Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Interactive Shallow-Water-Simulations in City Environments

August Anselm Eickhoff

16 February 2015

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Interactive Shallow-Water-Simulations in City Environments

Interaktive Shallow-Water-Simulationen in Stadtumgebungen

Supervisor and Advisor: Professor Nils Thuerey

Submission Date: 16 February 2015

August Anselm Eickhoff

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

Abstract

WebGL offers not only a graphics pipeline for modern web browsers, but can also be used as a simple GPGPU computation environment, making much more sophisticated web applications possible. As a proof-of-concept, *WebFlood,* a realtime, fully browser based flood simulation and visualization was implemented. It employs a semi-lagrangian approach to solving the shallow-water equations. All computation is done by GLSL shaders on the GPU. The simulation state is displayed in a 3D visualization, as part of a web page that allows user interaction with the simulation. WebFlood performs well in the classical 2D dam-break scenario of Fraccarollo and Toro (1995) and closely reproduces urban flooding behavior of Iowa City (USA), given digital elevation data. Based on its cross-platform compatibility and simple distribution, two main applications are suggested: interactive public flood information and simulation-aided education for the example of hydrology.

"Empty your mind, be formless, shapeless — like water.

Now you put water in a cup, it becomes the cup;
You put water into a bottle it becomes the bottle;
You put it in a teapot it becomes the teapot.

Now water can flow or it can crash.
Be water, my friend." – *Bruce Lee*

Acknowledgements

Thank you, Mr. Thuerey, for custom-tailoring this thesis for me and being so understanding and flexible regarding the unusual circumstances of this work.

Thank you to my parents for always supporting me and raising me to be universally interested. Thanks to my brother, for being my brother!

To my future parents-in-law, thank you for all your help and for making me feel at home!

To my future wife, I'm glad that you exist - thank you for everything.

Contents

Chapter 1

Introduction

1.1 Motivation

The first inspiration that would lead to this work were WebGL *demos* that use GLSL shaders to compute things like particle systems, flocking behavior, reaction-diffusion equations or hydraulic erosion in realtime.

Using WebGL for computation became even more interesting for me after I started developing *Citybound,* a large-scale, realistic city simulation game based on web technologies [Eic15]. As of now, all simulation in Citybound, like agent-based traffic simulation or economy/demand estimation, is computed on the CPU, in JavaScript. Since many of these sub-simulations are very parallelizable at heart, a framework for more efficient, GPU based parallel simulation would be very beneficial. This work constitutes a first step towards this goal: it explores the feasibility of doing general-purpose computation on the GPU *without* using frameworks like CUDA or OpenCL.

However, it was also important for me that this work would have a purpose of its own - that is why I tried to find a serious topic of application that would also be interesting completely independently from Citybound.

1.2 Fluid and Flood Simulation as a Topic

Fluid simulation turned out to be a very fitting topic, since it is computationally intensive, but its optimization, parallelization and implementation on GPUs is well explored (see chapter 2). Furthermore, its visualization is straightforward and interesting to look at.

The simulation of floods is one obvious application that goes beyond using fluid simulation for just visual effects or demonstrations - it has real world significance to both hydrology experts as well as laypeople who might be affected by floods. Especially for the latter

group, interaction with a complex simulation only makes sense if it is simple-to-use and easily accessible - both good arguments for an implementation of the simulation using web technologies.

The topic was then further specialized on city environments, making it more relevant for Citybound and more focused in general.

1.3 GPGPU using WebGL

Before the advent of GPGPU frameworks like CUDA and OpenCL, general purpose computation on GPUs had to be done by mapping data structures to graphics primitives and implementing algorithms as shader programs that operate on these primitives.

Although experimental web bindings for OpenCL exist as browser add-ons, it is not yet included in any browser, whereas WebGL is widely adopted - leaving general-purpose WebGL programmers in a similar situation as those before CUDA/OpenCL.

A basic introduction to such techniques will be given in section 3.2.1, for a more thorough overview I recommend the *Old-School GPGPU Tutorials* by Göddeke [Göd06].

1.4 Goal

My goal with this work was to experimentally develop a physically-based flood simulation using WebGL that produces realistic results at interactive framerates on commodity hardware.

Chapter 2

Related Work and Background

2.1 Fluid Simulation: Equations and Models

The numerical description of fluid motion is the main focus of the rich field of *Computational Fluid Dynamics* (CFD). This section will introduce the Navier-Stokes Equations - they are at the heart of almost all problems in the field. Then, a short overview of some common methods of solving them will be given. Finally, a drastic simplification of the Navier-Stokes Equations is explained, resulting in the two-dimensional Shallow-Water Equations.

2.1.1 The Incompressible Navier-Stokes Equations

Since this work is concerned with water, an almost incompressible fluid, we will only consider the incompressible Navier-Stokes equations, following [BM07]. The equations describe particle behavior at every point inside a fluid and are given by:

Incompressibility Condition

$$\nabla \cdot \vec{u} = 0 \tag{2.1}$$

Momentum Equation

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla^2 \vec{u} \tag{2.2}$$

Where $\vec{u}$ is the velocity of the fluid, ρ its density and p the pressure. $\vec{g}$ are the so called *body forces* that are applied to the whole fluid (usually just gravity). ν is the viscosity of the fluid (resistance of the

fluid to deformation during flow). ∇ is the gradient, $\nabla\cdot$ the divergence and $\nabla^2 = \nabla \cdot \nabla$ the Laplace operator.

(2.2) can be rewritten with $\frac{\partial q}{\partial t} + \vec{u} \cdot \nabla q = \frac{Dq}{Dt}$. This is called the material derivative - it combines changes of a quantity q over time with advection of the quantity along a vector field $\vec{u}$. In the case of (2.2), the advected quantity is the velocity itself. This process is also called the *self-advection* of velocity: $\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \frac{D\vec{u}}{Dt}$. The unusual-looking $\nabla \vec{u}$ is the component-wise gradient of $\vec{u}$.

Momentum Equation with material derivative

$$\frac{D\vec{u}}{Dt} + \frac{1}{\rho}\nabla p = \vec{g} + \nu\nabla^2\vec{u} \tag{2.3}$$

Since macroscopic bodies of water have low viscosity (compared to small drops, or other liquids like honey) and numerical methods for solving the Navier-Stokes equations often introduce an artificial viscosity as an artifact [BM07], it is common to drop viscosity from the Momentum Equation. The resulting *inviscid* Navier-Stokes equations are also called *The Euler Equations*:

Incompressibility Condition

$$\nabla \cdot \vec{u} = 0 \tag{2.1}$$

Inviscid Momentum Equation

$$\frac{D\vec{u}}{Dt} + \frac{1}{\rho}\nabla p = \vec{g} \tag{2.4}$$

From this simpler form, the step-wise general process of numerically solving these equations can be shown [Mül+08]: first, external forces $\vec{g}$ are applied to the system - this is usually the easiest step. More complex is solving the advection $\frac{D\vec{u}}{Dt}$. The most difficult step is to solve the pressure term $\frac{1}{\rho}\nabla p$ according to (2.1).

Many different methods have been developed in the field, all of them deal with these solution steps in different ways. Their discussion here is mostly based on [JX09].

2.1.2 Grid Based Methods

Grid Based Methods in CFD adopt the Eulerian viewpoint: changes of quantities at fixed points in space (in this case grid points) are observed over time. Fluids are then fields of quantities - such as densities, velocities and temperatures.

The Navier-Stokes equations are rewritten to:

Incompressibility Condition

$$\nabla \cdot \vec{u} = 0 \tag{2.1}$$

Momentum Equation (Eulerian)

$$\frac{\partial \vec{u}}{\partial t} = \vec{g} - (\vec{u} \cdot \nabla)\vec{u} + \nu \nabla^2 \vec{u} - \frac{1}{\rho}\nabla p \tag{2.5}$$

They are then discretized onto a grid and numerically solved for $\vec{u}$ using finite differences. (2.5) is typically decomposed into steps: external forces are introduced, velocities are advected in a forward timestep and diffused by viscosity. Finally, pressures and velocities are solved to satisfy (2.1) - this amounts to a solution of the Poisson equation and requires iteration over the whole fluid body.

The foundational work for this technique was done by Harlow and Welch [HW65], who also introduced the staggered *MAC-Grid*, which improves finite differences (by defining velocities at cell boundaries and pressures at cell centers), and the *Marker and Cell Method*, which uses virtual particles that are advected by the velocity field. Their only use is to mark filled/empty cells (to keep track of the fluid surface).

The first to animate fluids by solving the full three-dimensional Navier-Stokes equations, building on the work of Harlow and Welch, were Foster and Metaxas [FM96], they used an iterative relaxation scheme to solve for incompressibility.

2.1.3 Particle Based Methods

Particle Based Methods represent the Lagrangian viewpoint of CFD. Here, individual particles are followed along their motion through the fluid over time.

The idea to represent simulated fluids by particles and the notion of particle systems in general was introduced by Reeves [Ree83].

For particles, the Navier-Stokes Equations are written as:

Incompressibility Condition

$$\nabla \cdot \vec{u} = 0 \tag{2.1}$$

Momentum Equation (Lagrangian)

$$\frac{D\vec{u}}{Dt} = \vec{g} + \nu\nabla^2\vec{u} - \frac{1}{\rho}\nabla p \tag{2.6}$$

The left side of (2.6) can be seen as the acceleration of a particle, caused by the net force that is described by the terms on the right-hand side [JX09].

Advection in the Lagrangian viewpoint is obvious and exact [Mon05] but the reconstruction of of fluid quantities from particles is nontrivial - in particular the gradient ∇ and Laplacian operator ∇^2 need to be well defined in an irregularly discretized fluid made out of particles.

Smoothened Particle Hydrodynamics (SPH) is an important method to achieve this. It came from the field of Astrophysics, where it was introduced by Gingold and Monaghan [GM77]. Monaghan showed how it can be applied to the simulation of liquids in [Mon05].

In SPH, a fluid is represented by i particles with positions x_i, masses m_i, densities ρ_i and attributes A_i. It defines how to compute a smooth continuous field $A(x)$ for these attributes, as well as its gradient and laplacian, from the values of particles, weighted by a kernel function $W(r, h)$:

$$A(x) = \sum_i m_i \frac{A_i}{\rho_i} W(x - x_i, h) \tag{2.7}$$

$$\nabla A(x) = \sum_i m_i \frac{A_i}{\rho_i} \nabla W(x - x_i, h) \tag{2.8}$$

$$\nabla^2 A(x) = \sum_i m_i \frac{A_i}{\rho_i} \nabla^2 W(x - x_i, h) \tag{2.9}$$

$W(r, h)$ is typically radially symmetric and has finite support.

The main problem of employing SPH is to enforce the incompressibility constraint of (2.1) since it does not deal with pressure directly. This is the topic of diverse ongoing research (see [JX09, p. 3]).

2.1.4 Hybrid and Other Approaches

In Eulerian methods, the forward time integration of velocity advection is numerically unstable. Stam [Sta99] introduced the Lagrangian viewpoint into the advection step of an otherwise Eulerian method: at each cell, a virtual particle is traced backwards in time along $\vec{u}$. The Eulerian values at this origin position are then used as the new values in the current cell. This is also called the *method of characteristics.*

Additionally Stam solved diffusion/viscosity implicitly and employed a more accurate solver for pressure correction than [FM96]. All of these measures result in unconditionally stable numerical behavior even for large time steps, which led to this method becoming a standard framework for fluid animation.

Building on the semi-lagrangian method of Stam, so called *level set methods* were introduced by Foster and Fedkiw [FF01], which are able to keep track of very complex water surfaces.

An interesting, completely different approach from everything discussed so far is the *Lattice Boltzmann Method* (LBM). Like Eulerian Methods, it is grid based, but models macroscopic fluid behavior by averaging equations that describe microscopic statistical kinetic models. The LBM converges to the Navier-Stokes equations and has many advantages that make it more and more popular. For a detailed explanation and discussion see [Mül+08, p. 59] and [JX09, p. 4].

2.1.5 The Shallow Water Equations

A specialized and simpler two-dimensional version of the Navier-Stokes equations can be obtained using the *Shallow Water Assumptions* and integrating over a water column with a fixed base area, resulting in the so called *Shallow Water Equations* (SWE).

Here, only a short description of this process will be given, detailed derivations can be found in [Mül+08, p. 77-82] and [Geo06, p. 36-39].

First the Euler equations (2.1) and (2.4) are further simplified to vertical gravity as the only external force and complemented with boundary conditions for an impermeable bottom topography and dynamic free surface at the top of the water column.

Then, the central assumption is that pressure is strictly *hydrostatic* and thus given by $p = \rho g \Delta \eta$. This assumption holds for flows where the horizontal extent of the body of fluid is much larger than its depth [Geo06] (this is why these flows are called *shallow* water flows).

This has two important implications: the vertical velocity is zero everywhere and the horizontal velocities no longer vary along the vertical direction - making it possible to integrate over the height of the water column, reducing the problem to two dimensions.

In the process, the water height takes over the role of pressure and the fluid can be represented by a two-dimensional height field.

The final partial derivative form of the Shallow Water Equations can be given by:

$$\frac{\partial \eta}{\partial t} + \vec{v} \cdot \nabla \eta = -\eta \nabla \cdot \vec{v} \tag{2.10}$$

$$\frac{\partial \vec{v}}{\partial t} + \vec{v} \cdot \nabla \vec{v} = g \nabla h \tag{2.11}$$

Where $\vec{v}$ is the two-dimensional horizontal velocity, η is the water height above the bottom topography b, $h = b + \eta$ is the water level above zero and g is gravity.

This can again be simplified using the material derivative:

$$\frac{D\eta}{Dt} = -\eta \nabla \cdot \vec{v} \tag{2.12}$$

$$\frac{D\vec{v}}{Dt} = g \nabla h \tag{2.13}$$

2.2 Two Specializations of Fluid Simulation

Now that some general models of fluid simulation have been discussed, two specialized research fields that are more closely related to this work will be highlighted.

2.2.1 Real Time GPU Based Fluid Animation

All three models introduced (grid based, particle based, LBM) lend themselves well for implementation on the GPU, since many of the computations can be be done in parallel.

Still in the infancy of GPGPU programming, where GPUs were only used for fluid *visualization* techniques, Li, Wei, and Kaufman [LWK03] already implemented the LBM to fully simulate fluids on fixed-point graphics hardware.

After the advent of programmable fragment shaders and floating-point support of GPUs, Bolz et al. [Bol+03] mapped sparse matrices to textures and adapted the multi grid method to the GPU for fluid simulation (among other problems).

Wu, Liu, and Liu [WLL04] solve the Navier-Stokes equations using a grid-based method. They pack all simulation variables into the four color channels of a texel resulting in a more efficient fluid simulation than earlier methods - they also introduced an elegant way to represent arbitrary boundaries *inside* the simulation domain and store boundary information in texels as well.

Crane, Llamas, and Tariq [CLT07] demonstrated GPU-based realtime 3D simulation and visualization of different fluid phenomena like liquids, smoke and fire based on level-set methods.

One of the first to make use of an abstracted GPGPU framework for the LBM was Tölke [Töl09], who implemented it on Nvidia's CUDA framework.

Because of their inherent irregularity, particle based methods tend to be more difficult to implement on GPUs. Nonetheless Kolb and Cuntzn [KC05] developed a mapping of SPH on the GPU based on OpenGL (cross-platform) and Cg (which compiles to DirectX shaders).

Krog and Elster [KE10] achieved SPH simulation of 256k particles at realtime speeds based on a CUDA implementation.

The Shallow-Water Equations and heightfield based approaches are even more attractive for GPGPU, since their two-dimensional nature makes them not only computationally less expensive than full 3D models, but also maps well onto GPU hardware and graphics primitives.

Early work in this area was done by Hagen et al. [Hag+05], who implemented solver for the SWE using OpenGL and Cg. They observed a speed-up of 15-30 times compared to CPU implementations of the SWE. Their model already supports dry areas and complex bottom topographies.

By porting a well-balanced finite volume scheme of the SWE to CUDA, Asunción, Mantas, and Castro [AMC11] achieved significant speedups compared to earlier shader-based methods.

Recent research combines the heightfield with full 3D models to take advantage of the simplicity and realtime speed of the SWE while still simulating details that cannot be represented by a heightfield. Examples for this are the extensions of SWE to include bubbles and foam [Thü+07b] or breaking waves [Thü+07a], to put 3D grid cells on top of heightfield-like *tall cells* [CM11], or to couple a heightfield with a 3D grid as well as particles for large scale, yet detailed fluid phenomena [CMK14].

2.2.2 Shallow-Water Equations in Flood Simulation

Unlike the field of Fluid Animation, which is more concerned with simulation speed, the simulation of floods requires physically accurate models. Although off-line simulation is acceptable here, the simulation domains usually span such large scales that a simplification from three-dimensional flows to depth-averaged two-dimensional models such as the SWE is *required* to make them tractable at all [Geo10].

In addition, the particular topography of a target area is often taken into account to optimize the simulation with highly irregular meshes - allowing efficient handling of scenarios that contain both small and large scales, at the cost of a much more complicated formulation of the SWE. This tradeoff is discussed in detail in [Kim+14].

Another way of addressing this problem of multiple scales is adaptive refinement of a regular grid into smaller subgrids. Liang et al. present such a method in [Lia+08] - this paper incidentally also gives a good overview over the motivation, challenges and methods of flood simulation - much better than I could attempt with my limited understanding of the field - so I kindly refer you to them.

Chapter 3

Model and Implementation

Based on my initial inexperience with fluid simulation (and its application to flood simulation), I set out on a highly iterative process of implementing a basic shallow-water simulation, observing its shortcomings and overcoming them by a combination of research and empirical experimentation.

This rapid iteration and very practical learning and implementation process was made possible largely by the dynamic nature of the web development environment - giving immediate feedback to changes in code (without compilation time) and allowing for a very exploratory programming style.

Nonetheless, the final model and implementation scheme was chosen at the beginning of my work. I will outline these initial choices, describe the implementation of the simulation in detail and mention modifications that became necessary. Finally, I will show how the simulation forms part of a web application.

3.1 Selection of Shallow-Water Model

3.1.1 Model Characteristics

An Eulerian grid-based method to solve the SWE was chosen, because of its straightforward mapping to graphics programming concepts and my prior familiarity with it. Following [Mül+08], a semi-lagrangian advection scheme à la Stam [Sta99] was adopted, because of its stability and intuitive simplicity.

Also for the sake of simplicity, the common practice of a staggered MAC grid was not adopted - instead all quantities are defined at cell centers. This seemed to be stable enough throughout the work.

3.1.2 Numerical Method

When looking again at the Shallow-Water Equations:

$$\frac{D\eta}{Dt} = -\eta\nabla\cdot\vec{v} \tag{2.12}$$
$$\frac{D\vec{v}}{Dt} = g\nabla h \tag{2.13}$$

it can be seen that the left sides represent advection along the velocity field $\vec{v}$ while the right sides account for additional accelerations (in the case of η, this refers to the change in water height due to in/outflow determined by the divergence $\nabla\cdot\vec{v}$ of the velocity field).

Like in [Mül+08] an explicit time integration scheme is chosen (again favoring simplicity over accuracy), resulting in the following step-wise numerical solver for the SWE:

Advection

$$\vec{v}_*[\vec{x}] = \vec{v}_t[\vec{x} - \vec{v}_t[\vec{x}]\Delta t] \tag{3.1}$$
$$\eta_*[\vec{x}] = \eta_t[\vec{x} - \vec{v}_t[\vec{x}]\Delta t] \tag{3.2}$$

Integration

$$\eta_{t+1}[\vec{x}] = \eta_*[\vec{x}] - \eta_*[\vec{x}]\nabla\cdot\vec{v}_*[\vec{x}]\Delta t \tag{3.3}$$
$$\vec{v}_{t+1}[\vec{x}] = \vec{v}_*[\vec{x}] - g\nabla h\Delta t \tag{3.4}$$

Here, $\vec{v}_t[\vec{x}]$ and $\eta_t[\vec{x}]$ represent velocity and water height at time t and position $\vec{x}$. $*$ represents intermediate values between time steps.

3.2 Implementation on the GPU

3.2.1 Basic Graphics-Based GPGPU

This section assumes knowledge of basic 3D computer graphics primitives and operations and will explain the most basic way how they can be used for GPGPU.

The first important concept to grasp is that *textures can be used to store generic data*. This requires a two-dimensional layout of the data over the texture. In our case of grid based fluid simulation (and many other problems in GPGPU) the data is already organized two-dimensionally, making this step trivial.

At each discrete *texel* the according data at this position has to be encoded into the color vector of this texel. Depending on the

texture format, this vector usually has one, three or most commonly four components, which can be integers or floating-point values (the encoding used in our case is described in the next section)

Framebuffers can be used as virtual screens to render to, and can subsequently be read from like a normal texture. This is typically used for scene-in-scene effects and reflections.

By rendering a pair of triangles that together cover a whole virtual screen (a *full-screen quad*) and using a framebuffer as the render target, we can force the currently set fragment shader to execute exactly once per pixel of the virtual screen / texel of the framebuffer.

From our GPGPU viewpoint this means that the fragment shader can perform many computations in parallel and output the results into the framebuffer, using our special color encoding.

Since the fragment shader can arbitrarily read from other textures and framebuffers, we can keep swapping out two framebuffers after each render step to achieve an infinite feedback effect (also called the *ping-pong technique*). Many numerical schemes map naturally to this, since all they do per time step is to calculate the new state of a system based on an old one and then they repeat the process.

All that needs to be done is to fill one framebuffer with some intial data (in the case of simulations initial conditions) and then to chain draw calls of fragment shaders that perform operations on this data, swapping out framebuffers in between.

3.2.2 Data Representation

The shallow-water simulation is implemented to be completely based one *simulation framebuffer*, which directly represents the simulation grid. In each texel of the framebuffer, the simulation data for one grid cell is encoded as a 4-component floating-point color: the two-dimensional velocity vector resides in the red (x) and green (y) channels, the blue channel contains water height, and the alpha channel contains terrain height.

3.2.3 Simulation DSL and Shader Prefix

In initial implementations, this mapping of simulation values to texture positions and color channels was very explicitly visible in the shader programs describing the simulation steps, in many cases needlessly obfuscating what was happening.

As an exercise towards a more declarative *domain specific* programming style for these simulation shaders, a simple *shader prefix* was created, containing only a few macros and declarations that almost completely abstract away the fact that we are dealing with color vectors inside a fragment shader (see A.1).

Mainly, it offers are the `V()`, `Vx()` and `Vy()` macros for accessing the velocity and its components inside a texel, the `T()` and `H()` macros for accessing terrain height and water height inside a texel and the helper macro `L() = T() + H()` which adds terrain height and water height to give the water level above zero.

Because this shader prefix is prepended to all simulation step shaders, its definitions are accessible to them.

3.2.4 Simulation Steps

Each update step of the simulation is divided into substeps, according to the numerical method described in section 3.1.2. Their most basic forms are now given in order of execution.

Advection of Water Height and Velocity

First, velocity and water height are advected according to (3.2): The velocity vector at the current position is read and extrapolated backwards according to the timestep, giving the origin position of where a virtual particle would have been previously.

The simulation framebuffer is then queried at this origin position and the velocity/water height there is returned as the new velocity/water height of the current cell.

```
vec4 simulationStep() {
    vec4 here = simData(pos);

    vec4 origin = simData(pos - dt * V(here));
    float newHeight = H(origin);
    vec2 newVelocity = V(origin);

    return vec4(newVelocity, newHeight, T(here));
}
```

Since no staggered grid is used, velocity and water height for each cell will be looked up from exactly the same origin position – this makes it possible to perform velocity and water height advection together, using just two framebuffer lookups per cell instead of four.

Integration of Height

Height is integrated according to (3.3). The divergence of the velocity field is calculated based on finite differences of ±1 cell size (`unit`) in either direction:

```
vec4 simulationStep() {
    vec4 here = simData(pos);
    vec4 X1 = simData(posLeft);
    vec4 X2 = simData(posRight);
    vec4 Y1 = simData(posTop);
    vec4 Y2 = simData(posBottom);

    float dVelocityX = (Vx(X2) - Vx(X1)) / (2.0 * unit);
    float dVelocityY = (Vy(Y2) - Vy(Y1)) / (2.0 * unit);
    float velocityDivergence = (dVelocityX + dVelocityY);

    float fluxArea = H(here);
    float newHeight = H(here) - fluxArea * velocityDivergence * dt;

    return vec4(V(here), newHeight, T(here));
}
```

Integration of Velocity

For the integration of velocity according to (3.4), the slope of the water level is also calculated using finite differences, making use of the `L()` macro. The additional uniform value `gravity` is declared as an input for just this step.

```
uniform float gravity;

vec4 simulationStep() {
    vec4 here = simData(pos);
    vec4 X1 = simData(posLeft);
    vec4 X2 = simData(posRight);
    vec4 Y1 = simData(posTop);
    vec4 Y2 = simData(posBottom);

    vec2 slope = vec2(L(X2) - L(X1), L(Y2) - L(Y1)) / (2.0 * unit);

    vec2 newVelocity = V(here) - gravity * slope * dt;

    return vec4(newVelocity, H(here), T(here));
}
```

3.3 Adaptation for Flood Simulation

3.3.1 Modifications for Drying/Wetting

For application of fluid simulation to flood scenarios, a good representation and handling of dry areas is required. This poses two new challenges: finding correct boundary conditions at the interface of water with the underlying terrain, and handling *wetting and drying*

properly, which describes the propagation of water into dry areas and the creation of new dry areas by receding water.

Medeiros and Hagen [MH12] give a very good overview of different approaches to wetting/drying in current research. For this work, a lot of experimentation was required to find something that works.

Definition and Handling of Dry Areas

Dry areas are defined to be all cells where the water height is equal to, or smaller than zero. The velocity in dry areas is assumed to be zero, which completely disables advection into these areas. Modifications of height and velocity integration make water transport into dry areas possible again.

Advection

In the case of advection of nothing - out of a dry area into a wet area (leading to water loss), no advection is performed. The velocity at this cell is set to zero to prevent further erroneous advection.

```
vec4 simulationStep() {
    vec4 here = simData(pos);

    if (H(here) <= 0.0) return vec4(V(here), H(here), T(here));

    vec4 origin = simData(pos - dt * V(here));
    float newHeight = H(origin);
    vec2 newVelocity = V(origin);

    if (newHeight <= 0.0) {
        newHeight = H(here);
        newVelocity = vec2(0.0, 0.0);
    }

    return vec4(newVelocity, newHeight, T(here));
}
```

An early-return is added to speed up computation in dry areas.

Reasonable Velocities at the Boundary

For the integration of velocity, more complicated modifications are necessary: the calculation of slope has to be corrected. If one or more neighboring cells are dry, the naïve water level obtained by `L(neighbor) = H(neighbor) + T(neighbor)` might end up higher than the current cell.

This would lead to nonphysical velocities pointing into the wet areas, which in turn would cause incorrect appearance of water during the height integration step (because of a constant negative velocity divergence at the boundary).

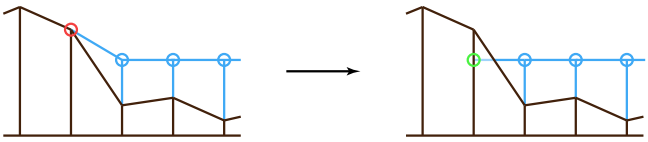


Figure 3.1: Slope Correction

Thus, in such cases, the water level at the neighboring cell is ignored and replaced by the water level of the current cell - this corresponds to a flat water surface in the real world (see figure 3.1.

Additionally, velocities that would point outwards of a wet area are set to zero, causing water to build up at wetting fronts during the height integration step. Later, water transport into dry areas will be ensured, based on the height of the built-up water. Without these measures, water would just vanish into dry areas.

```
uniform float gravity;

vec4 simulationStep() {
    vec4 here = simData(pos);

    if (H(here) < 0.0) return vec4(0.0, 0.0, H(here), T(here));

    vec4 X1 = simData(posLeft);
    vec4 X2 = simData(posRight);
    vec4 Y1 = simData(posTop);
    vec4 Y2 = simData(posBottom);

    // boundary: assume water is flat until shoreline
    float L_X1 = H(X1) < 0.0 && T(X1) > L(here) ? L(here) : L(X1);
    float L_X2 = H(X2) < 0.0 && T(X2) > L(here) ? L(here) : L(X2);
    float L_Y1 = H(Y1) < 0.0 && T(Y1) > L(here) ? L(here) : L(Y1);
    float L_Y2 = H(Y2) < 0.0 && T(Y2) > L(here) ? L(here) : L(Y2);

    vec2 slope = vec2(L_X2 - L_X1, L_Y2 - L_Y1) / (2.0 * unit);

    vec2 newVelocity = V(here) - gravity * slope * dt;

    if (H(X1) < 0.0 || H(X2) < 0.0) newVelocity.x = 0.0;
    if (H(Y1) < 0.0 || H(Y2) < 0.0) newVelocity.y = 0.0;

    return vec4(newVelocity, H(here), T(here));
}
```

Again, we add an early return, no computation is done in dry areas.

Wetting during Integration of Height

The integration of height naturally models drying, but for low velocities, residual thin layers of water can remain for a long time before drying up. In the real world, surface tension would prevent this. In order to achieve a similar effect, the flux area for each cell that determines how much water streams in or out is clamped to be at least `minFluxArea`. Otherwise, the computation for wet cells stays the same.

In contrast to the other two steps, dry cells are also handled by this step, but they don't participate in the normal simulation. Only a check is performed for them: if any neighboring cells contain water and their water level is higher than the dry cell plus a small `wettingThreshold`, the dry cell is wetted and initialized to `newlyWetHeight`. In the following time steps it will participate in the simulation.

```
const float wettingThreshold = 0.000001;
const float newlyWetHeight =   0.0000003;
const float minFluxArea = 0.01;

vec4 simulationStep() {
    vec4 here = simData(pos);
    vec4 X1 = simData(posLeft);
    vec4 X2 = simData(posRight);
    vec4 Y1 = simData(posTop);
    vec4 Y2 = simData(posBottom);

    float dVelocityX = (Vx(X2) - Vx(X1)) / (2.0 * unit);
    float dVelocityY = (Vy(Y2) - Vy(Y1)) / (2.0 * unit);
    float velocityDivergence = (dVelocityX + dVelocityY);

    float newHeight;

    if (H(here) <= 0.0) {
        if ((H(X1) > wettingThreshold
            && L(X1) > T(here) + wettingThreshold)
        ||  (H(X2) > wettingThreshold
            && L(X2) > T(here) + wettingThreshold)
        ||  (H(Y1) > wettingThreshold
            && L(Y1) > T(here) + wettingThreshold)
        ||  (H(Y2) > wettingThreshold
            && L(Y2) > T(here) + wettingThreshold)
        ) {
            newHeight = newlyWetHeight;
        } else {
            newHeight = H(here);
        }
    } else {
        float fluxArea = max(H(here), minFluxArea);
        newHeight = H(here) - fluxArea * velocityDivergence * dt;
    }

    return vec4(V(here), newHeight, T(here));
}
```

3.3.2 Modifications for Friction

In addition to wetting/drying, bed friction at the bottom of the water volume is required to achieve realistic flooding behaviour. It is commonly implemented as a *bed friction slope term* that counteracts acceleration due to water level slope. It is given by:

$$\vec{S}_f = n^2 \frac{\vec{v} \cdot |\vec{v}|}{h^{\frac{4}{3}}}$$

Where n is *Manning's Friction Coefficient*.

In the time-discretized shader code, the velocity value of the previous time step is taken as an approximation for $\vec{v}$, to keep the velocity update simple and explicit (otherwise the new velocity would reappear on the right hand side, in the friction slope term).

This approximation only caused problems when a large velocity in the previous time step resulted in a large friction slope that opposes and overpowers the terrain slope of the current time step, resulting in a very nonphysical inversion of velocities. This is prevented by limiting how much the friction slope.

```
uniform float manningCoefficient;
uniform float gravity;

vec4 simulationStep() {
    vec4 here = simData(pos);
    if (H(here) < 0.0) return vec4(0.0, 0.0, H(here), T(here));
    vec4 X1 = simData(posLeft);
    vec4 X2 = simData(posRight);
    vec4 Y1 = simData(posTop);
    vec4 Y2 = simData(posBottom);

    // boundary: assume water is flat until shoreline
    float L_X1 = H(X1) < 0.0 && T(X1) > L(here) ? L(here) : L(X1);
    float L_X2 = H(X2) < 0.0 && T(X2) > L(here) ? L(here) : L(X2);
    float L_Y1 = H(Y1) < 0.0 && T(Y1) > L(here) ? L(here) : L(Y1);
    float L_Y2 = H(Y2) < 0.0 && T(Y2) > L(here) ? L(here) : L(Y2);

    vec2 slope = vec2(L_X2 - L_X1, L_Y2 - L_Y1) / (2.0 * unit);
    float n = manningCoefficient;
    vec2 frictionSlope = V(here) * length(V(here)) * pow(n, 2.0)
                         / pow(H(here), 4.0/3.0);

    vec2 totalSlope = slope + frictionSlope;
    // make sure new slope doesn't point in other direction
    totalSlope.x = slope.x < 0.0 ?
        min(totalSlope.x, 0.0) : max(totalSlope.x, 0.0);
    totalSlope.y = slope.y < 0.0 ?
        min(totalSlope.y, 0.0) : max(totalSlope.y, 0.0);
    totalSlope.x = slope.x == 0.0 ? 0.0 : totalSlope.x;
    totalSlope.y = slope.y == 0.0 ? 0.0 : totalSlope.y;

    vec2 newVelocity = V(here) - gravity * totalSlope * dt;

    if (H(X1) < 0.0 || H(X2) < 0.0) newVelocity.x = 0.0;
    if (H(Y1) < 0.0 || H(Y2) < 0.0) newVelocity.y = 0.0;

    return vec4(newVelocity, H(here), T(here));
}
```

3.4 WebFlood as a Web Application

In addition to basic setup and scaffolding, WebFlood provides a 3D visualization and user interface for the simulation.

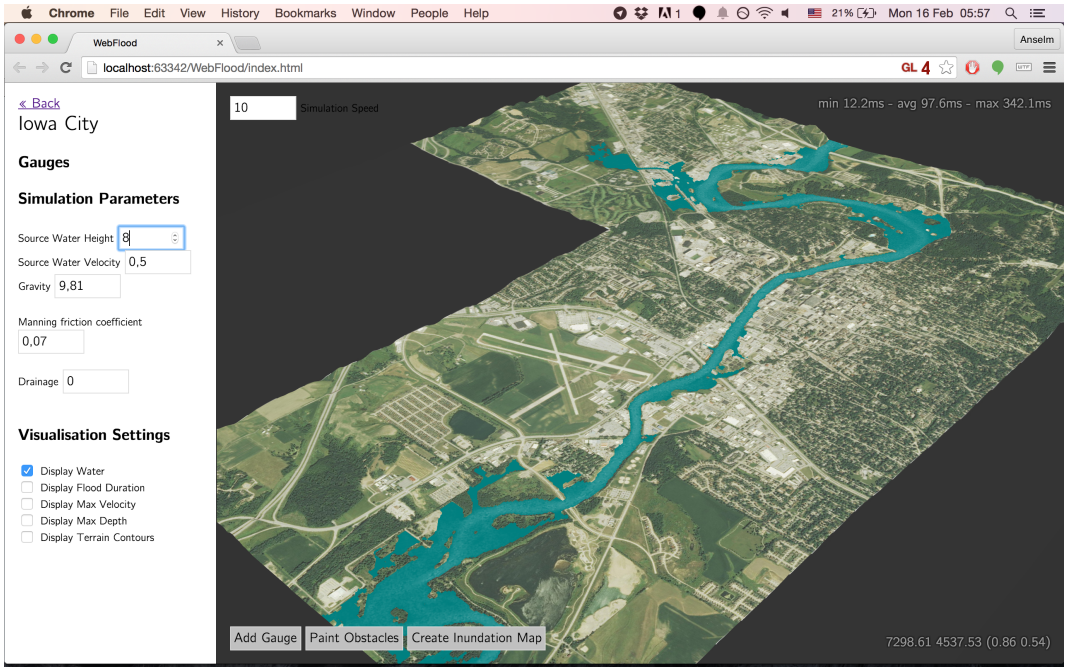


Figure 3.2: Running simulation with visualization and UI

This user interface allows loading of different scenarios, playback control for the simulation and on-the-fly changing of simulation parameters (See Figure 3.2). It offers tools for adding water gauges at arbitrary positions in the world (see Figure 3.3) and exporting their hydrographs as .svg files (in fact this is what the diagrams in section 4.1 are based on). It also offers various options to overlay the terrain with additional info, for example maximum inundation depth and terrain contours (See Figure 3.4).

WebFlood imports and exports most simulation and visualization data (for example terrain heightmaps, initial water conditions, generated inundation maps) from/to 16-bit greyscale TIFF images, which offer higher dynamic resolution than common 8-bit image formats, while still being widely supported by standard graphics software.

Scenarios and their default parameters and references to source data files are defined in a human-readable JSON file. Here, gauge locations can be predefined and complemented by reference data from literature, which will be displayed as small dots in gauge diagrams (see Figure 3.3)

WebFlood is hosted at `http://aeickhoff.github.io/WebFlood/` and its source code is available at `https://github.com/aeickhoff/WebFlood`

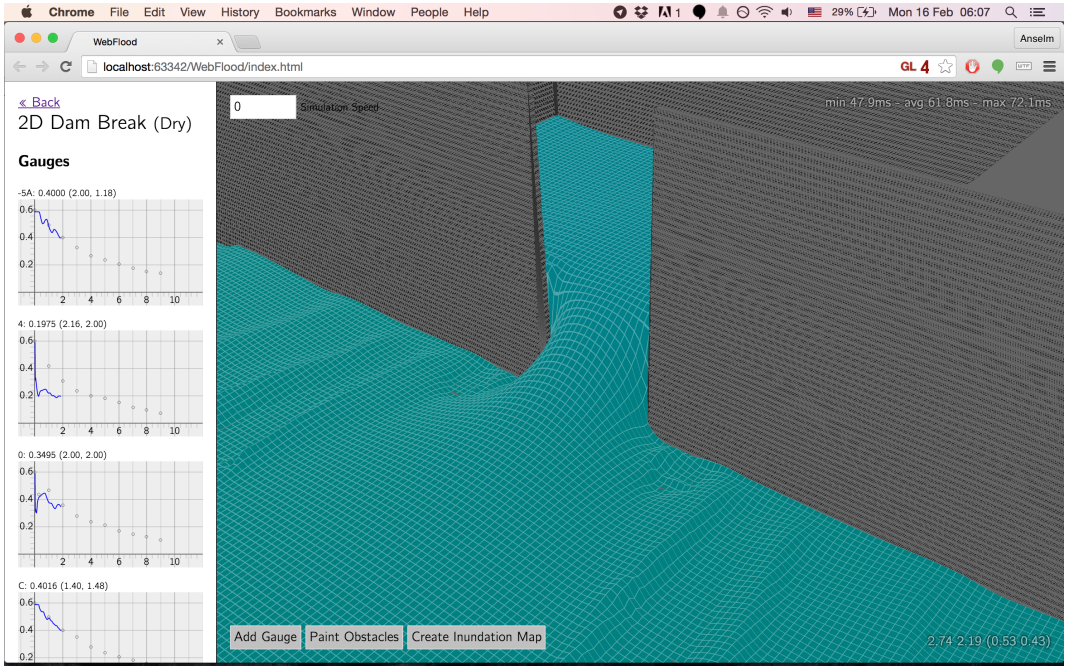


Figure 3.3: Water gauges UI

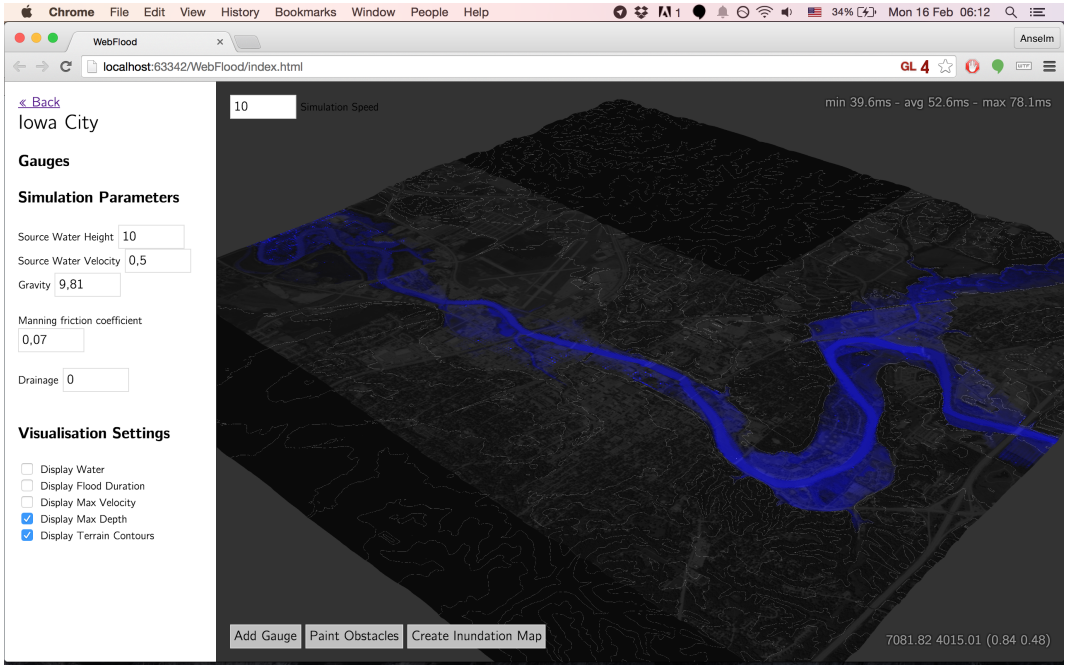


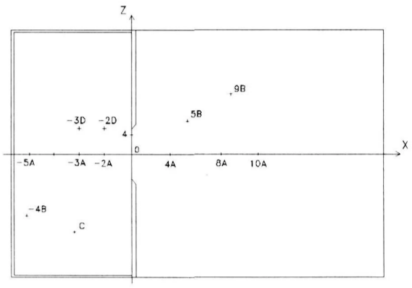
Figure 3.4: Terrain overlays

Chapter 4

Results and Validation

4.1 2D Dam Break of Fraccarollo & Toro (1995)

The two-dimensional dam break experiment by Fraccarollo and Toro [FT93] laid important empirical groundwork for validating the detailed behaviour of all kinds of fluid simulations. It was chosen as the main synthetic test case for WebFlood because of its simple setup and plentiful experimental data points to compare against.



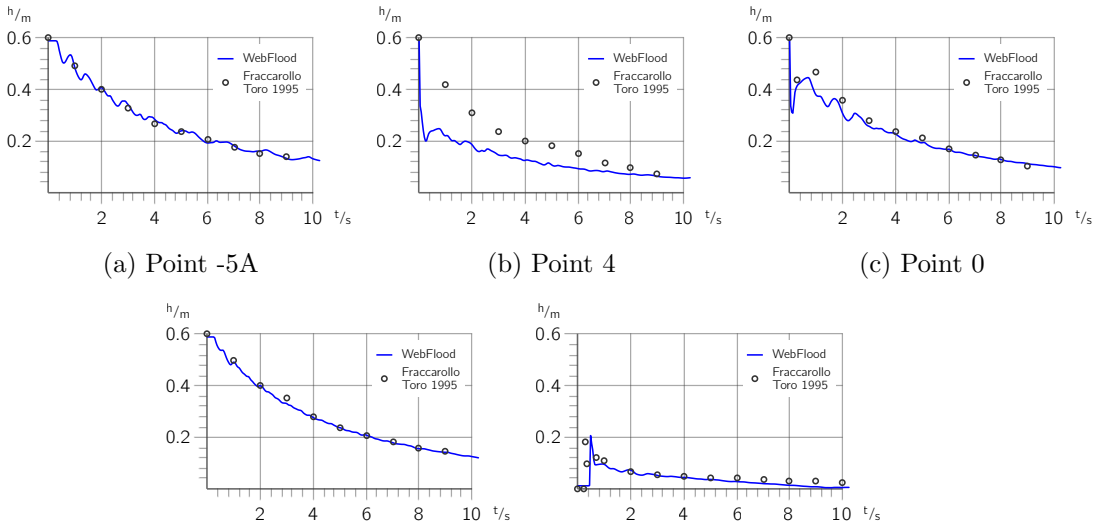
Position	0	4	-5A	-3A	-2A	4A	8A	10A	-4B	5B	9B	C	-3D	-2D
X (cm)	0	0	-82	-42	-22	32.2	72.2	102	-84.5	45.4	80.2	-52	-42	-22
Z (cm)	0	16	0	0	0	0	0	0	-50	25	45	-60	20	20

Figure 4.1: Probe setup of Fraccarollo and Toro [FT93]

A simplified subset of test points has been adopted of Lin, Lai, and Guo [LLG05], as well as the idea to test the simulation first with an already-wet floodplain.

WebFlood reproduces behaviour from the experiment well, with the exception of point 4. In [LLG05], too, a deviation at this point is observed - it is suggested that at this point (dam opening) the shallow-water assumptions are particularly inaccurate.

Later, the test was performed with a dry floodplain (see figure 4.3) to test the influence or side effects of drying/wetting, the results turned out to be identical.



(a) Point -5A (b) Point 4 (c) Point 0

(d) Point C (e) Point 8A

Figure 4.2: Water height at test points for wet floodplain

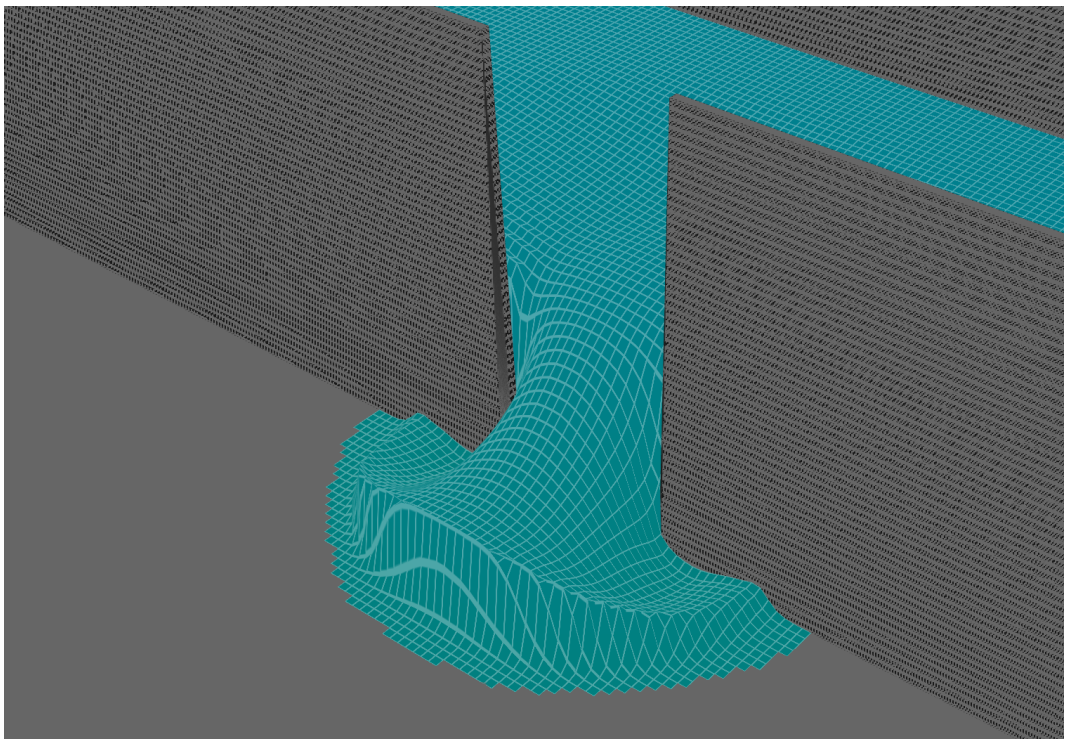


Figure 4.3: Dam break with a dry floodplain

4.2 2008 Floods in Iowa State, USA

Due to especially intense rainfall in May and June of 2008, the Iowa and Cedar rivers experienced record floods, severely affecting different municipal areas in Iowa State [USGS10].

In response to these floods, the Iowa Flood Center was established and funded by the the state of Iowa in 2009, it conducts flood related research and participates actively in projects to create flood inundation maps and high-resolution elevation maps (based on LiDAR laser scans) for the state of Iowa.

For this work, Iowa City was selected as simulation area, since Iowa River directly goes through its urban center and its hilly topography is both interesting to look at and challenging for the simulation.

4.2.1 Source Data, Processing and Possible Sources of Inaccuracy

Digital elevation data (already based on these LiDAR maps) was obtained from The National Map Viewer of the U.S. Geological Survey [NatMap].

This data was then reprojected, cropped, the elevation rescaled and converted to a 512×512px heightmap image (see Fig. 4.4) using utilities from [GDAL] and Adobe Photoshop.

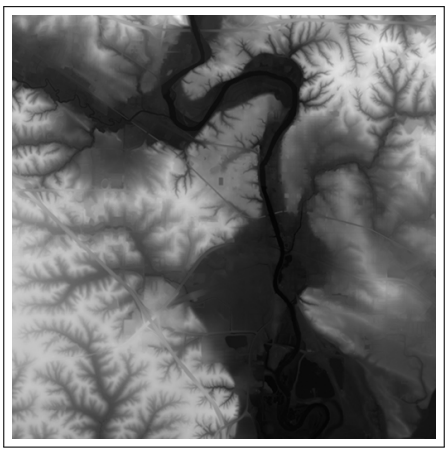


Figure 4.4: Heightmap of Iowa City

Since this was my first experience in handling GIS data, I might have inadvertently introduced inaccuracies - I could only roughly cross check distances and elevation ranges inside the simulation with real map data, they seemed to be in accordance.

The heighmap was then further edited by hand: first only bridges over the riverbed that were seen as obstacles by LiDAR were removed.

Later, after comparing the simulation results to real flood behaviour, additional measures were taken: the riverbed was deepened, since LiDAR reacts to the water surface of a river and thus does not faithfully represent the true riverbed geometry - this step was highly speculative but improved simulation behaviour.

Then a *friction map* was introduced, making a non-uniform value of Manning's friction coefficient possible. It is common to choose different friction coefficients for the riverbed and its banks (see "National Engineering Handbook", p. 11 [EngHbk]). The base value for the friction coefficient and local multiplicators in the friction map were selected according to resources such as *Guide for Selecting Manning's Roughness Coefficients for Natural Channels and Flood Plains* [ManCoef] and based on the quality of resulting simulation behaviour.

4.2.2 Deviation of Simulation from Inundation Maps

Keep the previous section in mind to take the following results with a grain of salt and see them more as qualitative rather than fully quantitative comparisons.

The main benchmark for the accuracy of WebFlood were existing inundation maps of Iowa. These are available in two types: inundation area outlines, which were obtained from the Iowa Flood Information System [FldInfo] and inundation depth maps which were obtained from the National Weather Service [WthrSrv] (both resources are available for the public).

The following two tables 4.1 and 4.2 compare simulation data with these inundation maps. The *Source Height* parameter of WebFlood and corresponding river stage value of the inundation maps is given.

In the top rows of the tables, the water depth of WebFlood (bluer is deeper, white shallower, transparent represents zero water) is overlaid by the inundation outline, showing the difference in flooded areas in general. It can be seen that the simulation follows these outlines quite well, but tends to flood areas slightly too late/too slowly.

In the bottom rows of the tables, the error between the simulation water depth and inundation depth map is shown. There seems to be some discrepancy between inundation depth maps and inundation outlines, since for a river stage of 27.5ft, there is an area that is flooded in WebFlood and the inundation outlines, but not (yet) in the inundation depth maps leading to an unusually large error. The error inside the riverbed is consistently quite large, but this is most likely due to my mis-estimated river bed depth (as mentioned in section 4.2.1). Apart from that, outside the riverbed, the water heights in flooded areas vary only very little compared to the existing inundation depth maps.

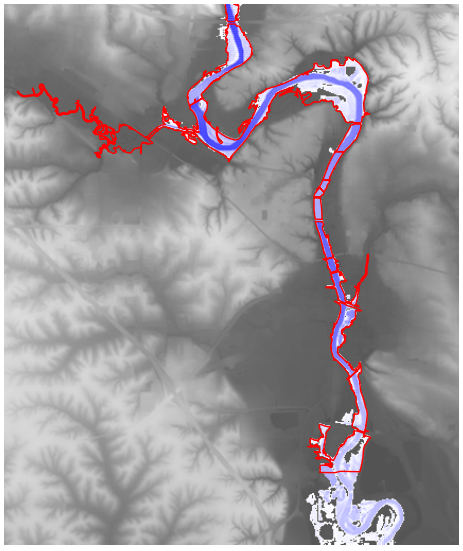
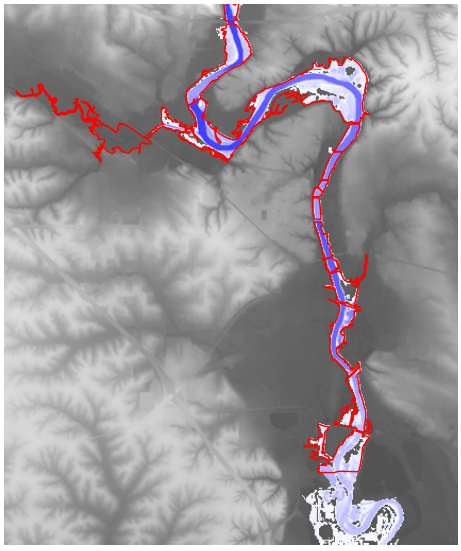
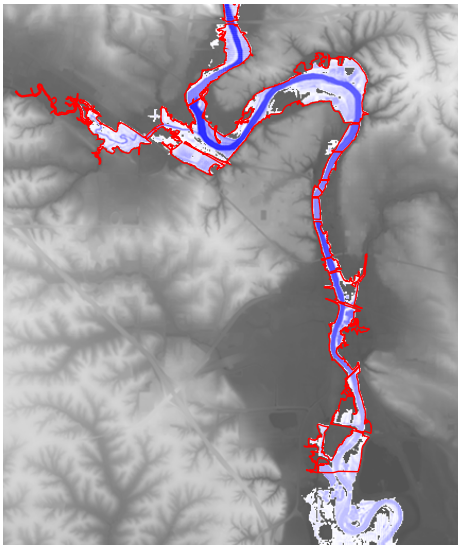
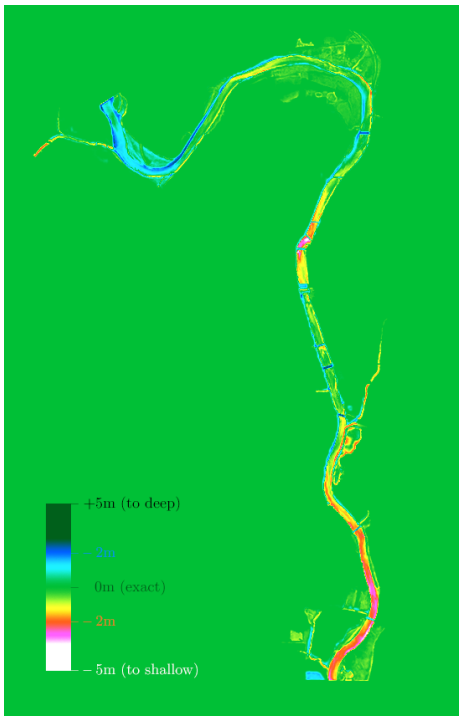
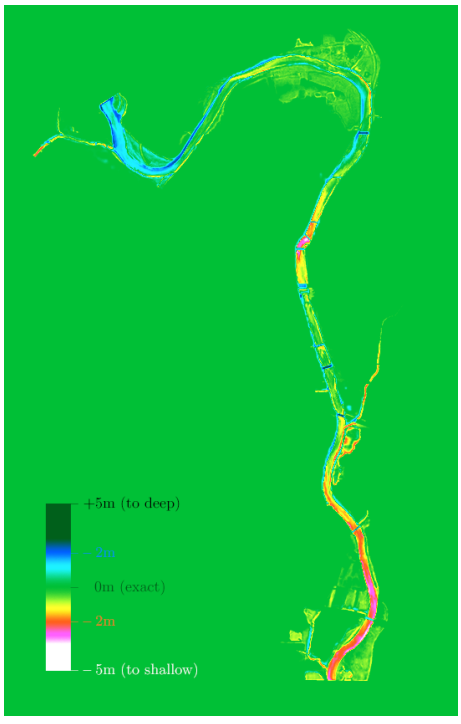
Source Height 7.5 / 25.5ft River Stage	Source Height 7.7 / 26.5ft River Stage	Source Height 7.9 / 27.5ft River Stage
		
		

Table 4.1: Deviation from Inundation Maps for River Stages 25.5-27.5ft

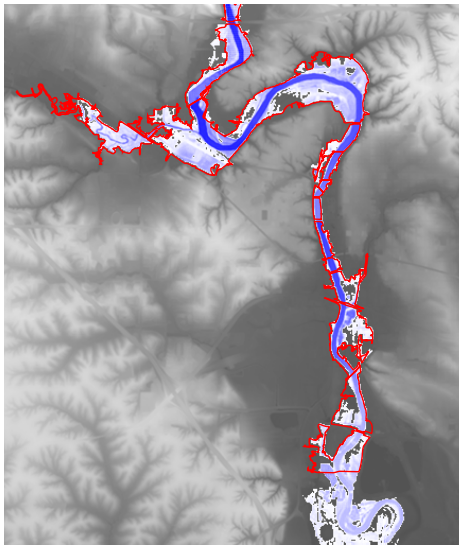
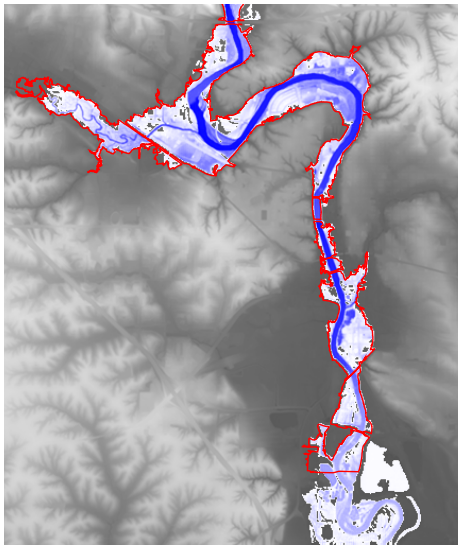
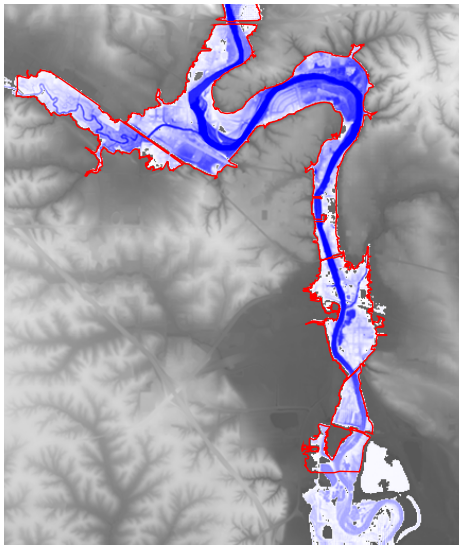
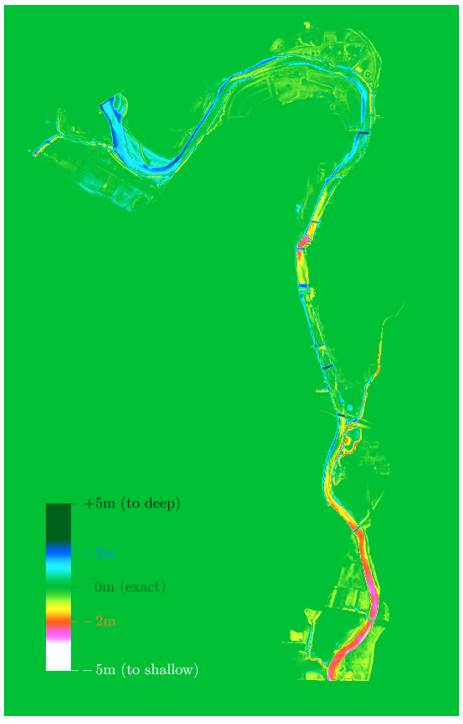
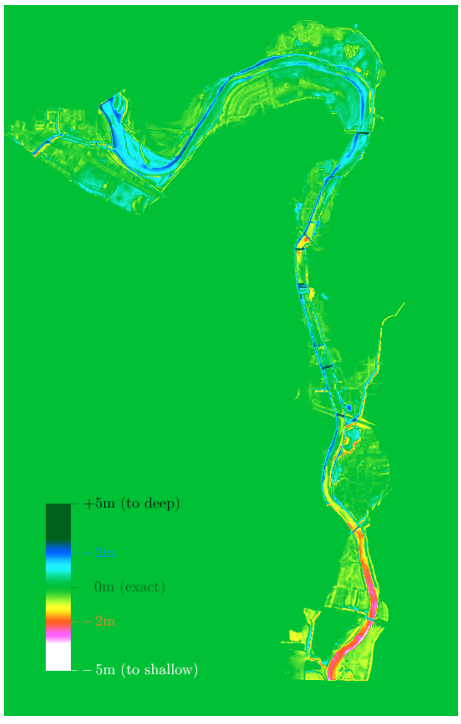
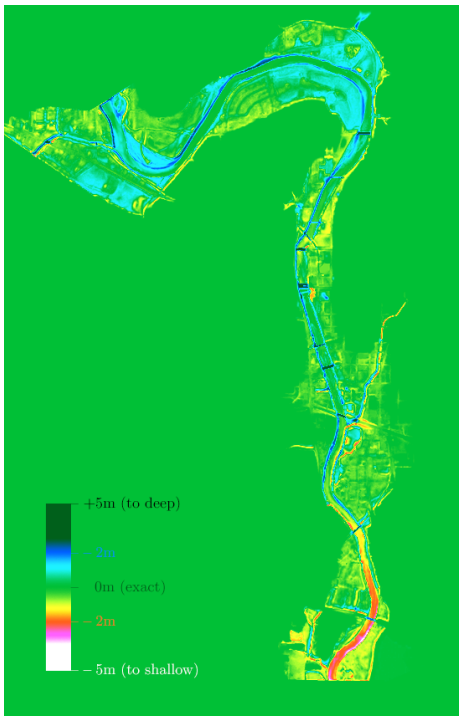
Source Height 8.1 / 28.5ft River Stage	Source Height 8.9 / 31.5ft River Stage	Source Height 9.5 / 34.0ft River Stage
		
		

Table 4.2: Deviation from Inundation Maps for River Stages 28.5-34.0ft

4.3 Performance

Since the simulation kept changing significantly up until the end of this work, I unfortunately did not find time to do rigorous performance tests. WebFlood includes timers that show one-second minimum, maximum and average frame durations, including both simulation and visualization.

To give at least a rough idea: on integrated graphics processors for laptops such as Intel Iris, a frame consisting of one simulation step on a 512×512px map takes 50ms, a frame consisting of 10 steps takes 100ms and a frame consisting of 100 steps takes close to 1s.

Contrast this with current-generation dedicated desktop CPUs where 10 steps on a 512×512px map run at the capped browser framerate of 60FPS (and thus 16.6ms) and 100 steps still only take 35ms.

CPU usage is constantly below 15% on laptops and 4% on desktop machines - performance seems to be mainly GPU-dependent. It also doesn't vary perceptibly between different browsers.

Chapter 5

Conclusion and Future Work

5.1 Suggested Applications

Inspired by the *Iowa Flood Information System* [FldInfo], one obvious usage of an in-browser flood simulation like WebFlood could be to inform the public about past, ongoing and predicted floods. Since the simulation runs interactively even on commodity hardware, the simulation parameters could be updated to realtime values from actual river gauges.

Thanks to its interactive features, it could also be used to quickly test and compare different flood situations. In this way, it could aid decision processes during a flood - even without access to a lot of computing resources.

Finally, WebFlood could be an example of demonstrating and teaching the fundamental theory of a field of interest (in this case hydraulics) in a hands-on, toy-like but still serious manner. Different target audiences can be supported by offering the full range of participation: just observing the simulation, interacting with its parameters and seeing live changes or even looking at and understanding its source code, development process and related work. I would call this idea *Simulation Aided Education*.

5.2 Possible Future Plans

One direction that was only hinted at in WebFlood (by offering overlay visualizations for maximum inundation depth, flow velocity and flood duration at each point of the map) was the modeling and prediction of damage due to flooding. This could be much more explicitly supported by incorporating for example land usage maps and estimating vulnerability of different structures to water depth, water

speed and flood duration. Based on that, actual damage predictions could be calculated.

Regarding the implementation of the simulation itself, there is still a lot of obvious room for optimization, at least for floods of rivers, where most of grid cells are dry and just use up computational resources. A simple tiling of the simulation domain and discarding of inactive tiles would probably already lead to significant improvements. Of course even more complicated methods such as adaptive multiresolution grids could be adapted.

In general, the simulation would greatly benefit from a more rigorous implementation according to current best practises, to ensure full mass conservation, better stability and less artifacts like nonphysical oscillations.

5.3 State and Future of WebGL for GPGPU

The main and almost show-stopping problem of WebGL for GPGPU is the difficulty of communication from GPU back to CPU: only a synchronous version of `gl.readPixels` is offered. A simple CPU read of an (unused) framebuffer can block the main and only javascript thread for the duration of a full frame or longer, preventing further computation on both CPU and GPU during that time. In WebFlood this was avoided where possible, but is required for export of simulation data and display of gauge data.

A smaller annoyance is the fact that floating-point framebuffers can be used on the GPU but not read back to the CPU - to circumvent this, a special shader has to be used that encodes a float into a vector of four unsigned integers. On the CPU this has to be decoded back into a float.

Since WebGL is still only based on OpenGL ES 2.0, it misses many features of more modern graphics APIs that would also be interesting for GPGPU. Examples are geometry shaders, which would allow generation of dynamic amounts of data on the GPU, 3D and array textures and transform feedback, which allows to also store the output of vertex shaders in buffers for reuse. Many of these features are planned to be introduced in WebGL 2, which will be based on OpenGL ES 3.0 [Jon13].

In light of these planned features and after seeing how much can be achieved even in such a restrictive environment as WebGL 1, I am excited about the future of WebGL.

5.4 Final Conclusion

Many complex systems that are difficult to describe with words and even pictures or videos can become intuitive in interactive simulations. This work has additionally convinced me of that and made me even more intrigued about the possibilities in this direction.

Bibliography

[AMC11] Marc de la Asunción, José M. Mantas, and Manuel J. Castro. "Simulation of one-layer shallow water systems on multicore and CUDA architectures". In: *The Journal of Supercomputing* 58 (2011).

[BM07] Robert Bridson and Matthias Müller-Fischer. *Fluid Simulation - SIGGRAPH 2007 Course Notes*. 2007. URL: `http://www.cs.ubc.ca/~rbridson/fluidsimulation/fluids_notes.pdf` (visited on 02/14/2015).

[Bol+03] Jeff Bolz et al. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*. 2003.

[CLT07] Keenan Crane, Ignacio Llamas, and Sarah Tariq. "Real Time Simulation and Rendering of 3D Fluids". In: *GPU Gems 3*. 2007.

[CM11] Nuttapong Chentanez and Matthias Müller. "Real-time Eulerian water simulation using a restricted tall cell grid". In: *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2011* 30 (2011).

[CMK14] Nuttapong Chentanez, Matthias Müller, and Tae-Yong Kim. "Coupling 3D Eulerian, Height Field and Particle Methods for the Simulation of Large Scale Liquid Phenomena". In: *Proceedings of ACM SIGGRAPH / EUROGRAPHICS Symposium on Computer Animation*. 2014.

[Eic15] Anselm Eickhoff. *Citybound*. 2015. URL: `http://cityboundsim.com/` (visited on 02/14/2015).

[EngHbk] U.S. Department of Agriculture. "National Engineering Handbook". In: Part 630 Hydrology. originally prepared 1971. 2012. Chap. Chapter 14: Stage Discharge Relations. URL: `ftp://ftp.wcc.nrcs.usda.gov/wntsc/H&H/NEHhydrology/ch14.pdf` (visited on 02/14/2015).

[FF01] Nick Foster and Ronald Fedkiw. "Practical Animation of Liquids". In: *SIGGRAPH '01 Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001.

[FldInfo] Iowa Flood Center. *Iowa Flood Information System*. URL: http://ifis.iowafloodcenter.org/ifis/main/?snap_view=fmap (visited on 02/14/2015).

[FM96] Nick Foster and Dimitri Metaxas. *Realistic Animation of Liquids*. Tech. rep. University of Pennsylvania, Center for Human Modeling and Simulation, 1996.

[FT93] Luigi Fraccarollo and Eleuterio F. Toro. "Experimental and numerical assessment of the shallow water model for two-dimensional dam-break type problems". In: *Journal of Hydraulic Research* 33 (1993), pp. 843–864.

[GDAL] *Geospatial Data Abstraction Library*. URL: http://www.gdal.org/ (visited on 02/14/2015).

[Geo06] David L. George. "Finite Volume Methods and Adaptive Refinement for Tsunami Propagation and Inundation". Dissertation. University of Washington, 2006.

[Geo10] D. L. George. "Adaptive finite volume methods with well-balanced Riemann solvers for modeling floods in rugged terrain: Application to the Malpasset dam-break flood (France, 1959)". In: *International Journal for Numerical Methods in Fluids* (2010).

[GM77] R. A. Gingold and J. J. Monaghan. "Smoothed particle hydrodynamics - Theory and application to non-spherical stars". In: *Monthly Notices of the Royal Astronomical Society* 181 (1977).

[Göd06] Dominik Göddeke. *Old-School GPGPU Tutorials*. 2006. URL: http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/index.html (visited on 02/14/2015).

[Hag+05] Trond Runar Hagen et al. "Visual simulation of shallow-water waves". In: *Simulation Modelling Practice and Theory* 13.8 (2005), pp. 716–726.

[HW65] Francis C. Harlow and J. Eddie Welch. "Numerical calculation of timedependent viscous incompressible flow of fluid with free surface." In: *The Physics of Fluids* 8.12 (1965).

[Jon13] Brandon Jones. *What's coming in WebGL 2.0*. 2013. (Visited on 02/14/2015).

[JX09] Tan Jie and Yang Xubo. "Physically-based Fluid Animation: A Survey". In: *Science in China Series F: Information Sciences* 52.1 (2009).

[KC05] Andreas Kolb and Nicolas Cuntzn. "Dynamic Particle Coupling for GPU-based Fluid Simulation". In: *Proceedings of 18th Symposium on Simulation Technique*. 2005.

[KE10] Øystein E. Krog and Anne C. Elster. "Fast GPU-based Fluid Simulations Using SPH". In: *Para 2010 – State of the Art in Scientific and Parallel Computing*. 2010.

[Kim+14] Byunghyun Kim et al. "Mesh type tradeoffs in 2D hydrodynamic modeling of flooding with a Godunov-based flow solver". In: *Advances in Water Resources* 68 (2014).

[Lia+08] Qiuhua Liang et al. "International Journal for Numerical Methods in Fluids". In: *Journal of Hydraulic Engineering* 134 (2008).

[LLG05] Gwo-Fong Lin, Jihn-Sung Lai, and Wen-Dar Guo. "High-resolution TVD schemes in finite volume method for hydraulic shock wave modeling". In: *Journal of Hydraulic Research* 43 (2005), pp. 376–389.

[LWK03] Wei Li, Xiaoming Wei, and Arie Kaufman. "Implementing Lattice Boltzmann Computation on Graphics Hardware". In: *The Visual Computer* 19 (2003).

[ManCoef] United States Geological Survey. *Guide for Selecting Manning's Roughness Coefficients for Natural Channels and Flood Plains*. URL: `http://www.fhwa.dot.gov/BRIDGE/wsp2339.pdf` (visited on 02/14/2015).

[MH12] Stephen C. Medeiros and Scott C. Hagen. "Review of wetting and drying algorithms for numerical tidal flow models". In: *International Journal for Numerical Methods in Fluids* 71 (2012).

[Mon05] J. J. Monaghan. "Smoothed particle hydrodynamics". In: *Reports on Progress in Physics* 68 (2005).

[Mül+08] Matthias Müller et al. *Real Time Physics Class Notes*. 2008. URL: `http://matthias-mueller-fischer.ch/realtimephysics/coursenotes.pdf` (visited on 02/14/2015).

[NatMap] U.S. Geological Survey. *The National Map Viewer*. URL: `http://nationalmap.gov/viewer.html` (visited on 02/14/2015).

[Ree83] Willian T. Reeves. "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects". In: *Computer Graphics* 17.3 (1983).

[Sta99] Jos Stam. "Stable Fluids". In: *SIGGRAPH 99 Conference Proceedings, Annual Conference Series*. 1999.

[Thü+07a] Nils Thürey et al. "Real-time Breaking Waves for Shallow Water Simulations". In: *Proceedings of the Pacific Conference on Computer Graphics and Applications 2007*. 2007.

[Thü+07b] Nils Thürey et al. “Real-time simulations of bubbles and foam within a shallow water framework”. In: *SCA ’07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 2007.

[Töl09] Jonas Tölke. “Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA”. In: *Computing and Visualization in Science* 13 (2009).

[USGS10] S. Mike Linhart and David A. Eash. *Floods of May 30 to June 15, 2008, in the Iowa River and Cedar River Basins, Eastern Iowa*. Open-File Report 2010-119. U.S. Geological Survey, 2010.

[WLL04] Enhua Wu, Youquan Liu, and Xuehui Liu. “An Improved Study of Real-Time Fluid Simulation on GPU”. In: *Computer Animation and Virtual Worlds - Special Issue: The Very Best Papers from CASA 2004* 15 (2004).

[WthrSrv] National Weather Service. *Advanced Hydrologic Prediction Service*. URL: `http://water.weather.gov/ahps2/download_gauge.php?wfo=dvn&gage=iowi4` (visited on 02/14/2015).

Appendix A

Appendix

A.1 Simulation Shader Prefix

```
#ifdef GL_ES
    precision highp float;
#endif

// main data texture and uniform inputs

uniform sampler2D texture;
uniform float unit, dt;

// current and neighbor positions, passed from vertex shader

varying vec2 pos, posLeft, posRight, posTop, posBottom;

// macros to access components of simulation data vector

#define V(D)  D.xy         // velocity
#define Vx(D) D.x          // velocity (x-component)
#define Vy(D) D.y          // velocity (y-component)
#define H(D)  D.z          // water height
#define T(D)  D.w          // terrain height
#define L(D)  H(D) + T(D)  // water level

// query (and interpolate) simulation data from texture

vec4 simData (vec2 pos) {
    return texture2D(texture, pos);
}

// forward declare simulationStep
// will be implemented by simulation shader

vec4 simulationStep();

// use return value of simulationStep as output color

void main(void) {
    gl_FragColor = simulationStep();
}
```