noted that other implementations have been equally successful with a ray casting voxel based renderer. [6]

2.7.2 3D Rendering Pipeline

The rendering pipeline of a polygonal mesh consists of layered projections between spaces of different dimension. Namely we aim to project the simulated terrain from its internal coordinates to the pixel coordinate space of the user's monitor screen. Let's assume that we have vertex $\vec{a}$ represented as a 3 dimensional vector in the coordinates of the world. Additionally we define the position of the camera $\vec{c}$ and it's viewing direction $\vec{d}$ as 3 dimensional vectors as well. We are easily able to calculate a 3x3 linear transformation matrix C together with a 1x3 offset vector $\vec{b}$ to express $\vec{a}$ in terms of the orthogonal base situated at our camera location.

$$\vec{a'} = C\vec{a} + \vec{b}$$

For the sake of being able to chain projections like this with just a single matrix multiplication, the coordinate system is in practical applications augmented into an 4x4 affine transform matrix which encompasses both the linear transformation and the translation. The 4th dimension of the position vector is set to 1 which serves the purpose of providing a way of encoding the translation step mathematically. Notice that this matrix multiplication is invertable up to this point which becomes essential later on.

Once the scene we want to render is expressed in terms of the camera-based coordinates a so called perspective matrix multiplication is applied. This step on a mathematical level distorts the geometry with distance to form a cone shape. There are multiple ways to express the parameters dictating the properties of this cone, but what they refer to is basically the same thing. Either the width and height of a projection plane are used which carries some convenience in implementation or a field of view angle is entered. Each can be used to solve for the other. Afterwards, dividing the x and y coordinates by the z component yields a projected 2D affine space vector coordinate in terms of the projection space base.

2.7.3 Inverting the Pipeline

In case we would like to let our user interact with the world they see on their screen we run into a little complication. The mouse only provides us with information in the screen space with reduced dimension. A projection reducing the dimension of our vectors is inherently irreversible as it takes away information. However knowing the 2D screen space coordinate vector a'' in combination with the parameters used to originally project into perspective enables us to reconstruct a line coming from the camera origin [0,0,0] through our known 2D point a'' cast into the camera coordinate system. If we

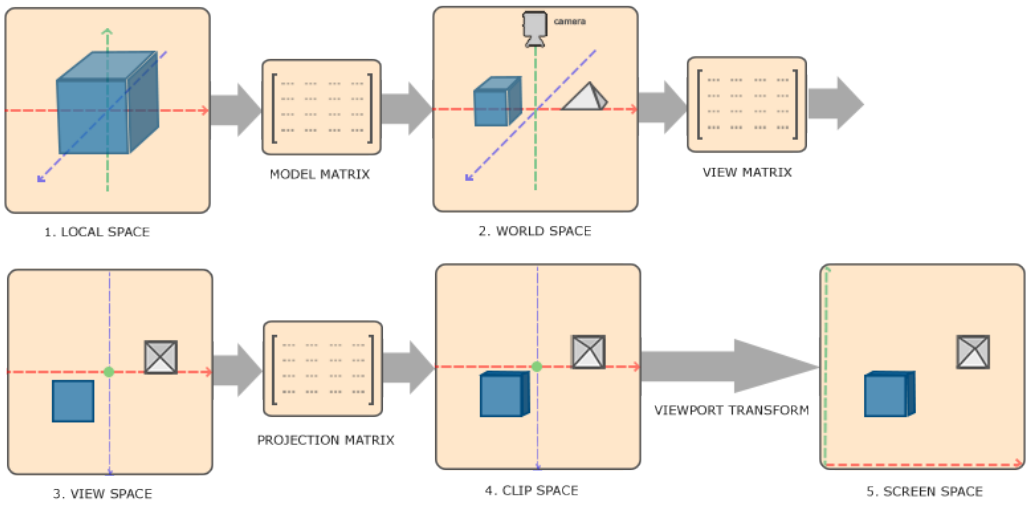

Figure 2.7. Rendering coordinate systems (taken from `learnopengl.com`)

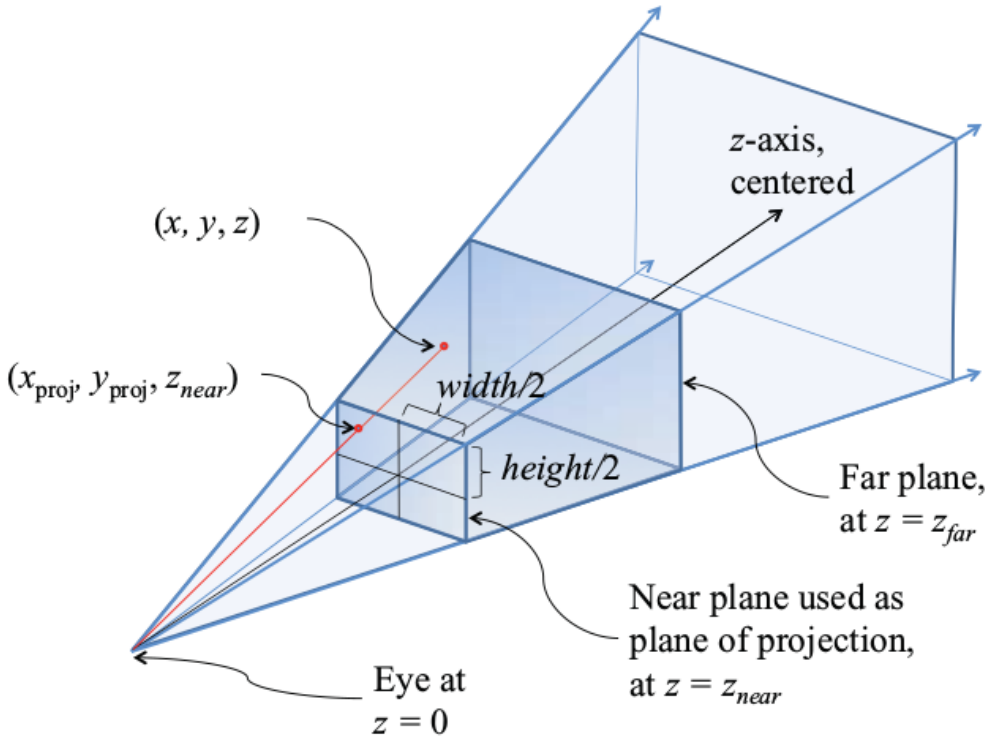

Figure 2.8. Perspective projection transformation (taken from `learnopengl.com`)

additionally recorded the previous z coordinate of a' – which is a common practice in application for all sorts of purposes – we may solve for an intersect of a plane parallel to our projection plane at the distance z and the line. This fully reconstructs the pre-flattening camera space coordinate with a unique solution in scenarios that we are concerned with. From that point onward the perspective transformation and all preceding affine projections are fully revertible. Note that by adjusting the z coordinate we may in fact solve for points other than just the visible projected one.

Chapter 3
Application Design

In this chapter we will break down the decision process behind individual elements of the design. In order to deliver a smooth end user experience the design is built on top of a set of interlocking threads and mutexes which facilitate the necessary communication between the frontend and the backend. 3.7 shows an outline of how these individual components are connected.

3.1 Tools Used

We have developed our application in C++ compiled under Cuda 8.0 for the Windows environment exclusively. The libraries used are only Freeglut [10], AntTweakBar [11] and CImg[12]. FreeGLUT provides an up to date GLUT implementation with both convenient OpenGL access and window context environment with update, render and callback loops. AntTweakBar operates on both OpenGL and DirectX and provides a lightweight UI utility for relatively advanced user inputs ranging from numerical values to value sliders and color pickers. CImg is an extremely light weight C++ image library which provides access to quick loading and saving of images.

3.2 OpenGL Rendering (Frontend)

The OpenGL section of the application is essentially a self-contained receiver unit. It holds its buffers and periodically renders, occasionally receiving vertex data updates through a mutex protected CUDA interlop.

3.2.1 View Modes

There are three major view points available – Texture, Mask and Analytical. Texture mode will render textures based on the visible materials. This mode is suitable for examining the composition of the terrain and erosion processes. The Mask mode exists mostly to visualize current selection without any obstructions from other communicating element. Finally, the analytical mode attempts to visualize depth by overlaying the terrain with contours and lightening the color with height. Additionally a trace of the the mask will be shown tinting the terrain either red or green. The contours have proven to be an invaluable tool especially while working with water flows as they draw out the optimal shoreline. While viewing water at an angle on a slope it is sometimes hard to estimate depth without these guidelines.

3.2.2 Geometry

The rendering of the terrain height map is executed via a single glDrawElements as a glTriangleStrip call with the use of a trick to link up the indexes together demonstrated in figure 3.2. There is one glVertexBuffer holding the xyz positions accompanied by a mask and texture glVertexBuffers. These are all bound and mapped as inputs for

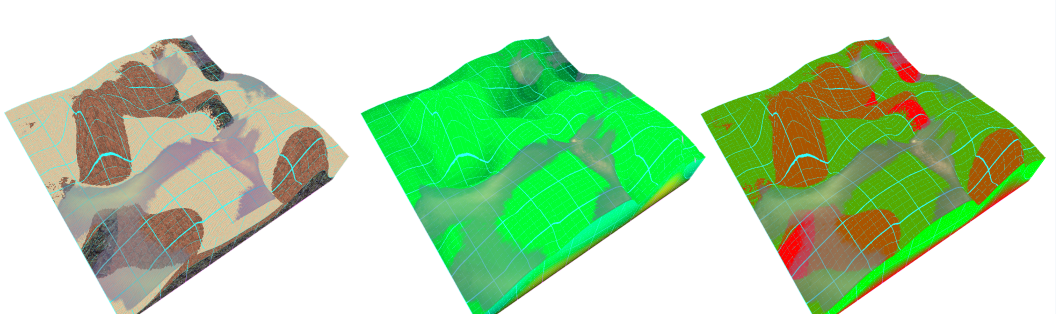

Figure 3.1. The texture, analytical and mask views in order.

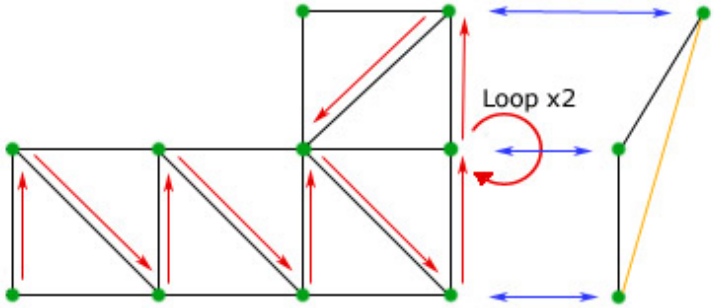

Figure 3.2. GLTriangleStrip trick allowing for drawing planes within a single strip.

the shaders. As the application only allows us to remove layers from the top down, a single surface spanning the terrain is more than sufficient. UV texturing coordinates are created based on world coordinates within the vertex shader itself and therefore require no designated buffer.

In a similar fashion the water surface is represented. The water shader variables are in fact an exact copy of those used in the terrain one. The selection mask buffer has been re-purposed as the amount of sediment dissolved on this cell and the texture buffer holds depth information. Shallow water is rendered progressively less opaque reaching 0 in dry areas. This smooths out shores significantly and subtly solves the issue of Z-fighting which would otherwise occur.

The sides of the terrain are a little bit more complicated as they change their polygon count with the number of visible layers. The indexing of the side panels is winding the triangle strip segments in a spiral like fashion upwards one level at a time. This way we pass height information to all vertexes with no need to change the structure of the buffer of break up the call. We simply multiply the number of drawn triangles per layer by the number of layers to display from the bottom up.

3.2.3 Vertex Shader

As is usual, the vertex shader mainly serves to pass arguments to the tessellation and rasterization. An effective trick mentioned in 3.2.2 is defining the texturing coordinates based on world coordinates $Texcoord = vertexPositionModelspace.xz * 0.03/0.4;$. Besides this there isn't much happening in our vertex shaders.

3.2.4 Fragment Shader

The fragment shader uses the modulo operation $mod(pos.x; interval) < width$ on the texturing coordinates to generate a grid over the terrain.3.3 Depth perception is a vital element even when working with a grid locked element like our terrain. Mode and pointer indicators are all passed through a uniform and used in an if statement to switch between textures and colors. Furthermore, the selection polygon and sprinkler locations are passed through a uniform array of fixed size 100 as well. So far, this amount of uniforms has not proven problematic.

When it comes to displaying the polygons there are two major elements. The connecting lines and the vertexes themselves. Vertexes are rendered as squares to ease the computation.3.4 In the case of the lines we are basing our calculation on the equation for the distance of a point from a line defined by two points:

$$dist(P_1, P_2, (x_0, y_0)) = \frac{|(y_2 - y_1)x_0 - (x_2 - x_1)y_0 + x_2y_1 - y_2x_1|}{\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}}$$

However this equation represents the distance from a line and not the segment, for this reason we additionally compute the distance between the position *pos* of the currently rendered point and both points A and B. If *pos* is further away from either one of those points then is their distance it means that pos is outside of the segment and therefore should not be painted as the line. This test is computed for each line of the polygon.

```
if (mod(Texcoord.x,1)<0.02 || mod(Texcoord.y,1)<0.03) {
  color = vec4(0.2+max(0.8-maskVert*0.8,0),0.2+(maskVert*0.8),1,opacity);
} else if (mod(Texcoord.x*3,1)<0.01 || mod(Texcoord.y*3,1)<0.015) {
  color = vec4(0.2+max(0.8-maskVert*0.8,0),0.2+(maskVert*0.8),1,opacity);
} else if (mod(Texcoord.x*6,1)<0.01 || mod(Texcoord.y*6,1)<0.02) {
  color = vec4(0.2+max(0.8-maskVert*0.8,0),0.2+(maskVert*0.8),1,opacity);
}
```

Figure 3.3. Displaying an overlay grid across the terrain with modulo.

```
//Grabbing points for polygons
for(int i = 0; i < polygonLength; ++i) {
   float diffx = polygon[i].x-pos.x+0.5;
   float diffz = polygon[i].z-pos.z+0.5;
      if(diffx > 0 && diffx < 1 && diffz > 0 && diffz < 1) {
         color = vec4(1,1,0.9,1);
         return;
      }
}
```

Figure 3.4. Rendering selection squares around vertexes from an array.

3.2.5 UI Layout

The logic behind the UI layout is to visually separate all the logically connected options into their respective sections. In the top left and right corners we have big buttons for toggling view modes and editing tools respectively. Bellow the actions there is the

```
//Polygon rendering
for(int i=1; i<polygonLength+connectPoly-1;++i) {
   //Distance between points A and B.
   float distAB = ( (polygon[i].x-polygon[i-1].x)*
                    (polygon[i].x-polygon[i-1].x)+
                    (polygon[i].z-polygon[i-1].z)*
                    (polygon[i].z-polygon[i-1].z) );
   //distance of point pos from the line.
   float linedist = abs( (polygon[i].z-polygon[i-1].z)*pos.x-
                         (polygon[i].x-polygon[i-1].x)*pos.z+
                          polygon[i].x*polygon[i-1].z-
                          polygon[i].z*polygon[i-1].x )/sqrt(distAB);
   //Distance between point pos and point A.
   float distA = ( (pos.x-polygon[i].x)*(pos.x-polygon[i].x)+
                   (pos.z-polygon[i].z)*(pos.z-polygon[i].z) );
   //Distance between point pos and point B.
   float distB = ( (pos.x-polygon[i-1].x)*(pos.x-polygon[i-1].x)+
                   (pos.z-polygon[i-1].z)*(pos.z-polygon[i-1].z) );
   //Clamp the line only to the segment between A and B.
   if(linedist<0.1 && distA<distAB && distB < distAB) {
      //close means rendering the line.
      close = true;
   }
}
```

Figure 3.5. Rendering connecting lines of a polygon

fixed global setting / erosion bar. This element may not be minimized and contains all the settings not related to materials and layers. Finally the movable Layer and Material panels initialized to the bottom left. The fixed position tools and erosion settings are located on the right mainly to use up screen space efficiently while scaling the application window since the display context is centered at top left. Arguably the erode action itself could warrant being a big button, but most of the time erosion is toggled with the spacebar anyway and the slot underneath all the options solidifies the role of the overall panel.

3.2.6 UI Implementation

As foreshadowed the UI coding consists of two major elements. The AntTweakBar library is used to allow the user to control numeric inputs, load texture and toggle booleans. On top of this however, there is a custom UI system in place consists of big clickable buttons fixed against the borders of the screen. While the TweakBar could suffice all the functionality on its own, the tool buttons stand out much better this way and help to reduce the clutter that is slowly building up with both the layer and material panel dragged out and being used. Additional screen management options come with the fact that the TweakBar enables dragging and minimizing windows.

3.3 Simulation Structure (Backend)

On the other side of the spectrum is the data representation. This part of the code operates entirely within the CUDA memory allocation space.

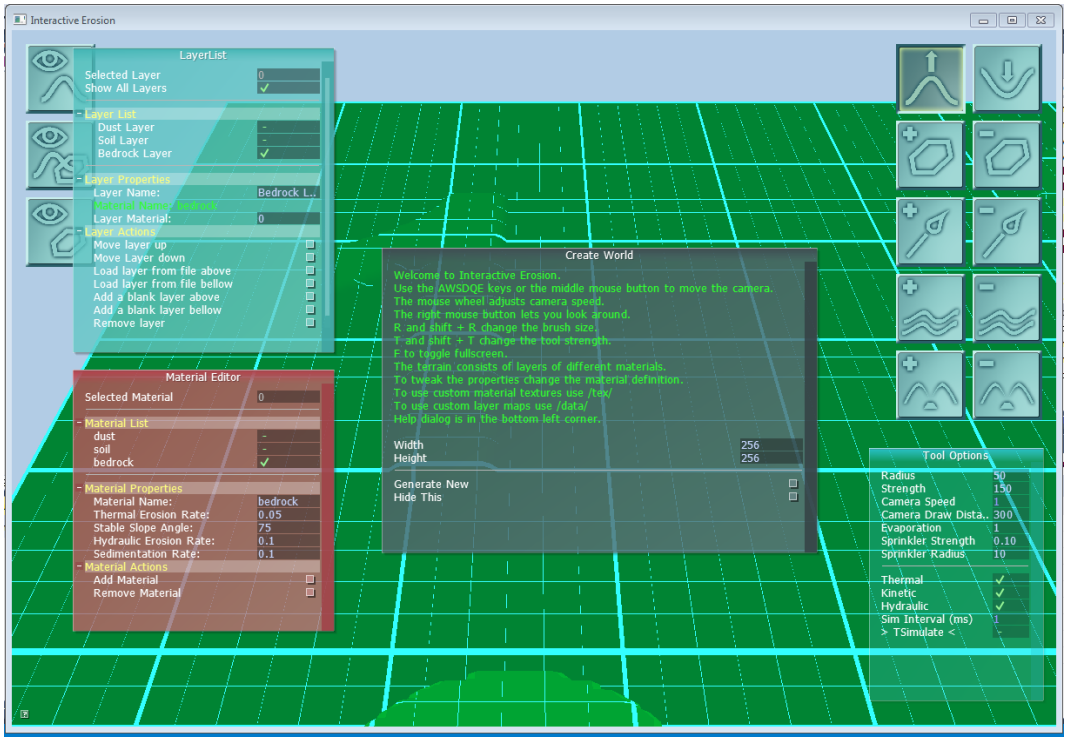

Figure 3.6. UI view upon loading the application. Manual at B.

3.3.1 Terrain Representation

At the core of the backend lies the SimMap class. It holds an std::vector of both SimLayers and SimMaterials and facilitates many of the calls requesting the addition or removal of either one. During initialization or runtime a SimLayer may be assigned data from a file. This transfers it directly to the GPU device memory avoiding any storage of data on the host. SimMap additionally holds about 14 SimLayers designated to tracking statistics such as water levels, water velocities, added sums of heights for fast rendering and sediment or regolith deposits. One of the problems when calling kernels with this many parameters is that the parameter memory is depleted resulting in a crash. To allow the SimMap to potentially scale indefinitely both layer and material pointer lists are copied into an array on the device. Then each kernel only receives a pointer to that list of pointers within a single argument. It is the task of the SimMap class to keep these references up to date. To help with this, the SimMap also holds a pointer to it's Windows mutex `HANDLE`.

A SimLayers device data storage is a 1D float array. All grid indexing is done through: `idx = row * width + column`. SimLayers do not actually hold any data regarding their behavior on their own, instead they refer to an id of a SimMaterial. This fetching of references causes additional memory access during parallel computation, however the primary focus of our application design is to open up the possibilities for the user to freely tweak as many options as possible. From the user perspective this allows them to entirely delete layer's data without losing their material settings or even reuse the same material on several layers. A convenient way of untangling these array index references when working with them is including a define header with each of the kernel definitions which turns something opaque such as `dTerrainData[dIntParams[3]][idx]` into `SEDIMENT[idx]`.

3.4 Thread Structure

In order to ensure a run without memory access collisions we introduce two major mutexes. One for the SimMap and one for the OpenGL buffers. These together with wait guards cover all critical points in our program and do not clog on the user side even while blocking the SimMap by updates with minimal interval set to 1ms.

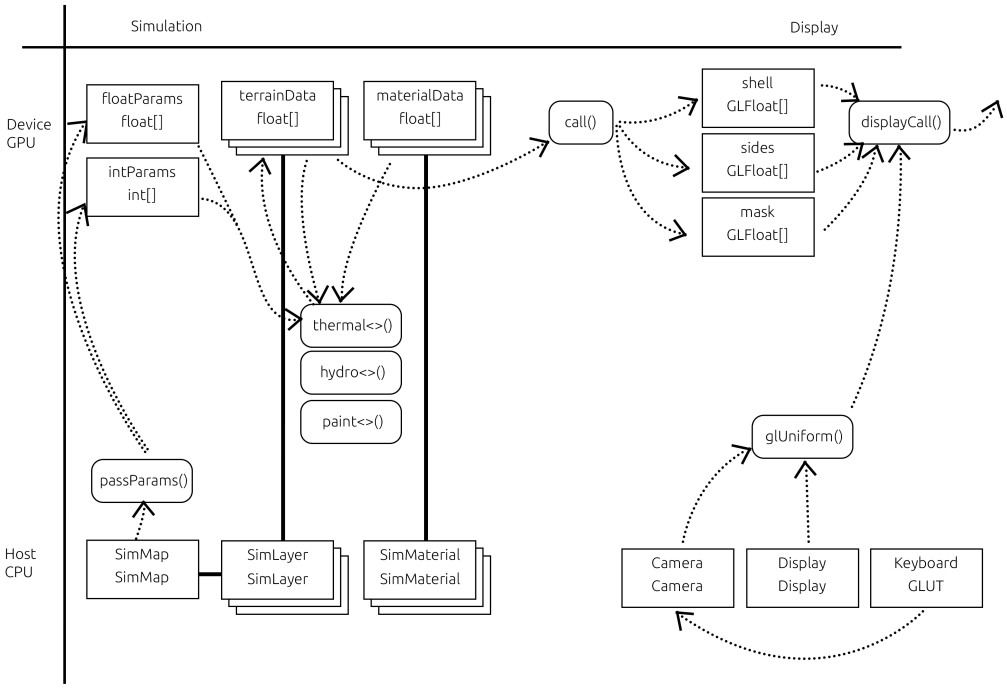

Figure 3.7. Diagram showing the layout of classes and calls.

We utilize the default GLUT update and render functions to do their respective things. Since GLUT only provides onClick, onRelease and onDrag callbacks, the intuitively expected behavior of holding down the mouse button to apply a tool repeatedly has to be done through the update loop checking the mouse state. Instead of taking the straightforward route and implementing the kernel calls within the body of this function we set up worker threads to do this instead.

Each activity that the user can take has its own separate CUDA file which launches a perpetual worker thread on the background during initialization and puts it to sleep immediately afterwards. All of this thread infrastructure is hidden away behind a unique namespace to avoid contaminating the global namespace with common function names such as *initThread()*. Once the user issues a command, necessary variables regarding the cursor position and tool options are stored in a struct. The worker thread is woken up and enters the mutex request state. At this point the update function and user navigation proceeds as usual with no further interruptions. Only once the worker thread receives the hold on the SimMap mutex it refreshes all data memory pointers, in case the user has taken a host based editing action in between.

Once all the pointers are up to date the kernel procedure is launched. Notice that at this point both the GL Mutex and main GLUT update thread are unlocked. Naturally, the GPU hardware cannot simultaneously render and compute on the same physical

processing units, but this way we are delegating this decision to the driver and most importantly of all the CPU activities are not blocked. After the kernel call finishes, the mutex is unlocked and the thread loops back to the beginning awaiting another wake call.

This sort of thread separation carries its own set of drawbacks and advantages. First of all, the order of operation is not entirely deterministic once multiple threads request access to edit the SimMap. Most of the time, this barely matters as none of the available editing tools really change their outcomes when applied in reversed order. Unless your application aims to implement hundreds of activities each with their separate kernel. The general format of how these services are implemented is presented bellow:

```
instructionParam::myCudaParam_t params;
std::mutex m;
std::condition_variable cv;
unsigned int requested = 0;
bool active = true;
volatile bool activityRequest = false;

void worker_thread()
{
   while (true) {
      std::unique_lock<std::mutex> lk(m);
       // wait until user requests this activity
      cv.wait(lk, [] {return activityRequest; });
       // escape sequence for termination
      if (!active) { lk.unlock(); return; }
      activity(); // the function calling the kernel.
      lk.unlock();
      activityRequest = false;
   }
}

void killThread() {
   printf("Terminating Thermal Thread\n");
   active = false;
   cv.notify_one();
}

void initThread() {
   printf("Initing Thermal Thread\n");
   std::thread worker(worker_thread);
   worker.detach();
}

void erodeThermal(float x, float y, float z, int idx,...) {
   simMap->passMaterialListPointers();
   params = { x,y,z,toolRadius, toolStregth*dir, idx, simMap,... };
    // ping thread
   activityRequest = true;
   cv.notify_one();
}
```

Figure 3.8. Example of a worker thread calling activity();

3.4.1 Editable Data

User defined parameters do have their host representation. This means that there is a variable in computer memory which holds their value. This variable is passed to the AntTweakBar interface via a pointer and designated for editing. Before each erosion call, after the mutex is locked, each worker thread is responsible for refreshing the device counterparts to these variables. This way we circumvent the issue of the ui layer attempting to write into GPU memory during an ongoing computation.

3.5 Computation Implementation Overview

The usual approach to computing update steps in CUDA is similar to how OpenGL paints to the screen with the use of two swap buffers being switched back and forth. This way each element is stored in the memory twice and during each update step the instances take turn being the read or written buffer. In our implementation we attempted a slightly different approach with an in-between buffer. We only define one SimMap and have a blank working memory buffer provided to the kernel. Deltas are written into this buffer and the kernel finishes up or synchronizes threads. In the subsequent step the computed delta values are applied to the original memory. In certain scenarios, such as adding up incoming material during thermal erosion, this is also an unavoidable measure to prevent memory write race conditions, as atomic add operations for floats and doubles have not been implemented in CUDA as of now. Processes such as advection which do not require this middle-step do so reusing the same buffer from the previous water transport simulation and therefore are not as taxing in terms of memory allocation. Due to the nature of how material transports are treated in our model, there never is a need to express movement of more than a single layer of data. In a scenario where up to 20 layers are allowed even when using 9 variables per tile at a time the delta value solution is more memory efficient than having a dual swap buffer for each of the 20 layers.

3.5.1 Helper Functions

There are two major device functions defined in each kernel definition file. Because of how CUDA source files are being linked [in version 8.0] there doesn't seem to be a clean way of including CUDA functions in a way that separates definition and implementation. [13] This introduces a minor redundancy in the code as certain frequently used device functions need to be pasted within each instruction file separately.

The first useful utility is a boundary checking function to prevent unauthorized memory access:

```
__device__
bool isOutside(int c, int r, int w, int h) {
   return ((c < 0) || (r < 0) || (c >= w) || (h >= h));
}
```

Figure 3.9. Device function checking bounds.

The second most noteworthy device function is the Gaussian falloff specifically crafted for the use in our application to be parameterized through the radius of the active area. Expressed mathematically the equation is

$$val = e^{\frac{-((x-c_x)^2+(y-c_y)^2)*4}{r^2}}$$

where c_x and c_y are the position of the center of the Gaussian effect, x and y are the position of the examined point and r is the radius of the brush. The cutoff with 0.01 is included to prevent placing minuscule amounts of material everywhere on the map when using a localized brush like this.

```
__device__
float gaussFalloff(float x, float y, float cx, float cy, float radius) {
   float val = expf(-((((x - cx)*(x - cx) + (y - cy)*(y - cy)) * 4) /
   (radius*radius)));
   return val < 0.01 ? 0 : val;
}
```

Figure 3.10. Device function computing 2D Gaussian falloff.

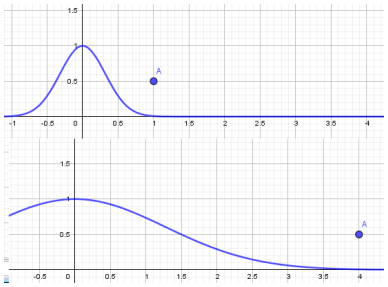

Figure 3.11. Gaussian falloff demonstrated in 2D for r=1 and r=4. radius is denoted by the x coordinate of the point A.

3.5.2 Kernel Parameters

Essentially all of our kernels share the same arguments making their comprehension a one-time undertaking. Most of the arguments are pointers to arrays generated by host-side simulation objects and when used in our code are referred to through their more intelligible simplified names. List of all simplified names has been attached at the end due to length.D

Due to the large amount of arrays in question we are using a modified naming convention for variables that appear in both host and device code. This avoids the problem of kernel code attempting to access `d_` variable in host memory while remaining clear:

- `h_` for host variable stored in host memory.
- `d_` for device variable pointer to `h_` data stored in host memory
- `dd_` for device variable pointer passed as an argument

variable	description
`int* dd_intParams`	Array lengths, selected layer index
`float* dd_floatParams`	Tool strengths, cursor position, evaporation
`float** dd_terrainData`	Pointers to individual SimLayer data fields
`float* dd_working`	Blank tmp memory to transfer data between calls
`int* dd_materialIndex`	Material index for each layer
`float** dd_materialData`	Pointer to erosion properties of each material

Table 3.1. Table of kernel parameters and their meanings.

3.5.3 Common Opening Sequence

Besides the convenient use of `\#define`3.3.1 all of our device kernels share the same opening sequence. It ensures that all subsequent operations have the same frame of reference and protects against executing outside of the bounds when the data set is not rounded to powers of 2:

```
//indexing
const int c = blockIdx.x*blockDim.x + threadIdx.x;
const int r = blockIdx.y*blockDim.y + threadIdx.y;
// Check whether we are within image bounds
if (isOutside(c, r, DW, DH))
   return;
const int cudaindex = c*DW + r;
```

`dd_working[]` is the name of a preallocated temporary memory array. Its function is to hold data between calculation steps during CPU synchronization which doesn't have a lasting memory address. Most commonly these are material or velocity deltas.

3.6 Thermal Kernel

The thermal erosion activity in our implementation consists of two kernels separated by a CPU synchronization step and one device subroutine for iterative erosion of layers.

3.6.1 Setup

In the first step we locate neighbor indexes and check whether they are valid recipients for erosion material and store this information in `other\_valid[i]`. The `idxs[i] >= 0 \&\& idxs[i] < DW*DH` ensures that the neighbor resides within the grid vertically, however because we are using a 1D array. we also need to check horizontal boundaries which is what the rest of the if-clause denotes in the form of a maxterm. We use the fact that the cells in specific directions have fixed position within `idxs` and therefore we can filter them out just through their index alone. The condition reads *either the cell isn't in the 0th column or the neighbor isn't to the west. (-1 direction)* Similarly for the east wards +1 direction with a little less fortunate indexing.

```
idxs[8] = { cudaindex - DW, cudaindex - DW + 1, cudaindex + 1,
          cudaindex + DW + 1, cudaindex + DW,
          cudaindex + DW - 1, cudaindex - 1, cudaindex - DW - 1 };
bool other_valid[8];
for (unsigned char i = 0; i < 8; ++i) {
   if (idxs[i] >= 0 &&
       idxs[i] < DW*DH &&
       (r > 0 || i < 5) &&
       (r < DW-1 || (i!=1 && i !=2 && i!=3))) {
      other[i] = SUMS[idxs[i]];
      other_valid[i] = true;
   } else {
      other_valid[i] = false;
   }
}
```

3.6.2 Subroutine Iteration

With the invalid neighbor targets filtered out we may proceed to the material shifting step as described in 2.3.3. We will erode for each material layer from the top until we reach a point where a layer is either stable or we run out of layers. We allocate several important arrays in the main kernel and pass them to the subroutine via pointers. `other_valid[8]` only checks whether the target is within bounds and this is an information we would want to preserve for each iteration. `validtmp[8]` as the name suggests is for the talus angle condition, `other[8]` is temporary memory for storing the height differences and `target[9]` stores the the material to be distributed and the material to be removed from our cell at the additional index.

```
bool validtmp[8] = { true,true, true, true, true, true, true, true };
float target[9] = {0,0,0,0,0,0,0,0,0};
float other[8] = {0,0,0,0,0,0,0,0};
float sum = 0, maxdif = 0;
```

With these variables the subroutine3.12 checks the conditions, increments `target[]` where necessary. The routine returns `true` in case the current layer has satisfied it's desired offset. In case the layer would want to offset more than it currently holds it means that the one underneath has been exposed. `false` is then returned and the next cycle loops.

```
//Start eroding each layer from the top most:
for (int i = DLEN-1; i >= 0; --i) {
   //If the layer is empty. Skip
   if (dd_terrainData[i][cudaindex] == 0) { continue; }
   //If the layer material was sufficient no further erosion takes place.
   if (d_evalLayer(&myh, other, other_valid, dd_terrainData[i][cudaindex]
                   ,MDATA[MATERIAL[i]]MTHERMAL, MDATA[MATERIAL[i]]MANGLE,
                   validtmp, &sum, &maxdif, target)) {
      break;
   }
}
```

After the subroutine iteration is finished we load up the results into the `dd_working` and close the first kernel. This method does not lead to memory write race condition, because each cell has a variable for each neighboring material source direction.

```
dd_working[cudaindex * 9] = target[8];
for (unsigned char i = 0; i < 8; ++i) {
   if(other_valid[i])
      dd_working[idxs[i] * 9 + 1 + i] = target[i];
}
```

3.6.3 Collection Step

The second step of the thermal algorithm is relatively straightforward as each cell only accesses it's own data index in it. The only complication is material removal with our layered representation.

```
int tidx = cudaindex * 9;
int topidx = 0;
for (unsigned char i = 0; i < DLEN; ++i) {
   if (dd_terrainData[i][cudaindex] > 0) {
      topidx = i;
```

```
    }
}
```

finds the top most layer with some material in it. Once we have it, we start removing material from the top down until the `sum` is satisfied. Afterwards we just add up the incoming materials passed to us from neighbors during the previous step and increment the topmost layer by that amount.

```
//remove material first
float sum = dd_working[tidx];
for (unsigned char i = DLEN - 1; i >= 0 ; --i) {
   if (dd_terrainData[i][cudaindex] >= sum) {
      dd_terrainData[i][cudaindex] -= sum;
      break;
   } else {
      sum -= dd_terrainData[i][cudaindex];
      dd_terrainData[i][cudaindex] = 0;
   }
}
//Add up the deltas.
sum = 0;
for (unsigned int i = 1; i < 9; ++i) {
   sum += dd_working[tidx + i];
}
DUST[cudaindex] += sum;
```

3.7 Hydraulic Kernel

The hydraulic kernel consists of four subsequent kernels for water pipe simulation, one Kinetic, one Hydraulic and two kernels for semi-Lagrangian transport. If the user disables either one of the simulations the water and semi-Lagrangian transports will be computed as usual, however the deposition / erosion step will be omitted.

- `d_simWaterA` *(compare pressure, pass accelerations to pipes)*
- `d_simWaterB` *(add accelerations to pipes)*
- `d_simWaterC` *(compute transported amounts of water)*
- `d_simWaterD` *(set water to new levels)*
  `if(a_kinetic == true)`
- `d_erodeWaterKinetic`
  `if(a_hydraulic == true)`
- `d_erodeWaterHydraulic`
- `d_erodeWaterSemiLag1` *(look behind advection and bilinear interpolation)*
- `d_erodeWaterSemiLag2` *(set fields to new values)*

3.7.1 Water Simulation

The first thing of all is adding the water entering the system:

```
float val = 0;
for (unsigned int i = 0; i < DPOLY_SPR_LEN; ++i) {
   val += DSPRINKLER_STR * d_gaussFalloff(c, r, POLY_SPR[i*3] * 2,
          POLY_SPR[i*3+2] * 2, DSPRINKLER_RADIUS);
}
WATER[cudaindex] += val;
```

```
/**thermal erodes a single layer of the terrain. Returns true if the
material in the layer has been sufficient and no further layer shall
be processed.*/
__device__
bool d_evalLayer(float* myh, float* other, bool* other_valid,
                  float layer_volume, float thermal_rate, float angle,
                  bool * out_valid, float * out_sum, float * out_maxdif,
                  float * out_target) {
bool retval = true;
*out_sum = 0;
*out_maxdif = 0;
//evaluate neighbours
for (unsigned char i = 0; i < 8; ++i) {
   float diff = *myh - other[i];
   //condition for thermal erosion
   if (other_valid[i] && diff>0 &&
       atan2f(diff, CELL_WIDTH) > (angle * PI) / 180) {
      //storing the sum of differences
      *out_sum += diff;
      out_valid[i] = true;
      //storing the maximal difference
      if (diff > *out_maxdif) {
          *out_maxdif = diff;
      }
   } else {
      out_valid[i] = false;
   }
}
//distribute material
float amount = thermal_rate * *out_maxdif / 2;
//check if enough material is present for our cell.
if (amount > layer_volume) {
    amount = layer_volume;
    retval = false;
}
//stop if nothing is being shifted
if (amount == 0) {
   return true;
}
//distribute the available material proportionally among valid neighbors.
for (unsigned char i = 0; i < 8; ++i) {
   if (out_valid[i]) {
   out_target[i] += amount * ((*myh - other[i]) / *out_sum);
   }
}
out_target[8] += amount;
//reduce the height for the next step.
*myh -= amount;
return retval;
}
```

Figure 3.12. Device subroutine eroding a single terrain layer.

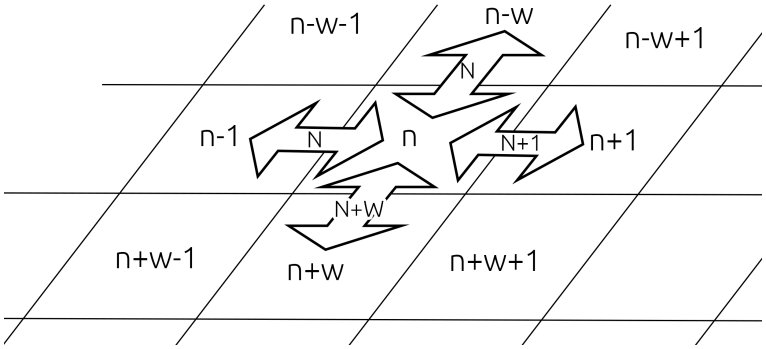

Table 3.2. Diagram showing the indexing of pipes and cells

Then we initialize our `idx[]` list together with validity check. We do not need to check whether the pipes themselves have a valid index, because each pipe connecting our cell to another valid one is bound to be valid as well.

```
const int idx[4] = { cudaindex-1, cudaindex-DW,
                     cudaindex+1, cudaindex+DW };
bool valid[4] = { idx[0] >= 0, idx[1] >= 0,
                  idx[2] < DW*DH, idx[3] < DW*DH };
const int dir[4] = { -1, -1, 1, 1 };
const int pipeidx[4] = { cudaindex - 1, offset + cudaindex - DW,
                         cudaindex, offset + cudaindex };
float dif[4] = { 0,0,0,0 };
```

Here comes a big efficiency boost. Since we established that the difference in pressure will be calculated the same looking from either direction unlike [4] we do not store pipe flow value in both directions. Instead we only allow the higher of the two cells to pass acceleration into the pipe. In the case that both cells have the same height the acceleration field is initialized to 0 and neither one of them needs to write to it.

```
float myh = SUMS[cudaindex] + WATER[cudaindex];
for (unsigned char i = 0; i < 4; ++i) {
   if (valid[i]) {
      dif[i] = myh - SUMS[idx[i]] - WATER[idx[i]];
      if (dif[i] < 0) { //do not write into higher slots.
         dif[i] = 0;
         valid[i] = false;
      }
   }
}
```

In the next step we add the calculated accelerations to water speeds. Then we look at the source cell and scale down the speeds based on the amount of water present. Without this precaution the terrain would keep its speed even if no water was present. Upon reentry water would build up into walls and swirl in unjustified ways.

Once the pipe speeds are up to date it is time to start moving the water mass. Here, we have to be careful and account for the fact that a cell may not hold enough water in it to satisfy all the demand. Similarly to the thermal material distribution step we sum the requests and the proceed to distribute the water proportionally.

```
if (amount > WATER[cudaindex]) {
   amount = WATER[cudaindex];
```

```
}
amount /= 2;
   for (unsigned char i = 0; i < 4; ++i) {
      if (pipevalid[i]) {
      dif[i] = amount * (dif[i] / sum);
      d_extra[idx[i] * 5 + 1 + i] = dif[i];
   }
}
d_extra[extraidx] = amount;
//calculate overal cell flows for use in kinetic hydro-erosion
WATER_CELL_VERT[cudaindex] = dif[3] - dif[1];
WATER_CELL_HOR[cudaindex] = dif[2] - dif[0];
```

We are again using stride of 5 with 4 indexes for outgoing water and the last one for the cell to store expelled material internally. Afterwards we also compute per-cell velocities, since the simulation algorithm only records the pipe velocity.

In the last kernel we sum up the differences calculated previously. To address the border conditions we set water, sediment and regolith levels along the border to 0. After this we apply evaporation to all non-border cells and finish up by updating the average water level – also necessary for the upcoming erosion equation.

```
if (isBorder(c, r, DW, DH)) {
   WATER[cudaindex] = 0;
   SEDIMENT[cudaindex] = 0;
   REGOLITH[cudaindex] = 0;
} else {
   float evaporate = DEVAPORATION/1000;
   if (evaporate < 0) { evaporate = 0; }
   WATER[cudaindex] += sum - evaporate;
   if (WATER[cudaindex] < 0) { WATER[cudaindex] = 0; }
}
WATER_LAST[cudaindex] = (WATER[cudaindex] + WATER_LAST[cudaindex]) / 2;
```

3.7.2 Kinetic and Hydraulic Erosion

In the case of kinetic erosion we place a minimal value on the angle condition. This ensures that the kinetic erosion effect isn't zero on perfectly flat surfaces.

```
const float velocity = sqrt(WATER_CELL_VERT[cudaindex] *
                            WATER_CELL_VERT[cudaindex] +
                            WATER_CELL_HOR[cudaindex] *
                            WATER_CELL_HOR[cudaindex]);
//set minimal transport capacity angle.
float transportCapacity;
if (SLOPE_SIN[cudaindex] < 0.1) {
   transportCapacity = velocity * 0.1 * 0.01;
} else {
   transportCapacity = velocity * SLOPE_SIN[cudaindex] * 0.01;
}
```

Afterwards we repeat a similar process to how we pealed away at the layers during the thermal erosion.

```
if (SEDIMENT[cudaindex] > transportCapacity) {
   //deposit
   float delta = SETTLE*(SEDIMENT[cudaindex] - transportCapacity);
```

```
    DUST[cudaindex] += delta;
    SEDIMENT[cudaindex] -= delta;
} else {
    //erode
    float delta = DISSOLVE*(transportCapacity - SEDIMENT[cudaindex]);
    //start removing material from the terrain.
    float remaining = delta - SEDIMENT[cudaindex];
    for (int i = DLEN - 1; i >= 0; --i) {
        const float request = MDATA[MATERIAL[i]]MHYDRO * remaining;
        if(request > dd_terrainData[i][cudaindex]) {
            dd_terrainData[i][cudaindex] = 0;
            remaining -= request;
        } else {
            dd_terrainData[i][cudaindex] -= request;
            remaining -= request;
            break;
        }
    }
    SEDIMENT[cudaindex] += delta-remaining;
}
```

The hydraulic erosion utilizes the same model. the only difference is that the capacity is dictated by a different constant and instead of sediment we store to regolith.

3.8 Material Transport

In the first step of the material transport we locate the previous location and then floor it to the nearest whole cell. the difference between the floored and non-floored coordinate denotes the weight. If we were to use a grid with a cell size other than 1 we would have to divide:

```
const float oldx = c - (WATER_CELL_HOR[cudaindex]);
const float oldy = r -(WATER_CELL_VERT[cudaindex]);
const int foldx = floorf(oldx);
const int foldy = floorf(oldy);
float wx = oldx - foldx;
float wy = oldy - foldy;
```

With the weights we sum up the amounts of each respective component in a bilinear fashion with some validity check before hand. In data passing section there is a missing cell compensation step. The logic behind it is that if we are missing a cell in our um, the weights do not technically add up to 1 and this would artificially decrease the mass of material around the edges. Notice that we are offsetting the regolith values by the length of the terrain in our temporary array.

```
float div;
if (valid[0]) { amountSediment += SEDIMENT[idx[0]] * (1 - wy)*(1 - wx);
                amountRegolith += REGOLITH[idx[0]] * (1 - wy)*(1 - wx); }
if (valid[1]) { amountSediment += SEDIMENT[idx[1]] * (1 - wy)*(wx);
                amountRegolith += REGOLITH[idx[1]] * (1 - wy)*(wx); }
if (valid[2]) { amountSediment += SEDIMENT[idx[2]] * (wy)*(1 - wx);
                amountRegolith += REGOLITH[idx[2]] * (wy)*(1 - wx); }
if (valid[3]) { amountSediment += SEDIMENT[idx[3]] * (wy)*(wx);
                amountRegolith += REGOLITH[idx[3]] * (wy)*(wx); }
```

```
for (unsigned int i = 0; i < 4; ++i) {
   if (valid[i]) {
      ++div;
   }
}
//compensate for missing cells
if (div > 0) {
   amountSediment *= 4 / div;
   d_extra[cudaindex] = amountSediment;
   d_extra[DW*DH+cudaindex] = amountRegolith;
}
```

The second step is only setting values from `d_extra[]` to either regolith or sediment values. This concludes the description of erosion techniques.

Chapter 4
Testing and Conclusion

4.1 Testing

In order to evaluate our application we put together an assignment for our testers.

4.1.1 Testing Assignment

They were tasked with recreating this provided landscape 4.1 the best they could with the tools available. Their responses are listed in C. Th times taken were measured. After the test they were asked to score the application on a scale from (1-5) in the following categories:

- How easy was the presented task?
- To what extent were you satisfied with your result?
- How intuitive did you find the controls in general?
- How easy was it to read the options?
- How easy was it to navigate the scene?
- To what extent did you feel in control of the editing?
- How clear were the instructions?
- How would you rate your overal enjoyment compared to other similar tools?

And additionally to answer these open ended questions:

- What was the most engaging part of the experience?
- What was the least pleasant part of the experience?
- Was there a feature you would welcome?
- Was there a feature you struggled to grasp initially?
- Do you have any additional remarks or suggestions?

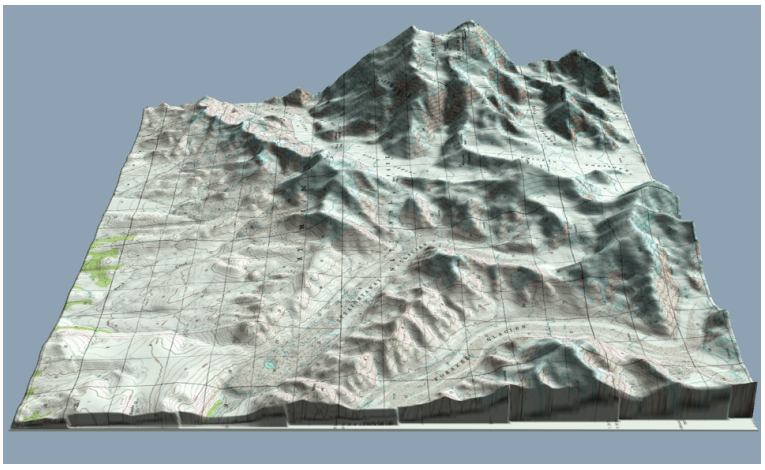

Figure 4.1. Reference terrain presented to the testers.

4.1.2 Testing Feedback

Some of our testers received a very early build of the application and experienced some crashes during their testing. However, their early attempts helped us improve the compatibility and ultimately fix all issues with the stability. In the second wave of tests we have discovered that users prefer to control the camera with either mouse or the keyboard. The AWSD and mouse control scheme comes from video game tools and gameplay and is intuitive to those users who are familiar with the fps genre. Users who play rhythm and strategic games had issues using both keys and mouse to navigate the scene. This lead to discarding the idea of having the mouse wheel change the camera speed and transitioning it to letting it zoom in and out instead. With this, the drag and look around function, the user is fully capable of navigating the scene without a keyboard which has proven to help inexperienced 3D application users a great deal.

Overall, the application turned out to have a much higher skill ceiling than it first appeared. In the hands of tester C - one of our non-programmer testers - even the process of navigating the scene became a herculean task. It is debatable to what extent it is the task of our application to teach fundamental 3D space navigation to complete beginners, however during the process the tester successfully employed the rest of the utilities provided from a static perspective, proving that the tool control itself is intelligible.

The testers generally stated that they would welcome more refined sculpting options in order to achieve. A height fixating material brush was suggested which would elevate a ridge, similar to what was discussed as *smart scribbles* at 2.1.

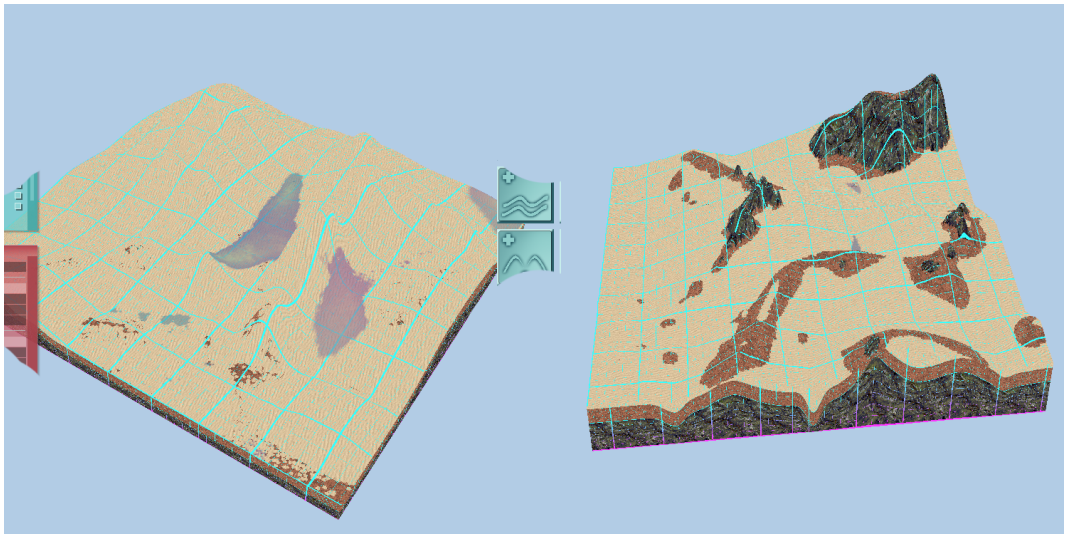

Figure 4.2. Terrains modeled by tester E and D respectively.

4.2 Evaluation

We have successfully developed an application fulfilling all the goals that we set out to do. Combining the elements of GPU parallel computing, OpenGL rendering and multi-threaded application has presented a number bottlenecks and challenges along the way, which we have overcome. Thanks to the testing feedback we managed to pinpoint areas where outside users may feel challenged and made a number of timely adjustments to further improve the experience.

References

[1] X. Mei, P. Decaudin, and B. G. Hu. *Fast Hydraulic Erosion Simulation and Visualization on GPU.* In: *Computer Graphics and Applications, 2007. PG '07. 15th Pacific Conference on.* 2007. 47-56.

[2] Irakli V. Gugushvili, and Nickolay M. Evstigneev. *Semi-Lagrangian Method for Advection Equation on GPU in Unstructured R3 Mesh for Fluid Dynamics Application.* In: 2012.

[3] "Donald Knuth". *"Knuth: Computers and Typesetting".*
`"http://www-cs-faculty.stanford.edu/~uno/abcde.html"`.

[4] Ondřej Šťava, Bedřich Beneš, Matthew Brisbin, and Jaroslav Křivánek. *Interactive Terrain Modeling Using Hydraulic Erosion.* In: *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation.* Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008. 201–210. ISBN 978-3-905674-10-1.
`http://dl.acm.org/citation.cfm?id=1632592.1632622`.

[5] Darby Stephen E., Alabyan Andrei M., and Van de Wiel MarcoJ. Numerical simulation of bank erosion and channel migration in meandering rivers. *Water Resources Research.* 38 (9), 2-1-2-21. DOI 10.1029/2001WR000602.

[6] B. Benes, and R. Forsbach. *Layered data representation for visual simulation of terrain erosion.* In: *Proceedings Spring Conference on Computer Graphics.* 2001. 80-86.

[7] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. *The Synthesis and Rendering of Eroded Fractal Terrains.* 1989.

[8] Keenan Crane, Lgnacio Llamas, and Sarah Tariq. *Real-Time Simulation and Rendering of 3D Fluids.* Addison-Wesley Professional, 2007. ISBN 0321515269.

[9] Lance M. Leslie, and R. James Purser. Three-Dimensional Mass-Conserving Semi-Lagrangian Scheme Employing Forward Trajectories. *Monthly Weather Review.* 1995, 123 (8), 2551-2566. DOI 10.1175/1520-0493(1995)123¡2551:TDMCSL¿2.0.CO;2.

[10] *FreeGLUT.*
`http://freeglut.sourceforge.net/`.

[11] *AntTweakBar.*
`http://anttweakbar.sourceforge.net/`.

[12] *CImg.*
`http://cimg.eu/`.

[13] *Nvidia CUDA Devtalk.* 2009.
`https://devtalk.nvidia.com/default/topic/411149/device-function-library-how-to-make-a-lib-of-device-functions/`.

Appendix A
Specification

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hrůša David** Personal ID number: **456910**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Interactive Erosion

Bachelor's thesis title in Czech:

Interaktivní eroze

Guidelines:

The aim of this thesis is to implement three different types of erosion and develop a user application to work with them. The deformations come in the form of thermal, hydraulic and river based for a terrain model described on a square grid of vertical vertex coordinates and implemented in a parallelized fashion with the use of Nvidia CUDA.
The application is to provide a brush painting interface that applies the chosen erosion type to the terrain in real time and is convenient to use.

Bibliography / sources:

[1] Bedrich Benes, Rafael Forsbach - Layered Data Representation for Visual Simulation of Terrain Erosion - IEEE, CS Washington 2001
[2] Arnaud Emilien, Ulysse Vimont, Marie-Paule Cani, Pierre Poulin, Bedrich Benes - World Brush: Interactive Example-based synthesi sof procedural virval worlds - ACM New York, NY USA 2015
[3] Ondřej Šťava, Bedřich Beneš, Matthew Birsbin, Jaroslav Křivánek - Interactive terrain modeling using hydraulic erosion - Dublin, Ireland 2008

Name and workplace of bachelor's thesis supervisor:

Bedřich Beneš, Ph.D., Department of Computer Graphics Technology, Purdue University, USA

Name and workplace of second bachelor's thesis supervisor or consultant:

prof. Ing. Jiří Žára, CSc., Dept. of Computer Graphics and Int., FEE

Date of bachelor's thesis assignment: **10.01.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Bedřich Beneš, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Appendix B
Manual

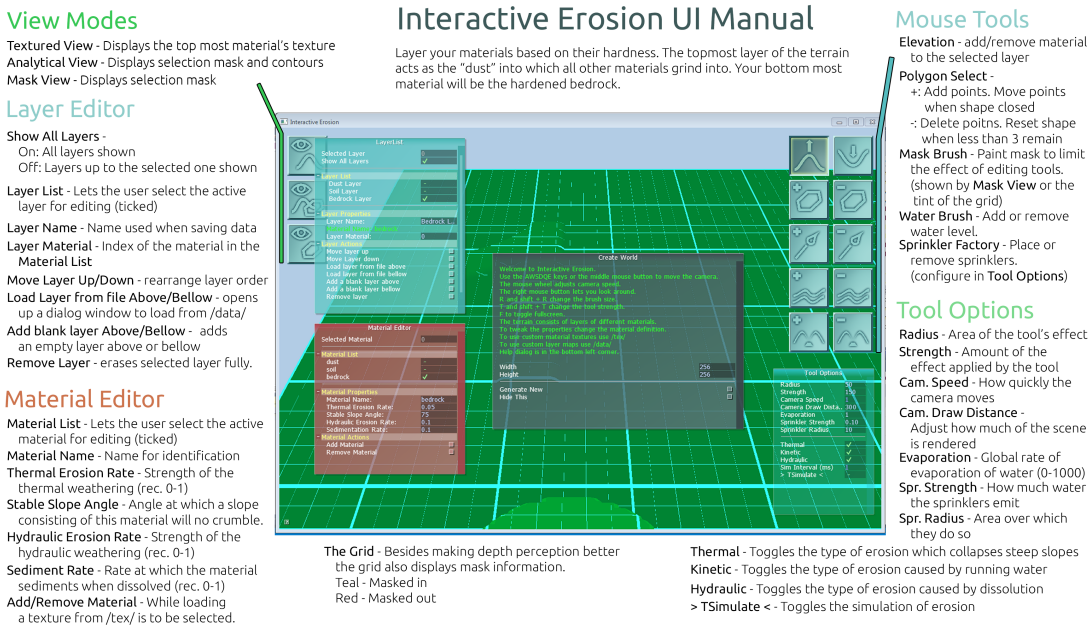

Figure B.1. Interactive Erosion UI Manual

action	description
right click (hold)	camera look around
left click	use selected tool
wheel scroll	zoom camera
shift + wheel scroll	camera movement speed

Table B.1. List of mouse controls

key		description
W	S	camera forward / backward
A	D	camera left / right
Q	E	camera up / down
R	shift + R	increase / decrease tool radius
T	shift+ T	increase / decrease tool strength
F		toggle fullscreen
	Spacebar	toggle erosion simulation

Table B.2. List of key controls

Appendix C
Tester Feedback

C.1 Point Ratings

Question	A	B	C	D	E
How easy was the presented task?	2	4	3	4	3
To what extent were you satisfied with your result?	3	4	5	2	3
How intuitive did you find the controls in general?	3	4	4	4	4
How easy was it to read the options?	2	5	3	5	5
How easy was it to navigate the scene?	4	4	1	4	3
To what extent did you feel in control of the editing?	4	5	5	3	3
How clear were the instructions?	4	4	3	5	4
How would you rate your overal enjoyment compared to other similar tools?	3	.	5	5	4

Figure C.2. Tester ratings on a scale (1 little/hard - 5 a lot/easy)

C.2 Open-ended Questions

- Q1 - What was the most engaging part of the experience?
- Q2 - What was the least pleasant part of the experience?
- Q3 - Was there a feature you would welcome?
- Q4 - Was there a feature you struggled to grasp initially?
- Q5 - Do you have any additional remarks or suggestions?

	Build version: 1.0 Time taken: 1 hour Former professional UI designer. Experienced programmer.
Q1	Painting rock formations quickly and fluidly.
Q2	Crashing while attempting to simulate, Trying to determine how to use each material type.
Q3	Tooltips over buttons would make it much easier to understand their function.
Q4	I still don't know what the second tool on the right does, looks like a waypoint tool.
Q5	A more dynamic random map generation would be nice. Be able to just open and simulate immediately.

Figure C.3. Background and open-ended feedback of tester A.

	Build version: 2.0 Time taken: 5 minutes Computer Science student. Experienced programmer.
Q1	It's really cool shaping the landscape and seeing it evolve, erode and respond to its surroundings in real time. Feels that in quite a short time, the process can be picked up.
Q2	The controls are a bit clunky - the camera, hotkeys and tool descriptions could work a bit more smoothly. We did however discuss changes that will make them a lot better. Also, the analytical and mask views can be oversaturated to the point of details being hard to see - maybe tone that a bit down if it's possible.
Q3	VR and AR controls :)
Q4	I haven't yet used the polygon select and mask tools - they might be easy to pick up, but I can't judge, as I didn't feel the need to use them.
Q5	Very cool piece of sim software - iron out the control/presentation quirks and it'll be even better.

Figure C.4. Background and open-ended feedback of tester B.

	Build version: 3.0 Time taken: 10 minutes Graphic Design and Marketing student.
Q1	Playing around with the water.
Q2	Controlling the camera. I am not very good with 3D editing tools.
Q3	Better camera control.
Q4	Camera control.
Q5	Make the manual more fun.

Figure C.5. Background and open-ended feedback of tester C.

	Build version: 2.0 Time taken: 5 minutes Computer Science student. Experienced programmer.
Q1	thinking through how to design the bedrock so it would erode into the shape i want
Q2	failing to predict water flow
Q3	setting progressively harder edges on the tools
Q4	yes, i failed to understand masking at first, leading me to believe the application was broken
Q5	no

Figure C.6. Background and open-ended feedback of tester D.

	Build version: 3.0 Time taken: 15 minutes Professional Illustrator.
Q1	Constructing the individual material layers.
Q2	Not having a maximal tool height option and mouse wheel not zooming.
Q3	Many game engines offer the ability to level terrain or to drag out ridges at an even height.
Q4	I would prefer to have the camera controls all dedicated to either the mouse or the keyboard. Also the texture view could use the same contour treatment as the second mode.
Q5	Overall, the application has a lot of convenient traits. I would welcome more keyboard shortcuts.

Figure C.7. Background and open-ended feedback of tester E.

Appendix D
Device Memory Mapping

```
#define DW dd_intParams[0] //Width of the terrain
#define DH dd_intParams[1] //Height of the terrain
#define DIDX dd_intParams[2] //Index of the selected layer
#define DLEN dd_intParams[3] //Length of the SimLayer list
#define DPOLY_SEL_LEN dd_intParams[4] //Length of the polygon
#define DPOLY_SPR_LEN dd_intParams[5] //Length of the sprinklers

#define DSTR dd_floatParams[0] //Tool strength
#define DRAD dd_floatParams[1] //Tool radius
#define DX dd_floatParams[2] //Cursor x
#define DY dd_floatParams[3] //Cursor y
#define DZ dd_floatParams[4] //Cursor z
#define DSPRINKLER_STR dd_floatParams[5] //Sprinkler strength
#define DSPRINKLER_RADIUS dd_floatParams[6] //Sprinkler radius
#define DEVAPORATION dd_floatParams[7] //Evaporation rate

#define DUST dd_terrainData[DLEN - 1] //upper most layer
#define WATER dd_terrainData[DLEN] //current water level
#define MASK dd_terrainData[DLEN+1] //selection mask
#define SUMS dd_terrainData[DLEN+2] //sum of heights in each cell
#define WATER_LAST dd_terrainData[DLEN+3] //average of water levels
#define REGOLITH dd_terrainData[DLEN+4] //regolith level
#define SEDIMENT dd_terrainData[DLEN+5] //sediment level
#define MISCOBJ dd_terrainData[DLEN+6] //unused
#define WATER_VERT dd_terrainData[DLEN+7] //verttical pipe speeds
#define WATER_HOR dd_terrainData[DLEN+8] //horrizontal pipe speeds
#define WATER_CELL_VERT dd_terrainData[DLEN+9] //per-cell vert. spd
#define WATER_CELL_HOR dd_terrainData[DLEN+10] //per-cell hor. spd
#define SLOPE_SIN dd_terrainData[DLEN+11] //sin of the max slope
#define POLY_SEL dd_terrainData[DLEN+12] //list of polygon vertexes
#define POLY_SPR dd_terrainData[DLEN+13] //list of sprinkler vertexes

//material of the selected layer
#define CUR_MATERIAL d_materialData[d_materialIndex[DIDX]]
#define MATERIAL dd_materialIndex //layer -> material index
#define MDATA dd_materialData //list of materials
#define MTHERMAL [0] //thermal erosion rate
#define MANGLE [1] //talos angle
#define MHYDRO [2] //hydraulic erosion factor
#define MSEDIMENT [2] //rate of kinetic sattling
#define MKINETIC [3] //rate of kinetic erosion

#define CELL_WIDTH 1  //Cell width for water calculations
```

Figure D.8. List of simplified decoded names for kernel parameters.