ADVECTED RIVER TEXTURES

by

Tim Burrell

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
October 2008

© Copyright by Tim Burrell, 2008

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "ADVECTED RIVER TEXTURES" by Tim Burrell in partial fulfillment of the requirements for the degree of Master of Computer Science.

Dated: October 14, 2008

Supervisors:

Dr. Dirk Arnold

Dr. Stephen Brooks

Reader:

Dr. Norm Scrimger

DALHOUSIE UNIVERSITY

DATE: October 14, 2008

AUTHOR: Tim Burrell

TITLE: ADVECTED RIVER TEXTURES

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: MCSc CONVOCATION: May YEAR: 2009

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

_____

Signature of Author

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests that permission has been obtained for the use of any copyrighted material appearing in the thesis (other than brief excerpts requiring only proper acknowledgement in scholarly writing) and that all such use is clearly acknowledged.

To all those who have come before me,

for without your efforts this research would not have been possible.

To mother nature,

for making fluid simulations so enticing.

And to Jessicah,

for her never-ending support.

Table of Contents

List of Tables

List of Figures

Abstract

Realistically simulating and rendering fluids is an area of computer science that has tantalized computer graphics researchers for years, simply because of the sheer difficulty, and vast range of knowledge that the field encompasses. Simulating rivers is itself a difficult problem within this field because it is one of the more general cases where computational fluid dynamics can be applied. In order to physically model a river the complex interactions between the fluid, air, and the underlying terrain must all be accounted for. And, intending to have this all occur in real-time means that the tactic of applying general, and lengthy, computations to solve the fluid system is not a possibility; some more specific method suited to real-time use is required.

In order to achieve our goal of simulating and rendering a river, in real-time, we took the approach of combining several techniques: a 2D fluid solver that can capture minute details in a river's surface, an efficient method for computing some 3D flow information (which gives the fluid solver just enough information to model the interactions between the water and the terrain, yet remain efficient enough to be computed in real-time), and an animated 3D procedural wave texture that gets advected through the fluid via "advection particles" in order to elicit the highly detailed fluid surfaces that are characteristic of rivers.

Our research, and implementation, shows that the novel technique of coupling an animated texture advection method to a fluid simulation can produce results that are representative of large scale real-world rivers, and that the technique is suitable for use in real-time settings. In one of our test scenes we simulate a ten kilometer long river section at real-time frame rates (60 frames per second) with a grid resolution of 1024x1024 cells. The method is stable (not prone to explosions), and currently has the limitation of only being able to produce planar river surfaces (an efficiency / quality trade-off), however we feel this research presents a successful step toward enabling more large scale dynamic fluid effects in real-time applications.

Acknowledgements

I would like to express thanks and gratitude to both of my supervisors Dr. Dirk Arnold, and Dr. Stephen Brooks, for their continual guidance, insightful commentary, and creative ideas for solutions to some of the problems we encountered along the way. I must also thank Dalhousie University and the Faculty of Computer Science for helping make my Master's research as enjoyable as it has been.

Additionally, throughout my research I have looked to a number of pioneers in the field of computational fluid dynamics for both inspiration and support and have found the community to be both warm and helpful. A special set of acknowledgements go out to some of these people: Joe Stam for his ground breaking work on stable Navier-Stokes simulations, and for his reference Navier-Stokes solver which was used in our fluid solver implementation, Jerry Tessendorf for his work with procedural and statistical wave generation methods, and Hilko Cords and Cem Yuksel for their recent work in the field which helped motivate this thesis.

Chapter 1

Introduction

Fluid simulation is often considered to be one of those areas of computer science that might always be an open problem; it is unlikely that a single fluid solving method will be invented that is suited equally to all simulation scenarios. Currently there are many existing techniques for simulating and studying fluids with computers, but "no single method [exists] that can capture all the subtle effects of water" [33]. We have options for doing offline rendering of complex fluid scenes for applications like movies, and still images (where rendering time is not an issue), and we also have some techniques available to us where we can make visual compromises if efficiency is a factor (i.e., for games, and other interactive simulations).

The scenario we are interested in, real-time river rendering, is a complicated one, because in order to physically simulate a river, the terrain (river banks and river bed) must be taken into account, which typically means that a full 3D simulation is required. It can also be seen that it is not just the fluid that must be simulated but its interaction with air and gravity as well. There is also the additional issue that rivers are generally very large so even many of the existing real-time techniques are either too expensive or do not exhibit the properties we need in order to present a realistic visualization of a river to viewers.

In the end, through experimentation, we have come to believe that the minimum requirements needed to physically simulate and realistically render a river are: some form of 3D free-surface solver that can either solve a highly detailed fluid surface in real-time, or some method that can be coupled with a technique that can adapt a lower resolution fluid solver to a higher detail fluid surface construction (carpet construction) method.

Current real-time techniques generally do not easily fit into this set of requirements. As soon as a 3D fluid solver is required the computational cost of the algorithm almost always increases to the point where the technique is no longer suitable for use over large scale fluid volumes (in real-time), so the goal of this thesis is to find some compromise between physical correctness and efficiency that would produce the most realistic results possible.

1.1 Problem Statement

Rendering large scale river fluid flows, for real-time applications, is difficult because of the complexity involved in generating a fluid surface that is both detailed enough to look visually realistic, and efficient enough to maintain interactive frame rates. Even slow moving rivers with a largely flat and calm surface still have highly detailed fluid surfaces where, for example, one can pick out sections of the surface that are moving in opposing directions and with differing velocities (see Figure 4.1 for examples).

There are a number of reasons for the complex nature of river flows, such as the interaction between the fluid and the river bed and banks, along with any obstacles that may be in the way. Aside from these obvious factors there are also a series of less obvious things that impact a river's surface definition such as particulate in the water that affects its viscosity and density. These changes in density and viscosity result in subtle temperature differences that alter the movement and flow of the water, and in turn help define what happens on the surface of the river.

Typically the Computational Fluid Dynamics (CFD) field has used two distinct classes of fluid solvers to deal with different scenarios (shallow water and open water), since the problem domains are different enough that using a generic solver for one task or the other would result in redundancy and unnecessary overhead. Simulating rivers is a problematic area because they often include situations where not only both shallow water and deep water are present, but where the two situations can overlap.

Given all these issues the goal of the method described here was not to mathematically model all of the minutia of details that can comprise a river fluid volume, but rather to attempt to find a technique that would approximate as much of this detail as possible while still remaining efficient enough for interactive purposes. We also chose to add the additional requirement that the method should be suitable for coupling with a rigid-body physics engine to enhance interactivity (i.e., it should be possible to have objects in the scene interact with the river's surface).

1.2 Introduction to the Solution

We present a method for generating intricately detailed fluid flow surfaces for the purpose of realistically visualizing rivers as they flow over arbitrary terrains in real-time.

By combining a number of computational fluid dynamics and visualization techniques we have achieved the desired goal: detailed fluid surface construction that responds appropriately to underlying terrain information and simulates the detail present in real fluid volumes without the requirement to implicitly solve for it. The algorithm runs in real-time on current hardware and although we did not include any rigid-body physics the algorithm was designed with this ability in mind.

In order to achieve these goals we chose to combine a 2D Navier-Stokes solver for its stability, efficiency, and accuracy, with 3D information gleaned from a series of Hydrostatic Pressure (HSP) columns. We do not use HSP columns by themselves because although it is an efficient fluid simulation method, it is not suitable for large scale river representations as it cannot capture detail effects such as vortices and eddies [19, p. 16]. We then couple the results of our pseudo-3D Navier-Stokes-HSP fluid solver with a procedural wave generation and Texture Advection method in order to derive the highly detailed river like fluid surfaces we require.

Chapter 2

Prior Work / Literature Review

Researchers in the Computational Fluid Dynamics (CFD) field currently employ many techniques for different types of fluid simulation scenarios. There are algorithms for deep water, shallow water, enclosed volumes of water, and volumes that contain a mixture of fluids and gasses, called free-surface solvers, that can simulate the common scenario of mixing water and air. Even things such as smoke, sand, and clouds can all be simulated with fluid dynamics [14, 30, 47, 54, 56]. Some simulators are stable (meaning they do not explode even when faced with large simulation time deltas or with differing scenario parameters), while others are not (which may not be a problem depending on the circumstance). Some run in real-time, and others are either too computationally expensive or do not adapt well to real-time situations.

Amongst all these techniques there are a few different underlying methods for solving the fluid systems that range from numerically solving the differential equations required to mathematically model fluids, to methods that aim to simulate water at the molecular (or particle) level, and yet others that are somewhere in-between.

Our aim was to build a system that could realistically simulate and render rivers which, in terms of fluid simulations, can be viewed as large bodies of water that have the ability to be both shallow and deep, and which require interaction between terrain and fluid. Rivers also typically have the free-surface (air to fluid) requirement built-in so they can be represented in a realistic environment. Our goal was to stably simulate the river in real-time, and to have the ability to employ a method for two-way coupling between objects in the scene and the fluid, which meant the technique could not be overly computationally expensive. With these parameters in mind we will present some of the prior work that has been done in CFD and discuss how it relates to our topic as well as give a brief synopsis of some past and recent work that specifically targeted river rendering.

2.1 Navier-Stokes Based Solvers

The most common class of fluid solver, for real-time use, is the Navier-Stokes based solver. The Navier-Stokes (NS) equations (which we also use, and describe in detail in Section 3.3) are a set of equations that are derived from applying Newton's Second Law of Motion to incompressible fluids. The Navier-Stokes equations take the form of differential equations and are applicable over a wide range of problems. Solvers for the equations exist that can simulate shallow to deep water, 2D and 3D fluid volumes (both free-surface and otherwise), and are available in both stable and unstable variants.

Real-time CFD can trace its roots back to approximately fifteen years ago. Computers were becoming powerful enough that fluid simulations algorithms were feasible in real-time, and as a result more researchers were beginning to show interest in interactive and real-time fluid simulations. But, at that time, finding a stable algorithm that worked well with different simulation time step sizes (a beneficial property to have for real-time simulations where the simulation may not be able to be held to strict time delta constraints) and for varying simulation purposes was difficult. In 1994 Chen and da Vitoria Lobo proposed a method for doing real-time Navier-Stokes [5] on limited size volumes of water by directly manipulating the NS pressure field. Their system allowed for interactivity and provided the appearance of using a three dimensional fluid solver, however they used the less expensive 2D Navier-Stokes and combined it the resultant pressure values from their NS advection step to represent the 3^{rd} dimension. They also solved the NS equations with an unstable advection solver, which meant the system suffered from the standard explosion issues that many unstable solvers share. However, despite its limitations, their work was impressive at the time and had an impact on many papers to come, including this thesis. Our use of a 2D NS based solver, combined with another method to provide 3D information, comes as a direct result of the influence of Chen and da Vitoria Lobo's contributions.

Joe Stam's work with Navier-Stokes based solvers has been groundbreaking as it was his method that allowed, "for the first time, ... an unconditionally stable model which still produces complex fluid-like [flows]" [47, p.1]. In 1999 Stam published "Stable Fluids" [45] where he outlined a technique for performing stable advection, thus enabling the technique for applications, like games, where stability is of primary concern. In 2001 he co-authored a paper where Navier-Stokes was used for visualizing smoke [14], then two years later, in 2003, he published "Real-Time Fluid Dynamics for Games" [47] (the algorithm used in

this thesis is, in-part, based upon Stam's 2003 NS solver), which was a refinement on the technique he published in 1999, that emphasized stability and efficiency.

In 2007 Lee and Sullivan, presented a method for efficiently computing shallow water equations "suitable for ponds, lakes, or oceans" [28, p. 1]. Their paper represented a step forward in free-surface Navier-Stokes based fluid solvers in the realm of efficiency (it can also be easily parallelized), but it is at the expense of stability, and the method falls into the unstable category of solvers. To this day full blown free-surface 3D fluid solvers remain computationally complex enough that they are unfeasible for large volumes of fluid.

2.2 Particle-System Based Solvers

Another area that shows exciting and promising work is that of particle systems. Rather than trying to implicitly solve the differential equations involved in fluid systems these methods take the approach of modelling the interaction between water molecules by representing them as particles. One active area of research in the particle based class of fluid solver is the Smoothed Particle Hydrodynamics (SPH) method which was originally developed by Lucy [32] and Gingold and Monaghan [16] in 1977 to study astrophysics phenomena. Despite its origin in astrophysics, the technique is generalizable enough to apply to fluids, and has since been used to create some very impressive fluid simulations.

The basic idea behind SPH is to break the fluid up into a number of elements (or particles) intended to generalize that portion of the fluid. These elements have a predefined spatial distance from one another called a "smoothing length", to which their properties are smoothed via a kernel function. The benefit of SPH based methods is they get conservation of mass automatically, as well as free-surface (the computations are essentially a simulation of objects interacting with one another in air). SPH solvers are also very good at modelling splashes, waves, and other fluid phenomena that require parts of the fluid to fold over or break away from itself; the method is not heightfield based, and is typically coupled with a mesh construction algorithm such as Marching Cubes (which produces a number of fully enclosed convex-hull meshes as output). Solvers in the SPH class are also amenable to being integrated into rigid-body physics systems as their calculations are already closely related to those employed by 3D rigid-body physics engines.

One issue with SPH methods is the limited number of particles that can currently be simulated. In 2003, Müller, Charypar, and Gross were able to achieve interactive rates with

5000 particles (enough to represent a small glass of water) [37]. In Clavet, Beaudoin, and Poulin's 2006 paper, "Particle-based Viscoelastic Fluid Simulation," they achieved 1000 particles at 10 frames per second [8], which is even fewer than Müller in 2003, but their system was far more complex, offering two-way arbitrary object to fluid coupling, including buoyancy and other realistic effects.

In 2006 Kipfer and Westermann from Havok (a commercial physics engine company) published a paper "Realistic and Interactive Simulation of Rivers" in which they simulate rivers using SPH. They achieved interactive rates with 3000 particles [24, p. 1] (a great achievement at the time), and produced a fluid volume large enough to approximate a river. Their method allowed the highly dynamic fluid paths that are consistent with SPH based solvers, but their fluid surface is not very realistic in terms of surface deformations or fluid behaviour. The number of particles they used was simply not high enough to capture the minute details present in a real river surface, or inside the fluid volume itself. The fluid can also be seen to be behaving more like a small stream would (i.e., a garden hose, or water tap) rather than a river, which is again because the particle set size was limited and thus did not have the appropriate mass behind it that a large fluid volume like a river would.

SPH has successfully been used, in an offline manner, to create stunning fluid simulations such as Fedkiw, Losasso, Talton, and Kwatra's "Two-Way Coupled SPH and Particle Level Set Fluid Simulation" [31], so the technique has been proven to scale to large quantities of fluid. All that remains for it to be usable in the real-time domain, with large scale fluids, is for computers and graphics cards to keep increasing their computational power. In recent years GPU based solutions have appeared in commercial physics engines that can currently handle roughly 100,000 particles (nVidia's own reference implementation, as presented at GDC 2008 on an 8800 GTX, simulated 65,000 particles in a small box [18, p. 13]), but even this is still not enough to simulate a river with fine grained detail. The body of water that we are aiming to simulate would require perhaps tens to hundreds of millions of particles (or more) to be simulated in real-time via SPH which is simply not feasible at this time.

Another interesting particle based method that has shown up in recent years is the "Wave Particles" method from Cem Yuksel, et al.. [55]. It bears no resemblance to SPH, and is a purely 2D method (the surface is a 3D heightfield but the simulation extends only in

two dimensions). Rather than attempting to simulate an entire body of water at the molecular level, the Wave Particles encapsulate deviation functions that perturb a fluid surface heightfield based on interactions between rigid body objects and the fluid. Their method shows much promise, especially in regard to river rendering, and was one of the primary inspirations behind this thesis.

We intended to use a Wave Particle based approach because at first glance it appears to satisfy all our constraints, however after a basic test implementation we discovered that it would not yield the surface detail we wanted, and was still too expensive. The method does not, itself, provide a two way coupling at both the fluid to fluid and fluid to object levels (only fluid to object, and vice versa) which meant it would still need to be tied to an underlying fluid simulation. Furthermore, Yuksel et al. managed to interactively simulate 600,000 particles at a 256x512 grid resolution which told us that even without a secondary fluid simulation, it would still not possible to get the desired detail level and simulation realism for the volume of fluid we wanted to simulate, using their method.

In 2008 Hilko Cords published "Moving with the Flow: Wave Particles in Flowing Liquids" [9], where he tied a Wave Particles based simulation to a 3D Navier-Stokes simulation in order to achieve something akin to what our aim was. Cords' solution was an enclosed, small body, shallow water system, so it was not directly applicable to our problem. However, it did prove that the idea of coupling Wave Particles to a fluid simulation is both sound and produces excellent results. It also showcased another shortcoming of the method however, which is that effects like non-planar fluid volumes (splashes, waterfalls, etc.), must still be simulated and constructed via other means, for which Cords used a particle system method similar to SPH combined with Marching Cubes.

2.3 Hydrostatics Based Solvers

There is yet another approach to solving fluids called Hydrostatic Pressure (HSP) columns, first presented by Kass and Miller in 1990 [23], that indirectly approximates the incompressible fluids equations in a different way. Rather than mathematically solving the Navier-Stokes equations, or representing the fluid with a particle system, they created a system of columns and pipes that move water from one column to another based on the laws of hydrostatics (the laws of fluid at rest). It may seem counter-intuitive to use the laws of fluid at rest to simulate moving fluids, but the notion is that fluid generally wants to come to

rest while moving, so we can also use these principles to implement a system of motion constraints and apply them to pipes, and columns, which ends up approximating how fluids behave.

There are a number of benefits to using the HSP approach, the first and foremost being efficiency. The method is extremely fast for large fluid volumes; rather than simulating the entire volume in 3D (recall that for SPH, as the volume grows in size, the number of particles increases in a cubic fashion) the columns increase in height with the depth of the fluid, so adding more fluid to a system has a much less significant impact on the cost of the algorithm than with SPH. The equations being solved are also very quick to evaluate, and can easily account for rigid body to fluid interactions because there is an external pressure component built into the algorithm.

One of the problems that plagues SPH simulations is that it is an unstable method, and simulation explosions will occur if time deltas become too large or vary too much from frame to frame [33, p. 108]. We chose to only rebuild the HSP information when needed (not every frame) so instability is not a problem, and we are only using it to gain 3D pressure and volume transfer (velocity) information (i.e., to augment our 2D NS solver with some 3D information). In a river that does not change fluid paths this pressure information does not vary much over time, so we were primarily looking for an algorithm we could use over a huge volume of fluid and not have it take hours (or days) of processing time.

Several river based simulators also incorporate HSP solvers as it is one of the few techniques currently available that is efficient enough to produce a large scale fluid simulation in real-time. Holmberg and Wünsche's "Efficient Modeling and Rendering of Turbulent Water of Natural Terrain" [19] from 2004 used an HSP based system to represent a very small river section. Their aim was more to generate splashing based on interactions between fluid and terrain than large scale rivers or a detailed surface representation. Two years later, in 2006, Maes, Fujimoto, and Chiba presented "Efficient Animation of Water Flow on Irregular Terrains" [33] which used HSPs and a particle system to model both a river fluid surface and splashing effects.

Aside from the previously noted unstable nature of HSPs, they are also unable to simulate the complex fluid phenomena that can be captured with an NS based simulator such as vortices, eddies, etc. This was directly experienced during our experimentation with the method and noted by others as well:

> The model is not capable of simulating certain situations such as vortices. A second problem arises from the fact that turbulence is a feature of flow and not of the fluid "at rest". This means that while hydrostatics may be easy to use the equations generated for flow are incomplete and ignore many of the visible characteristics of water such as viscous shear stresses.[19, p. 2]

Based on the results of these papers we decided we did not want to use HSP columns as the basis for the algorithm, but instead use it to augment the NS solver, and use another method for generating a highly detailed surface mesh.

2.4 Procedural Fluid Generators

Much work has also been done in the area of statistical or procedural fluid generation. The idea behind procedural systems is not to directly simulate a fluid volume, but rather to produce a realistic looking illusion of water being simulated. The results in this area have been very good and go back much further than the 15 years of history that real-time fluid dynamics has, so the body of work is much larger. This is mostly because the methods are not nearly as computationally expensive and thus have had application for a longer period of time.

However, although procedural methods are efficient and yield the detailed results we are interested in, they are not directly suited to the purpose of simulating realistic fluid volumes for rivers. It may be possible to capture fine, and realistic surface details for oceans and other deep water simulations by procedural means, but complex fluid interactions that require a fluid simulator (such as fluid interacting with obstacles, or terrain) can not be done easily via procedural wave generators by themselves.

Perhaps the most important work done on statistical wave generation dates back to 1989, with Perlin's work on noise functions [41], which influenced a whole series of statistical and FFT based approaches to wave generation, including Jerry Tessendorf's FFT based deep water ocean simulators [49, 50] which come as a direct result of work done by Mastin et al. [34]. In 2004 Mitchell, an ATI researcher, published "Real-Time Synthesis and Rendering of Ocean Water" [35] in which he used Tessendorf's technique to great effect, as well as integrating a shallow water solver and some object to fluid coupling. The procedural method used in this thesis is directly based on Mastin's, Tessendorf's, and

Mitchell's work.

Another interesting procedural method we investigated was Gerstner waves. Brown presented a short paper on his work with Gerstner waves for the Ice Age movie [4], which resulted in a realistic rendering of very choppy, and rapid river water. Gerstner waves are good at producing these types of fast moving and turbulent fluids because the Gerstner function typically results in crested and pointed waves. However, one must explicitly define the parameters of the Gerstner functions, and setting these parameters for a series of Gerstner waves, in ways that produce nice and realistic looking waves, is very much an artistic endeavour. That being said, there may still be room for future work in this area by augmenting or replacing our Tessendorf based procedural texture generator with Gerstner waves if a particularly turbulent river is being visualized.

In 2007 Shi, Ye, Dong, and Zhang published a paper entitled "Real-time simulation of large-scale dynamic river water" [44], where they used a statistical (FFT) based procedural approach to generating a fluid surface (i.e., they do not physically simulate the fluid). Although it is difficult to discern their exact methodology from their paper it appears as if they use a combination of Gerstner waves and other statistical approaches to produce the visualization of the fluid's surface. They were more interested in dynamic flood routing (fluid path finding) than the fluid representation itself, so their results are not directly applicable for our application.

2.5 Texture Advection / Synthesis

Texture synthesis is the procedure of creating a large image from a small sample, or samples of images by looking at the sample images' content. Texture advection is the procedure of advecting (or transporting) colour from one location in a texture to some other location based on user defined parameters. Texture synthesis and advection are often seen used in conjunction because the two techniques can be coupled to create useful and interesting results. For our purposes we were primarily interested in texture advection; moving portions of a texture from one location to another, or moving procedurally generated waves along a river as per the results dictated from a fluid simulation.

The texture advection / synthesis field is a relatively new field of computer graphics; the body of research in the area that has been published with regards to fluid simulations is still quite small, however Kwatra, et al., published "Texturing Fluids" in 2007 [26] in which

they used texture synthesis and advection tied to a fluid simulator to produce very realistic looking results. They chose to complement a full 3D NS solver with texture advection to produce textured fluids where the fluid's texture flows naturally with the fluid. They were not concerned with performance in terms of interactivity or real-time results, so their technique is not immediately applicable to our problem domain, but their work was more proof that combining texture advection with a fluid simulator can be done and produces good results.

Ultimately we decided not to include texture synthesis and instead use the previously mentioned Tessendorf based procedural wave generation method as the "texture" to be transported via texture advection. This was both a functional decision, as well as one that was made for performance reasons. Kwatra et al. required up to 200 seconds per frame in order to render a small volume of fluid [26, p. 3], and while it is unclear how much time was spent in each part of the algorithm (Navier-Stokes, synthesis, advection, rendering, etc.), it was clear that the technique is not yet efficient enough (or computers and graphics cards powerful enough) for it to be used in real-time applications. Despite this it would be an interesting experiment to swap out the procedural wave generator in favour of a texture synthesis mechanism and observe the results in order to gauge whether or not the end results are more visually realistic. If so, perhaps there is room for performance enhancements and optimizations to the technique which could allow it to be used in real-time settings.

Chapter 3

The Fluid Simulation

There are a number of issues that make river rendering such a problematic real-time task. The volume of water present in a river is typically quite large, and it is usually not enough to use a deep water statistical approach to simulate them since rivers have directionally flowing fluids that must interact with the terrain in order to produce realistic looking results. This means that some form of a 3D free-surface fluid simulation is needed (the interaction between fluid and terrain as well as between fluid and air must be modelled). There are techniques that are capable of this (such as 3D Navier-Stokes) but they are typically too computationally expensive to be used in real-time (for the large scale fluids). For example, at the time of writing, using the most advanced graphics card currently available and a fully optimized GPU implementation of a 3D Navier-Stokes (NS) solver, researchers have only been able to render very small enclosed fluid volumes at interactive rates [18]. For the purpose of river rendering this is clearly not an option, so we needed to devise an approach that would give us the detail we wanted in order to produce a realistic looking fluid volume, as well as have the efficiency we need in order to be able to run the simulation in real-time.

After experimentation with many pre-existing fluid simulation techniques we decided our hybrid approach will use a combination of 2D Navier-Stokes, Hydrostatic Pressure Columns, and Texture Advection in order to achieve the final result. The 2D Navier-Stokes is the heart of the simulation, and provides the fluid flow velocity and density information for the simulation. The NS portion of the algorithm gives the fluid surface the ability to appropriately interact with river banks and obstacles, but some method of providing the NS solver with the ability to simulate inter-fluidic shear stresses and 3D interactions with the terrain was still a requirement (i.e., the ability to approximate a 3D solver).

For this purpose we decided to use Hydrostatic Pressure (HSP) columns. The HSP columns output a pseudo-3D fluid flow (with a depth resolution equal to the HSP column depth) in addition to pressure information, and also has the advantage of not being very expensive to compute. Our method couples the HSPs with the NS solver in order to affect

the NS velocity and density fields based on the 3D pressure information from the HSP grid. Since the HSP columns are not the primary simulation it can be run under precisely controlled conditions such that explosions do not occur. In our case we only re-run the HSP simulation whenever the river changes paths (once, at startup), and instead vary the 3D pressure information over time via random fluctuations (the details of this is discussed in Section 3.4). In a large fluid body, such as a river, our experiments showed that the HSP pressure field did not vary significantly unless some major change to the fluid path was introduced, so rather than continuously compute the HSP data we only do a full update whenever necessary and approximate the changes the rest of the time. This results in an efficient, and stable fluid solver that, although not completely physically based, results in a realistic approximation of a river fluid flow.

After obtaining some 3D flow information we need a method to yield the highly detailed surface deformations commonly seen in rivers, for which we decided to use Texture Advection of an animated, procedural, ocean wave generation algorithm. We use the velocity and density information from the NS step (which has been affected by the pressure information from the HSP grid) to advect the ocean wave texture using what we call "River Particles", a method for transporting procedurally generated waves with the fluid simulation results (discussed in detail in Section 4.1). This produces very a finely detailed river surface where fluid features such as vortices, and intermingling or directionally opposing fluid flows can be seen. The parameters of the algorithm can be tweaked to produce many different types of rivers (from wide and slow moving, to thin and fast moving, deep to shallow, etc.).

3.1 River Path Finding

Rather than allow a totally dynamic river path that can be changed significantly during a simulation we chose to pre-compute the area which the fluid can occupy. We argue that this is a reasonable compromise to make because, for the most part, rivers are static and do not change their paths frequently. As such, the method presented here is suitable only for large scale rivers that are not going to change either in terms of their flow path, or fluid depth. While this may sound limiting, it can be seen that many representations of rivers, in real-time visual simulations (such as games), already adhere to similar sets of limitations, and since fluid simulations are exceedingly computationally complex the

trade off in loss of flexibility versus gain in performance and simulation accuracy is both worthwhile and desirable. In order to accomplish this we used a river path finding algorithm that determines, based on information from the terrain (and a simple directional flow hint), where a fluid will flow.

Algorithm 1 CalculateFluidFlowPath(TerrainGridCell c)

```
1:  if c.visited then {Ensure no cell is visited twice}
2:      return false
3:  end if
4:  c.visited = true
5:  if c.height[1] > WaterLevel then {If the cell is dry mark it and move on}
6:      return false
7:  end if
8:  if c.isOnBorder(InitialBorder[2]) then
9:      return true
10: else
11:     if CalculateFluidFlowPath(TerrainGrid.rightOf(c)) or
            CalculateFluidFlowPath(TerrainGrid.leftOf(c)) or
            CalculateFluidFlowPath(TerrainGrid.topOf(c)) or
            CalculateFluidFlowPath(TerrainGrid.bottomOf(c)) then
12:         return true
13:     end if
14: end if
15: return false
```

It should also be noted that pre-computing the river's flow path does not preclude the ability of having an interactive fluid simulation (quite the contrary actually, as this technique has been designed with interactivity specifically in mind). As we will see in a later chapter the method can easily be expanded upon by including an interactive rigid-body to fluid simulation coupling, as long as the rigid-body interactions are not intended to alter

[1] Here height refers to the height of the terrain, so that if a position on the terrain has a height that is greater than the desired water level, there is no chance it will ever be a wet cell.

[2] InitialBorder is a user supplied hint that lets the algorithm know which grid border the fluid will flow from. No information is required as to where the fluid shall flow to; this is only used as a starting point for the algorithm.

the river's path.

In order to facilitate easier Level of Detail (LOD) transitioning for the terrain, the terrain geometry is first split up into conveniently sized patches. The patch size parameter is scenario specific, and its purpose is merely to help create a very simplistic form of geometry LOD, however for our application we chose the size of one patch to be equivalent to the area that would regularly be on screen, and present in high detail, were the camera placed at approximately ground level. We did not wish to employ an advanced geometry LOD algorithm, and instead chose to represent each terrain patch with a different detail level depending on the distance from the center of the patch to the viewer; for more information regarding this see Section 5.2.

After the terrain is split up we make the assumption that for any given point on the terrain, that point will only be able to contain fluid if it neighbours a terrain patch border, or there is an unbroken path of wet cells between it and another edge of the terrain patch, and that the height of the terrain at that point is less than or equal to the desired water level height. The algorithm stores the results in a table and thus, if it has already checked a cell on the terrain it does not recurse on it again. For a detailed look at how to calculate the fluid flow path, see Algorithm 1.

After *CalculateFluidFlowPath* completes, the fluid simulation will be confined to the area above the terrain where the algorithm has determined a river should flow. Ie, the algorithm determines which cells on the grid will be designated as "wet", and these cells will be the only ones capable of containing fluid. The non-wet cells will not be considered at all by the fluid solvers which ensures that all the expensive fluid simulation computations are calculated only for the precise volume that is required at each simulation step.

However, we still need to compute the grid cell boundary normals at each boundary location. Wherever there is a divide between wet and dry cells a boundary forms, and we need to calculate the normal for the boundary. This will become important later on during both the Navier-Stokes solver updates, and during the Texture Advection phase, when the Advection Particles are being transported through the river. The Navier-Stokes solver will need the boundary normals in order to perform boundary calculations during advection, and similarly for the texture advection step. In order to facilitate this, we calculate boundary normals in the usual fashion (the normal points directionally orthogonal to the boundary edge).

3.2 Bootstrapping the Hydrostatic Pressure Columns via Navier-Stokes

Once we have obtained the fluid volume from the River Path Finding algorithm, we are ready to start simulating the fluid, and the first step in that process is to generate an initial Navier-Stokes flow. The initial flow will not be affected by the Hydrostatic Pressure Column (HSP) grid yet, and is only used in the initial phases of the algorithm to give the HSP grid information as to which direction the fluid will be flowing, so that we can generate an accurate 3D pressure field.

We have not yet discussed Hydrostatic Pressure Columns in detail, and it is not our intent to digress into it here, but since our algorithm requires that the two steps (NS, and HSP) be dependent upon one another we would like briefly discuss the underlying idea behind the combination of these two distinct fluid solving techniques. Ultimately the goal is to give the 2D NS solver the 3D flow information that the HSP columns provide, so what we do is simply provide the pressure grid that the HSP solver outputs, as input to the NS solver.

More details regarding the HSP column technique will follow (see Section 3.4), however the HSP bootstrapping process is trivial. In Equation 3.22 it can be seen that the HSP computations are dependant on the velocity in the previous time step. Therefore in order to generate an HSP pressure map one must simply substitute the initial phase of NS velocity output into the velocity component, $\mathbf{u_0}$, of the HSP velocity update equation.

However, it is difficult to produce the types of large scale constant flows that rivers exhibit with HSPs. Because of this we decided to bootstrap the the HSP solver with external constant flow information. Since we already adapted our NS solver to achieve a constant flow (see Section 3.3.1), we decided to give the HSP simulation its flow information from a single run of the Navier-Stokes solver. This has the unfortunate consequence of causing a co-dependency between the Navier-Stokes solver and the Hydrostatic Pressure columns, and in turn makes the algorithm sound more complex than it is. Essentially we are doing nothing more than giving the HSP simulator constant flow data, but we are providing it with data from the Navier-Stokes solver. This was done for no reason other than convenience, and there are likely other methods or means that could be used to provide this information to the HSP simulation, but it made sense for us to re-use the NS solver component for this purpose.

It should also be noted that the although we bootstrap the HSP simulation with 2D flow

results from the NS step, this does not affect the HSP simulator's ability to produce 3D pressure information. The bootstrapping procedure is only done in order to give the HSP simulation a means to determine the flow rate and direction of each hydrostatic pressure column cell. Once these values are input into the HSP solver, the results are 3D flow information that are completely based on the 3D terrain geometry underneath (and potentially inside) the fluid volume.

After the initial Navier-Stokes system is solved, and the results have been provided as input to the HSP solver, the resultant pressure grid can then be used, in turn, to influence the 2D Navier-Stokes solver with the 3D pressure data.

3.3 Fluid Flows via the Navier-Stokes Equations

This section is not meant to provide the reader with a comprehensive understanding of the math involved with solving the Navier-Stokes equations. We provide a basic explanation of the details only in so far as they help to yield insight into how and why the Navier-Stokes equations fit into our river simulation technique. We will focus primarily on the relationship between the results of the Navier-Stokes solver, how they apply to the Hydrostatic Pressure Columns, how they are used during the Texture Advection phase, and how we achieve a constant flow by applying impulses to to the Navier-Stokes velocity field. The mathematical basis behind the Navier-Stokes equations, as presented in this section, is merely meant to provide the basic underlying knowledge required to understand the relationship between the input and output results from the NS solver.

At the highest level the NS solver we chose (which is based directory off of Stam's reference implementation [47]) can be thought of as a black box that takes, as both input and output, a grid of cells where each cell has a velocity and density value associated with it. The initial state of the grid is a zero state where all grid cells have zero for both velocity and density, and something external to the NS solver starts the system by applying either a velocity, or a density, to one (or more) of the cells. Through solving the Navier-Stokes equations the NS solver modifies the velocities and densities of all the cells in the system and what was the input grid will then contain the output of the solved fluid system. Keeping the high-level view in mind, we will briefly discuss the basic principles involved in Stam's algorithm [47] for solving fluid systems with Navier-Stokes and how the solver method applies to our overall river simulation strategy.

As with all fluid simulations, some mathematical assumptions must be made in order to facilitate the task of solving a fluid system, and the Navier-Stokes equations are also based on a set of these assumptions: that the fluid is both incompressible and homogeneous. Often the NS method is referred to as the Navier-Stokes equations of incompressible flows, which means that the volume of any sub-region of the fluid must be constant over time, and that its density (ρ) is constant in space [2, 15, 45]. This combination of assumptions leads to the simplification that the fluid density is always constant in both time and across space, which is perfectly and physically valid for many known real fluid interactions such as the one between water and air [15, p. 641].

The basic idea behind a Navier-Stokes simulation is to represent the fluid using a regular grid that adheres to a spatial co-ordinate system such that $\mathbf{x} = (x, y)$, using time variable, t, [15, p. 641]. The fluid itself is represented via a velocity field, $\mathbf{u} = (\mathbf{x}, t)$, and a scalar pressure field, $p(\mathbf{x}, t)$. Assuming that the velocities and pressures are known for the initial time ($t = 0$), then the Navier-Stokes equations for incompressible flows[1] can be used to describe the fluid as an expression over time:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + v\nabla^2\mathbf{u} + \mathbf{F} + \mathbf{H}, \tag{3.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{3.2}$$

where ρ is the fluid density, v is the kinematic viscosity of the fluid, $\mathbf{F} = (f_x, f_y)$ are the external forces acting on the fluid, and $\mathbf{H}(x, y)$ is a function that returns the results of HSP pressure information[2].

This equation covers a wide variety of fluid phenomena and can be seen to express the four standard components of any fluid simulation: advection, pressure, diffusion, and external forces.

Advection, a term we use frequently, is defined (in fluids), to be "the velocity of a fluid [that] causes the fluid to transport object, densities, and other quantities along with the flow." [15, p. 642]. We often use this term more generally to mean the transportation of one portion of an object to another location within that object, or moving under an external velocity [2, p. 13]. Ie, we could advect coloured dye inside water by moving the water.

[1] Note, for the purpose of brevity, and completeness, the Navier-Stokes equations introduced in the following sections will include the components from the HSP simulation. However, wherever appropriate it will be noted what changes will need to be made in order to perform the initial non-HSP affected solver step.

[2] Should be omitted during the initial non-HSP pre-compute phase.

The motion of the water itself will transport both the water and dye throughout the fluid. We can see this advection component expressed by the first term on the right-hand side of Equation 3.1.

The pressure component of a fluid solver simulates how the fluid moves. The equations being represented here are built around the notion that fluids are largely incompressible, which makes modelling this pressure component easier. The idea is that if a force is applied to the fluid this force does not instantly propagate throughout the fluid, rather any high pressure area transmits its higher pressure to low pressure areas in an attempt to equalize the pressure. These pressure transmissions take the form of changes in velocity within the fluid volume, and are represented by the second term on the right-hand side in the above equation.

Diffusion is a somewhat complex term that owes its roots to the viscosity of a fluid. The diffusion component of a fluid solver is an attempt to capture "diffusion of momentum", meaning the rate at which the velocities (and pressures) within the fluid come to a standstill. The complicated part of diffusion is in the fact that in most cases diffusion has come to denote an intermingling of molecules, ions, etc., or the mixing of two or more substances. Consider the dye we injected into the fluid in the advection definition. As the dye is advected throughout the fluid it becomes faint and will eventually dissolve and slightly tint the overall fluid volume. This is generally what we consider diffusion, however in fluid solvers this can be seen as a direct result of the viscosity of the fluid. The more viscous the fluid the slower diffusion takes place, which is why this component affects the fluidic velocities, and can be seen as the third term on the right-hand side in the aforementioned incompressible flow equation.

The final component modelled in the Navier-Stokes fluid solver are any external forces that may happen to be acting on the fluid. These forces can include both local forces (such as a floating object in the fluid), and global forces (like gravity and air pressure). These external forces are modelled via the fourth component on the right-hand side in Equation 3.1.

3.3.1 Solving the Navier-Stokes Equations

As previously mentioned, this section is meant to provide only a basic understanding of one technique for solving the Navier-Stokes equations. The math presented throughout

section 3.3 is based directly on papers published by Stam [47], Fernando [15], and others [2, 5, 14, 25, 45, 46, 54]. For a more detailed discussion of the mathematics required to solve the Navier-Stokes equations please see one of the aforementioned works.

The next task in building a Navier-Stokes based fluid simulation is to solve the equations for the system we are interested in. We will use numerical integration to solve them over a number of steps. There are several different ways to break up and integrate the Navier-Stokes equations, all with different properties. However, we chose to base our fluid simulation on work done by Stam in 1999 [45] and subsequently in 2003 [47], where he presented a method for an inherently stable yet efficient NS based fluid solver. For our purposes efficiency is the primary concern, but a stable simulation is also a hard requirement.

Helmholtz-Hodge Decomposition

In order to decompose our base NS equations into something that is easily solved computationally we must first transform the equations. Note that Equation 3.1 leaves us with three equations to solve for (velocity, $\mathbf{u}$, viscosity, w, and pressure, p). In order to help the transformation process we convert the vector field into a sum of vector fields using the *Helmholtz-Hodge Decomposition Theorem* [3] that says a vector field $\mathbf{w}$ on a region in space, D (the plane on which the fluid is contained), with smooth (differentiable) boundary ∂D, can be uniquely decomposed into the form more commonly known as the *Helmholtz-Hodge Decomposition Theorem*:

$$\mathbf{w} = \mathbf{u} + \nabla p, \tag{3.3}$$

where $\mathbf{u}$ has zero divergence and is parallel to ∂D (i.e., $\mathbf{u} \cdot \mathbf{n} = 0$ on ∂D).

The result, a new velocity field $\mathbf{w}$ can therefore be comprised of the sum of any two other vector fields where one is a divergence-free[4] vector field, and the other a gradient of a scalar vector field[5]. Helmholtz-Hodge provides two very useful properties for the purpose of solving the NS equations:

1. Recall that the Navier-Stokes solver we are using takes four distinct components as input in order to solve the fluid system: advection, diffusion, the application of

[3] See Chorin and Marsden in 1993 [7]. Chorin was one of the first people to develop an algorithm for numerically solving the Navier-Stokes equations.

[4] A divergence-free vector field is one without any sources or sinks, no change in density (also known as solenoidal)

[5] A gradient scalar field is a vector field which points in the direction of the largest increase in rate of the scalar field, where the field's magnitude is the greatest rate of change of the entire vector field.

external forces, and an external pressure component. Another requirement is that at the end of each time-step we need to have a divergence-free vector field as a result. Since the result of Helmholtz-Hodge is a new velocity field, $\mathbf{w}$, that has a non-zero divergence, we deduce that we can make the necessary correction to our fluid system velocity field by subtracting the gradient of the resultant pressure field:

$$\mathbf{u} = \mathbf{w} - \nabla p \tag{3.4}$$

2. At this point we still need to compute the NS pressure field, and Helmholtz-Hodge gives us a way to do that. By applying the divergence operator to Equation 3.3 we get:

$$\nabla \cdot \mathbf{w} = \nabla \cdot (\mathbf{u} + \nabla p) = \nabla \cdot \mathbf{u} + \nabla^2 p \tag{3.5}$$

Now, recall that by Equation 3.2 we know that $\nabla \cdot \mathbf{u} = 0$, Equation 3.5 then simplifies to:

$$\nabla^2 p = \nabla \cdot \mathbf{w} \tag{3.6}$$

This results in a Poisson Equation called the Poisson-pressure equation. We will solve for this later because we get another Poisson equation (for viscosity / diffusion) from a future step and it will be convenient to solve for both simultaneously.

For now, let us assume we have our divergent velocity field, $\mathbf{w}$, and we solved for p via Equation 3.6. We can then use $\mathbf{w}$, and p to compute the divergence-free field, $\mathbf{u}$, with Equation 3.4.

We still need to be able to actually compute $\mathbf{w}$ before we can use it. Again let us use Helmholtz-Hodge to define a projection operator, $\mathbb{P}$, that projects a vector field ($\mathbf{w}$), onto its divergence-free component ($\mathbf{u}$). Then let us apply $\mathbb{P}$ to Equation 3.4:

$$\mathbb{P}\mathbf{w} = \mathbb{P}\mathbf{u} + \mathbb{P}(\nabla p) \tag{3.7}$$

However, if we notice that by the very definition of $\mathbb{P}$, that $\mathbb{P}\mathbf{w} = \mathbb{P}\mathbf{u} = \mathbf{u}$, we can simplify to produce $\mathbb{P}(\nabla p) = 0$. Then we use this technique to simplify the Navier-Stokes equations; first we apply the projection operator $\mathbb{P}$ to both sides of Equation 3.1:

$$\mathbb{P}\frac{\partial \mathbf{u}}{\partial t} = \mathbb{P}\left(-(\mathbf{u} \cdot \nabla)\,\mathbf{u} - \frac{1}{\rho}\nabla p + v\nabla^2 \mathbf{u} + \mathbf{F}\right) \tag{3.8}$$

We can then simplify even further because $\mathbf{u}$ is divergence-free. The derivative on the left hand side is then also divergence-free and because $\mathbb{P}(\nabla p) = 0$ the pressure term can be removed and we have the following equation:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbb{P}\left(-\left(\mathbf{u} \cdot \nabla\right)\mathbf{u} + v\nabla^2\mathbf{u} + \mathbf{F}\right) \tag{3.9}$$

Equation 3.9 is a good representation for Stam's Navier-Stokes solver. Scanning the equation from left to right (following parenthesis and order of operations) we can see that we first perform advection, then diffusion, and then the application of external forces. These computations result in the divergent velocity field, $\mathbf{w}$, which use use to apply our projection operator to and we get the desired divergent-free field $\mathbf{u}$. As previously mentioned we then solve the Poisson equation from Equation 3.6 to get p, and subtract the gradient scalar field from $\mathbf{w}$.

It should be noted that real world implementations of a Navier-Stokes solver will typically not use a direct translation of Equation 3.9. Generally, the equation will be broken down into four steps: advection, diffusion, force application, and finally projection. Every simulation time step will consist of a vector field being perturbed by each of these four components which all take as input another vector field and manipulate it based on the time delta.

Fluid Advection

Once we have applied Helmholtz-Hodge we still need to advect the fluid, or find some means to transport both the fluid and its contents through the fluid volume. Remember that we have discretized the fluid volume by breaking it up into a number of distinct fluid cells. The natural and obvious thing to do would be to merely advect each cell by computing its new location based on an interpolation of the grid cells' velocity field components (i.e., simply move cell c_{ij} by velocity $\mathbf{u}_{ij}$).

This method had been used almost exclusively prior to Stam's work in 1999 [45], but any system that uses such a forward-solving method is ultimately prone to instability. The reason is that since the next state of a fluid simulation is affected by the time delta, the more that $\triangle t$ varies from simulation step to simulation step, the more inaccurate the simulation becomes. Simulations that use such a method retain higher accuracy when $\triangle t$ is small, and as $\triangle t$ gets increasingly large the simulation becomes less and less stable, and can

eventually lead to simulation explosions.

What Stam did in 1999 [45] was to use implicit integration. Instead of advecting the fluid using the traditional forward-tracing method of tracing cells from one position to the next over discrete time intervals, we trace quantity paths through each cell back in time to its last known position. We then grab the cell's quantity, q, from the current position and place them in the starting grid cell so that they are always available and known. This can be seen in the advection equation introduced by Stam [45, p. 4]:

$$q\left(\mathbf{x}, t+\triangle t\right) = q\left(\mathbf{x} - \mathbf{u}\left(\mathbf{x}, t\right)\triangle t, t\right), \tag{3.10}$$

where, $\mathbf{u}\left(\mathbf{x}, t\right)$ is the cell's current velocity, and $-\mathbf{u}\left(\mathbf{x}, t\right)\triangle t$ is the vector that we use to translate the quantity back through time by the amount specified in $\triangle t$. Note that the inverse advection step will almost certainly leave the quantity somewhere in between four grid cells, so in order to deal with this the result is simply bilinearly interpolated from the four neighbouring grid cells (see Figure 3.1).

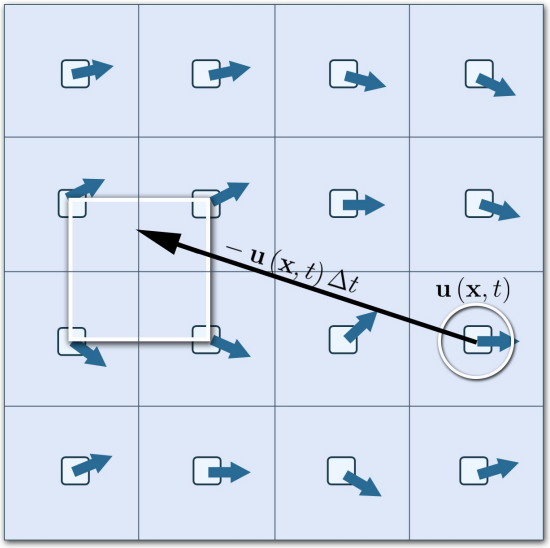

Figure 3.1: Stam's inverse advection step at work. The quantity at $\mathbf{u}(\mathbf{x}, t)$ moves backward through time by vector $-\mathbf{u}\left(\mathbf{x}, t\right)\triangle t$, and the resulting position is bilinearly interpolated from the four neighbouring cells it ends up near. Adapted from: [15, p. 648].

Diffusion

As previously stated, the diffusion step is required in order to simulate the viscous nature of fluids. All fluids have some amount of resistance to flow, and although the relationship

between dissipation of energy and velocity is quite complex in real life (a completely accurate model would require simulating many factors such as temperature, particulate in the fluid, friction, shear stress, etc.) we can simplify the simulation based on the idea that the overall velocity of the cells should ideally want to approach zero, given a closed system. In doing so we can represent fluidic diffusion via the following partial differential equation:

$$\frac{\partial \mathbf{u}}{\partial t} = v\nabla^2\mathbf{u} \tag{3.11}$$

We now solve the equation and represent it in a way that is easier to compute numerically:

$$\mathbf{u}\left(\mathbf{x}, t + \triangle t\right) = \mathbf{u}\left(\mathbf{x}, t\right) + v\triangle t\nabla^2\mathbf{u}\left(\mathbf{x}, t\right) \tag{3.12}$$

This equation will work by itself, however this is the explicit method, and as we noted with the advection step could lead to instability when large values are used for $\triangle t$. We therefore substitute the above equation for the one proposed by Stam which is stable for any (and all) time delta and viscosity combinations [45, p. 3]:

$$\left(\mathbf{I} - v\triangle t\nabla^2\right)\mathbf{u}\left(\mathbf{x}, t + \triangle t\right) = \mathbf{u}\left(\mathbf{x}, t\right), \tag{3.13}$$

where $\mathbf{I}$ is the identity matrix.

This equation is another Poisson equation, and once solved will give us our system's viscous diffusion component. As the reader may recall our *Helmholtz-Hodge Decomposition* left us with the Poisson-pressure equation which needed solving, along with this second equation.

Solving the Poisson Equations

The two equations we need to solve (Poisson-pressure, and viscous-diffusion), are easily solvable with any known Poisson technique (of which there are many), and Poisson solvers are common enough that many pre-existing implementations and libraries already exist; Stam, for example, used FishPak [22] in *Stable Fluids* [45, p. 5]. As such we will only outline the basics of Poisson solving, as it is useful to understand these steps while implementing the fluid simulation algorithm. That being said, as with the rest of this section, the methods presented here are by no means meant to be complete or to provide an in-depth look into solving the Poisson equations. There are many ways to solve them, with varying degrees of efficiency and complexity.

The general idea behind Poisson solvers is to find a way to iteratively solve the partial differential Poisson equation:

$$\nabla^2 p = f \tag{3.14}$$

Rather than try to solve the differential equation we start with an approximate solution and attempt to improve the result over a number of iterations.

One of the simpler methods, known as Jacobi iteration, is a useful technique for performing these iterative approximations. While it converges on a solution slower than the more advanced techniques it has the advantage that we can easily solve both equations with it, and it is also easily parallelizable which, on today's processors, can lead to faster performance than some of the more advanced techniques that do not easily lend themselves to parallel implementations.

We can discretize both Equation 3.6, and Equation 3.13 with the Laplacian operator:

$$\nabla^2 p = \frac{p_{i+1,j} + p_{i-1,j} + p_{i,j+1} + p_{i,j-1} - 4p_{i,j}}{(\triangle x)^2} \tag{3.15}$$

in order to rewrite into the following form:

$$x_{i,j}^{(k+1)} = \frac{x_{i-1,j}^{(k)} + x_{i+1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i,j+1}^{(k)} + \alpha b_{i,j}}{\beta}, \tag{3.16}$$

where α and β are constants, and x, b, α, and β are different for the two equations we are interested in. Refer to the following table for the Poisson equation substitution rules:

Equation	x	b	α	β
Poisson-pressure	p	$\nabla \cdot \mathbf{w}$	$-(\triangle x)^2$	4
Viscous-diffusion	$\mathbf{u}$	$\mathbf{u}$	$\frac{(\triangle x)^2}{v \triangle t}$	$4 + \alpha$

Table 3.1: Substitution table for the Poisson equations. The formulae for Poisson-pressure and Viscous-diffusion utilize can be solved in the same way if the above substitution rules are followed.

Now we have the means to solve both Poisson equations, and the Navier-Stokes solver is nearly complete. Next we need to ensure that the solver can handle boundary and object collision conditions.

Boundary Conditions

In a fluid simulation having the ability to handle boundary conditions is extremely important because it provides the ability to model interesting fluid phenomena such as closed systems (i.e., fluid in a box or cup), and systems with obstacles. In our case we use boundary conditions to define where the fluid can and cannot flow; essentially we use fluid boundaries to determine the path of the river, as well as to handle any obstacles that may be present within the river (such as sand-bars, large boulders, etc.). Fortunately, one nice feature of using a Navier-Stokes grid based solver is that dealing with boundary conditions can be done with relative ease.

The basic observation that governs boundary conditions in a Navier-Stokes based simulation is that fluid reacts to boundaries in a way that most closely resembles what is commonly known in physics as "no-slip friction," which dictates that the velocity should equal zero at boundary cells. Since we are also dealing with pressure in a fluid simulation we need to account for that as well, so we use the *Neumann Boundary Condition*, which is defined as:

$$\frac{\partial p}{\partial \mathbf{n}} = 0, \tag{3.17}$$

over ∂D, where $\mathbf{n}$ is the boundary normal. This means that at any boundary, the pressure must have a rate of change equal to zero (in the direction of the boundary normal). So we know that at any boundary the velocity and pressure must equal zero in the direction of the boundary normal. This is key to the algorithm, and although not an issue, it must be noted because it means that any boundary cell can act as a boundary in one specific direction. Therefore the smallest possible fully enclosed boundary consists of four cells with boundary normals extending out in a diamond pattern (from the center of the square to the furthest edges of the cells).

It is also worth noting that boundaries are defined to exist at the edges of the cells whereas the regular fluid computations use the center of the cell as a reference point. We therefore need to calculate an average of the two cells adjacent to the boundary in order to figure out how equalize the velocity and pressure. In order to solve for the boundary velocity we use the following equation:

$$\frac{\mathbf{u}_{a,j} + \mathbf{u}_{b,j}}{2} = 0, \text{ for } j \in [0, N], \tag{3.18}$$

where a is the first grid cell being averaged, b is the second, and N is the grid resolution.

All we do in order to solve the equation is set $\mathbf{u}_{a,j}$ to $-\mathbf{u}_{b,j}$ (or vice versa depending on the situation).

We do something similar to what we did with the velocity boundary condition in order to ensure that the *Neumann Boundary Condition* holds; we use the forward difference approximation of the derivative:

$$\frac{p_{a,j} - p_{b,j}}{\triangle x} = 0. \tag{3.19}$$

We then simply solve for $p_{a,j}$ or $p_{b,j}$ and apply the result to the cells adjacent to the boundary.

This procedure must be completed once for each simulation step which means a simulation scenario containing large numbers of boundaries might have reduced performance. Recall the river path finding algorithm (Algorithm 1); it figures out where the river should be flowing and creates appropriate boundary cells. As can be seen from the pseudo-code, the algorithm makes no attempt to discover enclosed boundary cells and mark them as non-boundaries in order to save computation time.

This means that any non-boundary cell will always be part of the fluid simulation, meaning its velocity, pressure, and density will be advected. In order to ensure that enclosed boundary cells are not detrimental to the performance of the simulation we add a simple check: if a boundary cell is enclosed on all four sides by other boundary cells then it does not need to be computed either as a boundary cell or as a fluid cell. This simple observation allows very large simulation spaces with comparatively small amounts of fluid on them to be handled efficiently. In our case, since we are dealing with vast terrain sections with rivers running through them this type of optimization is necessary.

Achieving a Constant Flow with Navier-Stokes

We now have a fully working Navier-Stokes solver which is one of the major components of the system, however there are a number of issues inherent in simulating a large dynamic flowing fluid when not all of the fluid can be simulated at once. The basic problem is that in order to model an open system accurately (or at all) the simulation needs to be primed with large areas of off camera fluid in order to generate the necessary momentum inherent in an open system.

This can be seen in the scenario where a section, or sections, of a river flow uphill. There is no problem modelling this behaviour if the water can be simulated from the source

(which is going to be at a high elevation) to the sink (at a low elevation) because the built up force and pressure from the massive body of water flowing from a high altitude has no problem flowing up and over inclines. However, recreating this behaviour in a physically-based fluid system where only small sections of the river can be simulated at a time poses a significant problem.

Moreover, it is not possible to accurately simulate a river, via Navier-Stokes, by simply placing a source at one end and a sink at the other. In order to achieve the desired flow rates with the huge bodies of fluid typical of rivers the source and sink have to be made to have such large positive source and negative sink pressure values that the simulation at the source and sink ends of the river will not look like a river. There will be a distinct bulge or dip in the fluid surface in best case, and in the worst, some odd artifact of pressure values that approach infinity.

For these reasons we need a method for achieving a constant flow with our NS solver, and the method we came up with is based on a technique that is commonly used in rigid-body physics engines: the application of impulses. We apply small impulses to each solver cell in order to keep the flow at the desired rate. A number of questions arise in doing this, however. For example: in what direction, and with what force should an impulse be applied?

Prior to a full discussion on driving the simulation with impulses we need to first integrate our 3D pressure data from the HSP simulation with the NS solver as it will become important when applying the impulses. We want to ensure that the river surface can be seen to have varying rates of flow in any given cross-section so we first calculate the 3D information, and then pass that off to the impulse flow driver (see Section 3.5 for an in-depth discussion on the constant flow algorithm).

3.4 Hydrostatic Pressure Columns

The Hydrostatic Pressure column (HSP) method, as first implemented, was originally a 2D simulation only [23, 39]. It was later adapted to a pseudo-3D method (having a limited resolution in the third dimension) by Mould and Yang [36], so it generally satisfies all of the requirements we need in a fluid simulation: some level of 3D information can be obtained, yet it is efficient enough for use in real-time settings.

However, there are a couple of issues with the technique that keep it from being able

to be used as the primary fluid simulation technique in our system. As the name suggests, HSPs operate using the laws of hydrostatics (i.e., the laws fluid at rest), so it is not suitable for use in situations where constantly flowing fluids (such as rivers) are being modelled. Due to the nature of the hydrostatic laws, HSPs are not capable of modelling fine grain details such as vortices or viscous shear stresses [19, p. 16]. The technique also suffers from instability issues that get more prevalent as the fluid volume increases (and the simulation frame rate drops). For these reasons the HSPs are not used to simulate the main fluid volume, but they are still useful in giving us an important portion of the final results.

A side effect of building an HSP grid is that, at any distinct point on the related height field, one can immediately obtain the fluid pressure at either the fluid surface or any point inside the volume (by interpolating the pressure from all columns that intersect or neighbour the point). Because of this it becomes trivial to account for changes in fluid flow based on terrain features which, to the HSP solver, are merely differences in pressure.

After the Fluid Flow Path has been constructed, and the Navier-Stokes solver has produced an initial flow that is unaffected by any 3D pressure information, the next step is then to create an HSP grid, and use it to compute the pressure value at each grid cell in the fluid volume. Since the rivers we are simulating are mostly static in terms of their flow path we do not need to re-calculate the HSP grid often. If the river path does not change (or no large obstacles are introduced into the scene) the pressure results from the hydrostatic pressure columns will remain largely constant, however the HSP technique is efficient enough to run in real-time in conjunction with the primary fluid simulation, on current hardware, if it is run in parallel or on the GPU. This is the primary reason the technique was chosen; our goal was to create a method that could easily be used to create a completely interactive fluid surface and the use of HSPs as the source of three dimensional fluid flow data allows for this.

As previously mentioned, Hydrostatic Pressure Columns work via the laws of hydrostatics. The fluid volume is discretized into small chunks, or columns, and the pressure of each column is calculated by taking into account the pressure values of neighbouring columns. In the 3D version of the algorithm each column is split into cells and the process is the same except each cell must take into account the pressure of neighbouring cells rather than columns. Since the laws of hydrostatics dictate that fluid will always attempt to equalize its pressure, we know, based on parameters such as fluid viscosity, air pressure,

etc., at what rate fluid will flow from one cell to another based on their respective pressure values.

In an HSP simulation movement of fluids between cells is accomplished by connecting the cells to their neighbours via a series of pipes (see Figure 3.2). These pipes can have different volumes and flow rates depending on the scenario, however most commonly the pipe sizes (and thus pipe flow rates) are determined by the overlapping cross sectional area between pipes. For example, Figure 3.3 shows that some cells have multiple neighbours overlapping a single side. In the case where there are multiple cells touching one side of a cell, a pipe is created for each overlapping cell, and the size of the pipe is determined by the amount of overlap (the greater the overlap, the greater the pipe).

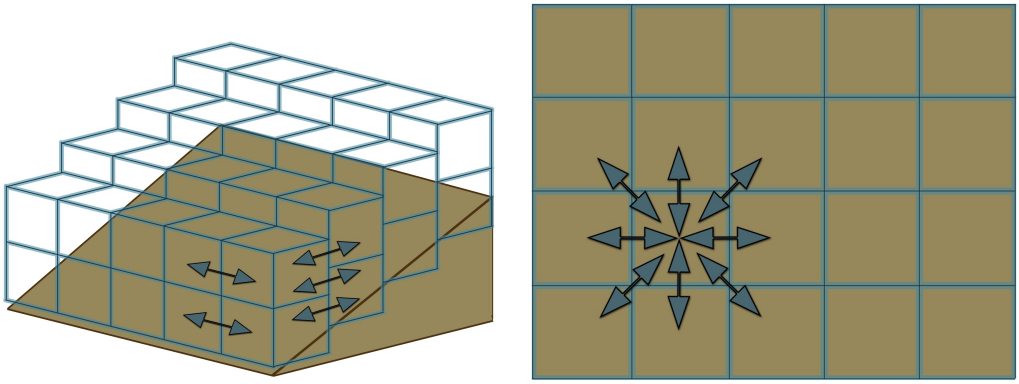

Figure 3.2: Hydrostatic Pressure Column grid showing individual cells (clear), pipes (arrows), and sloping terrain (brown), from two angles: **left** shows above and to the side, and **right** is as seen from the top. Adapted from: [19].

This pipe size is called the pipe cross-section and is defined as:

$$C_{pipe} = (h_{top} - h_{bottom}) \times d, \tag{3.20}$$

where d is the length of the pipe (equal to the grid spacing), and the tops and bottoms of the pipes are the `min` and `max` of the tops and bottoms of the cells into which the pipe flows:

$$h_{top} = \texttt{min}(a_1, a_2),\ h_{bottom} = \texttt{max}(b_1, b_2), \tag{3.21}$$

where a_n is the height of the top of cell n, and b_n is the height of the bottom of cell n.

These pipe cross sections need to be re-calculated at every simulation step because the cross sections will change as the pressures in the cells (and columns) change, and are used solely to calculate the volume transfer (i.e. fluid flow) between cells and thus create the fluid

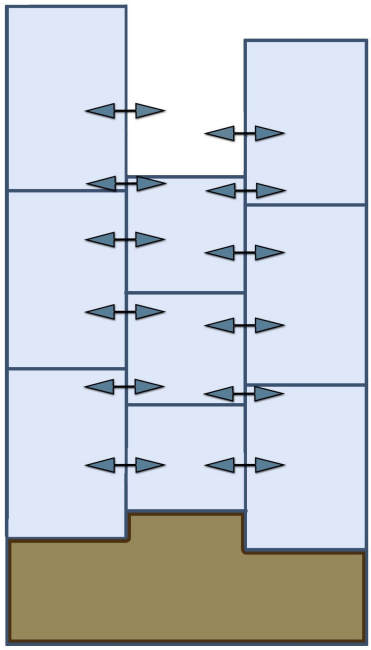

Figure 3.3: Hydrostatic Pressure Column grid showing individual cells (blue), and associated pipes (arrows), from the side. Adapted from: [36].

simulation. However, before calculating any volume transfer we need to first calculate the pressure of each cell, which is determined by the hydrostatic pressure law as follows:

$$p = h_{ij}\rho g + p_0 + p_{e_{ij}}, \tag{3.22}$$

such that h_{ij} is the height of column at position (i, j), ρ is the density of the fluid, g is the force of gravity, p_0 is the atmospheric pressure, and p_{eij} is an external pressure being exerted on the column at position (i, j) due to rigid body interaction or an impact on the fluid surface.

Once all of the column pressures are computed and the pipe cross sections are determined, the next step is to calculate the flow velocity and resultant flow volume through the pipes using the pressure differences between cells. The velocity through any given pipe is given as:

$$n = f\mathbf{u}_0 + \Delta t\frac{(p_{head} - p_{tail})}{\rho}, \tag{3.23}$$

where f is a friction coefficient, $\mathbf{u}_0$ is the flow velocity from the previous time step, Δt is the time delta, p_{head} is the pressure of the cell at the head of the pipe, p_{tail} is the pressure of the tail cell, and ρ is the density of the liquid. Note that the friction coefficient f (introduced by

Mould and Yang [36]) efficiently adds a viscosity component and allows excess pressures to escape the system. This friction parameter is not physically accurate, but helps improve simulation stability, and accounts for viscosity.

The volume of fluid that passes through a pipe for a given time delta is defined as:

$$v = \Delta t * n * c_{\text{pipe}}, \tag{3.24}$$

where c_{pipe} is the cross sectional area of the pipe (Equation 3.20).

Fundamentally this is all that is needed to construct an HSP based fluid simulation, however in practice there is one final step that remains. During the simulation, due to various reasons, including floating point rounding error, and varying time deltas, there will be cases where the fluid volume transfers will cause one or more cells to have a negative volume. This needs to be avoided, because if more volume is allowed into the system than is actually present, it is possible that a simulation explosion could occur.

In order to compensate for this problem the total outflows from each cell are first calculated, then each cell is checked to see if it will contain negative volume if the outflows are applied. If any cell would contain a negative volume all the pipes flowing out of the cell are scaled back by an amount equal to a percentage of the total cell outflow and the negative amount of fluid that the cell would contain. This process is continued until all cells contain a positive amount (or zero) fluid.

Note that it is important to first perform a complete out-flow calculation before attempting to volume correct. If volume corrections are attempted on the fly (i.e., if a pipe overflows a cell, immediately reduce its outflow), the simulation will not be physically accurate. All the outflows should be calculated first, because this simultaneously produces the inflows, and a cell is only considered overflown if its outflow is greater than its current volume plus its inflow. Also, the pipes that get scaled back during an overflow must be done such that, for example, if a 10% overflow has occurred each outflowing pipe is scaled back by 10% of the amount that flowed through it, as opposed to 10% of the total overflow. It is not correct to scale back each pipe by 10% of the overflow because this could reduce the flow of some pipes to a negative value, which will cause a simulation inaccuracy or explosion, and it makes more sense to scale the pipe back by the amount that actually flowed through it. Ie, if a cell previously contained 100 units of fluid and overflowed by 50 units, and some pipe contributed 75 of the 150 units of total outflow, then it should be scaled back by 50%.

We now have all the necessary information to create a full HSP simulation, but for our purposes we only need to update the HSPs whenever the river path needs to be changed, which is usually only once, at river construction. While running experiments we observed that there was little deviation between HSP pressure data from one simulation state to the next, which is logical. A largely static river will not see major pressure changes occur over short time spans, so rather than re-compute the HSP simulation at each time-step we decided to modulate the pressure information between real HSP updates via a random function, $\bar{p}$, in order to approximate the subtle pressure changes that would occur were the simulation being re-computed at every time-step:

$$\bar{p}_{ij} = p_{ij}\texttt{rand}(r_{\text{min}}, r_{\text{max}}) + p_{ij}, \tag{3.25}$$

Where p_{ij} is the pressure at column (i, j), and `rand()` returns a random value from r_{min} to r_{max}. For our simulation we set r_{min} to -0.05, and r_{max} to 0.05, such that the value of $\bar{p}_{ij}$ will be plus or minus five percent of what p_{ij} was calculated to be in the last HSP update. Note that it is not recommended to update $\bar{p}$, from $\bar{p}$. This could create a situation where $\bar{p}$ will differ from p more than the prescribed range of plus or minus five percent. Other values were tried as ranges for the random function, and through observation it was deemed that the ten percent range was optimal for our scenarios.

It should also be noted that this step is optional, and if not included does not cause an overt visual detriment to the simulation. Running simulations with and without the random variance produces results that look similar, albeit slightly different. The biggest effect comes from including the pressure information in the simulation, even if it is not varied from frame to frame. As long as the pressure information makes sense it adds to the realism of the river flow. The pressure information at each grid cell is handed off to the 2D Navier-Stokes simulation and is then used only to give the 2D simulation a way for the river bed, terrain details, and pressure differences within the fluid to affect the 2D surface simulation. The inclusion of the random function is merely meant to keep the simulation closer to the results that would be were the HSP columns being updated at each frame, and it does not reduce the performance of the algorithm in a significant way by including it.

One could argue that since we do not update the pressure columns often it would be easy enough to use a full blown free-surface 3D Navier-Stokes solver to generate this information, and while this is true, our aim in building the system was to keep in mind how it might be used in a variety of settings. Using a 3D NS solver on the volume of fluid we

are interested in could take hours to compute on current hardware, and although the data could be cached, it would mean that any hopes of ever updating this information during simulation at run-time would be lost (for example, to handle a case where the river path undergoes an extensive modification from a large obstacle entering its path). Since the HSP simulation is suitable for real-time use it would be possible to have it run in conjunction with the NS algorithm (perhaps on another core) and frame rates would still remain at the interactive level.

3.5 Impulse Driven Navier-Stokes via Pressure Columns

At this point in the fluid simulation pipeline, we have all of the basic pieces we need for a traditional closed environment, or static-flow fluid simulation. In typical fluid simulation scenarios, the fluid solver is most often interested in solving a system that has defined constraints (such as boundaries), or has flows created by source / sink pairings. For large scale constant or dynamic-flow systems, a different approach must be taken because a closed environment will not work.

We initially investigated using sources and sinks to approximate a river flow (because of its comparative simplicity to implement), and found a number of problems with this technique (for our purposes). Introducing a source into the simulation causes a volumetric imbalance around the area where the source is located, meaning the source will result in an increased height in the fluid around the fluid, which will then propagate through the liquid as flow. Even if the height displacement is small, the unwanted side effect of this is that the flow created is not constant over the entire fluid (it is faster closer to the source), so while this may be sufficient for some fluid simulations, it does not produce a realistic flow for a river. A similar scenario can be seen for sinks where the flow rate will also increase the closer the fluid gets to the sink.

What we need is a way to simulate the constant gravity, pressure, and mass driven flow of a river without resorting to techniques (such as source / sink), and without using an expensive full blown 3D fluid solver. The technique we propose is to drive the simulation with impulses that increase the flow velocity by very small amounts at every cell in the simulation grid at regular intervals.

There are, of course, a few immediate problems that can be seen with this technique. For example, what direction should the impulses be applied? The natural inclination is

to say "in the direction the river is flowing," but what if the flow is in opposition to the mean direction of the river (i.e. in a vortex or eddy)? This could (and will) eventually cause a scenario where too many reverse impulses get applied and the simulation ends up reversing the flow of the fluid volume. Another potential idea would be to apply impulses in the direction that each specific cell's velocity is currently pointing. Again, a similar issue arises: if the cell happens to be facing in opposition to the general mean flow of that particular section of river, it can set up a cascading effect where a section of the river changes flow direction, and after that happens the simulation is essentially meaningless.

Another issue is, what should the initial state of the river be, and how are the first few rounds of impulses to be applied? If there is no current or previous state available to glean flow knowledge from, what should the initial rounds of impulses be? In our experiments we found that care must be taken to apply impulses in such a way so as not to overtly influence the flow in improper directions during the bootstrap procedure. The initial state of the river is a special case because at program start-up we have to go from having a river with zero flow and a large volume of water to the same large river with the desired flow properties. In real-life situations this will likely almost never occur, so physically modelling it is both difficult and not worth while. Since it does not happen in real-life and is not a part the fluid simulation itself, our goal was merely to bootstrap the river so that the natural flow produced by the underlying fluid simulation algorithm would take over.

Before applying the impulses, the initial case must be handled (the flow needs to be bootstrapped). For an overview of the algorithm see Algorithm 2. The idea is to propagate an impulse field across the simulation grid over a number of steps starting with an initial impulse hint from the user at one edge of the river (the basic direction the river should be flowing at that edge). The first time step causes the impulses to be applied and the fluid is solved for that time-step. At any cell where the velocity is not zero, that velocity vector is normalized and saved as the *flow hint* for that particular cell. Because the flow hints are normalized from the velocity they become unit vectors that represent direction only. This is repeated until all the cells have a flow hint associated with them (unreachable cells are handled with a special case so as to not introduce a non-termination clause to the algorithm). For a detailed look at the algorithm implementation see Appendix A.2.

As the algorithm runs it sweeps across the river from a user specified source location and adds impulses along the flow path. In the first time step impulses are applied directly

Algorithm 2 Bootstrap River Flow

```
1: FlowHints = Empty 2D Array
2: for all cells c in UserHintedCells do
3:     FlowHints[c.x][c.y] = c.initialFlowDirection
4: end for
5: repeat
6:     Unhinted = 0
7:     computeNavierStokes(SimulationHz)
8:     for y = 0; y < GridHeight; y++ do
9:         for x = 0; x < GridWidth; x++ do
10:            if c.active and FlowHints[c.x][c.y].length != 0 then {Ensure cell is active and
               does not already have a flow hint}
11:                velocity = calculateFlowHint(c)
12:                if velocity.length != 0 then {Make sure we can get a flow hint for this cell}
13:                    FlowHints[c.x][c.y] = normalize(velocity)
14:                else
15:                    Unhinted++
16:                end if
17:            end if
18:        end for
19:    end for
20: until Unhinted == 0
```

at the edge of the river in the direction the user gave as the initial flow direction. The fluid simulation is then computed using those results. In the next time step impulses are applied to the initial cells given by the user and those that were affected by the simulation computation (likely most of those that neighbour the initial cells). The fluid simulation is computed again, and this process continues until all the cells have had impulses applied to them. These initial impulses are saved as the flow hints and will be used later to help in determining how impulses are applied; they are also used to solve the previously mentioned problems we discovered while attempting to employ this method.

After the flow hint field is produced, bootstrapping is not yet completed. In order to finish the bootstrapping procedure we run the simulation with flow hinting enabled until

it is possible that fluid from one end of the river reaches the other end (technically until it becomes possible that one "advection particle" could travel the length of the river, however we have not yet discussed the advection particles – for now it is only important to know that bootstrapping is not yet complete).

Armed with the flow hinting field, and a bootstrapped fluid volume we can now begin adding impulses to each cell in the simulation as it runs. As we do so we need to ensure the simulation stability is not compromised (i.e. complex flows can still be visualized, and the overall flow does not exhibit incorrect flow behaviour such as a river reversing its direction).

In order to accomplish this, the impulse we add to a specific cell is crafted so that the magnitude of the impulse is proportional to the current velocity of the cell in relation to the desired flow volume (this keeps the impulse application scaling linearly and the ratio between one cell and another does not change after the impulses are applied), but the direction of the impulse is modified based on the deviation between the flow hint vector and the current velocity vector.

In order to calculate the cell's directional deviance, d, from the flow hint we use the standard geometric angle difference calculation:

$$d = \texttt{abs}\left(\texttt{rad2deg}\left(\texttt{atan2}\left(v.y, v.x\right) - \texttt{atan2}\left(f.y, f.x\right)\right)\right), \tag{3.26}$$

where v is the cell's velocity, and f is the flow hint.

We then check if d is greater than or equal to the cutoff deviance value (we ran a number of experiments and found 15 degrees to be a reasonable value that worked across a number of different simulations), and if so we calculate an impulse for the cell using the cell's current velocity vector, and the cell's previously calculated flow hinting value.

In addition to the above parameters the most important step in the impulse application is to incorporate the data provided by the hydrostatic pressure columns, which augments our 2D Navier-Stokes simulation with the 3D HSP data, and gives our fluid simulator the appearance of being a full blown 3D simulation. All we do to accomplish this is transform the impulse length (velocity) by the pressure value returned from the cell's associated HSP column:

$$l_{ij} = \parallel \mathbf{u}_{ij} \parallel \frac{p_{ij}}{P} z, \tag{3.27}$$

Where l is the new impulse magnitude for cell at index (i, j), $\mathbf{u}$ is the cell's velocity, p is the HSP pressure, P is the maximum pressure the HSP simulation can contain, and z is a

user defined value such that $0 \leq z < \infty$. By changing z one can change how much the HSP information affects the simulation. For example, setting z to a value of 0.5 will cause the effect that the pressure information has to affect the simulation half as much as if z were set to 1.0. It is possible that z can be set to values greater than one, however we chose to leave influence at 100% of normal ($z = 1.0$). The sum of all these modified impulses is then equal to the desired flow velocity.

We also add in some other factors as well, such as a small random variance which can be modified to cause a more turbulent river, and a maximum and minimum flow check in order to further confine the river flow to a desired look. For a more detailed look at the application of flow hints see Algorithm 3, and Appendix A.1.

For our purposes we chose the random variance factor to be plus or minus one percent, and maximum and minimum flow speeds to be no more (or less) than two hundred times greater than or less than the mean flow speed. It should also be noted that setting the flow speed of the river itself should be done in accordance to the grid cell size of the river in order to maintain grid size independent flow rates. For example, if the desired flow rate is 1.3 meters per second, and the cell width is 1.0 units, the flow rate should be set to one half the value of the same flow rate on a grid with a cell width of 2.0 units (assuming a regular grid).

3.6 Summary

We now have all the pieces needed to simulate a river's fluid volume, from beginning to end. We can calculate the river's path, bootstrap its flow, gather the flow hints, calculate pressure information to coax the 2D simulation into and the appearance of 3D via HSPs, and safely and stably drive the simulation with impulses via the flow hints, but we are still missing an important piece of the puzzle: the ability to render a finely detailed visualization of a river.

Algorithm 3 Apply Flow Hint

Require: c is set to the current cell

Require: FlowHints is a 2D array containing the grid's flow hinting information

Require: FlowSpeed is the desired overall flow speed of the river

CurFlowHint = FlowHints[c.x][c.y]
FlowSpeed = updateFromHSP(FlowSpeed)
if UseRandomVariance **then**
    FlowSpeed *= getFlowSpeedPercentile(c.x, c.y);
end if
if UseMaxScale **and** FlowSpeed.length $>$ MaxSpeed **then**
    c.velocity.scale(MaxScale)
else if UseMinScale **and** FlowSpeed.length $<$ MinSpeed **then**
    c.velocity.scale(MinScale)
end if
if FlowHint.length $>$ 0 **then**
    AngleDiff = `abs`(`atan2`(c.velocity.y, c.velocity.x) - `atan2`($FlowHint$.y, $FlowHint$.x)) * `rad2deg`);
    DiffTrigger = 15.0f;
    DiffTriggerOverTwo = DiffTrigger / 2.0f;
    if AngleDiff $\geq$ 180.0f - DiffTriggerOverTwo `and` AngleDiff $\geq$ 180.0f + DiffTriggerOverTwo **then**
        FlowHint.scale(c.velocity.length)
        c.velocity = FlowHint
    end if
end if

Chapter 4

Texture Advection

With the combination of Impulse Driven 2D Navier-Stokes and the Hydrostatic Pressure Columns we now have the basics for a highly efficient pseudo-3D fluid simulator, however these two techniques alone will not provide a detailed enough fluid surface in order to produce a convincing representation of a river. The problem lies in the sheer volume of water being represented and the computational cost of even this relatively inexpensive approach to fluid dynamics.

On current hardware this technique, without having undergone extensive optimizations or parallelization, can operate in real-time on a 512 by 512 sized grid. This is a large enough size for certain applications, however we wanted to provide a much more finely detailed river surface. Real rivers exhibit minute details in the surface, even for a largely slow and flat river, and it was precisely this type of fluid volume that we wanted to simulate (see Figure 4.1 for example images).

Figure 4.1: Real images showing examples of the highly detailed, yet flat rivers, that we aim to be able to simulate in real-time.

We needed a way to give the fluid surface more detail than the simulation itself can

provide, while still producing a visually realistic result. After much experimentation with different existing procedural techniques it was decided that a simple statistical (i.e., FFT) based approach would not be sufficient. For ocean simulators these techniques are both reasonable and desirable, however in order to simulate a river, water features (such as waves, vortices, eddies, ripples, etc.) must move with the flow and none of the existing procedural techniques provide the results we are looking for.

Procedural wave generation techniques are procedural because they are not intended to work with or simulate in any way a real fluid volume. Therefore, coercing them into moving realistically with a fluid simulation is problematic in a number of ways. Transporting the static wave fronts such that they have the appearance of realistically moving with the flow of the river, as determined by the underlying fluid simulator, is difficult, and the body of research on this topic is small. However, recent advancements in Texture Synthesis and Texture Advection have led to some work in this area.

The basic principle behind Texture Advection is to take, as input, one or more textures and advect them (transport, or morph them over time) based on a series of input parameters. The technique is usually used in other visualization areas and is not typically seen in fluid simulations. Since our goal is to couple the results with the fluid simulator's velocity and pressure information we came up with a solution we call "Advection Particles."

An Advection Particle is essentially an encapsulation of a mathematical deviation function that describes how a particular section (pixel) of an image, or texture, is to be modified. A single particle contains all the information necessary to track the particle through space and time in addition to containing the data required to modify the resultant texture based on the input textures and parameters. These particles will be discussed at length in Section 4.1, so for now we will describe the underlying principles involved in Texture Advection before coming back to the Advection Particles.

The basic idea behind texture advection, or colour advection as it is sometimes referred to, is to perform colour transfer on a volumetric grid. For our purposes this volumetric grid is the same velocity field that is used by the fluid simulator, and we map texture colours to height-values thus transforming the 2D colour-mapped texture into a 3D surface grid. The reason for this is we wish to produce a highly detailed simulation and although many large rivers look primarily flat we obtain better visual results using a 3D surface mesh rather than a 2D colour-mapped one. By actually perturbing the fluid surface rather than

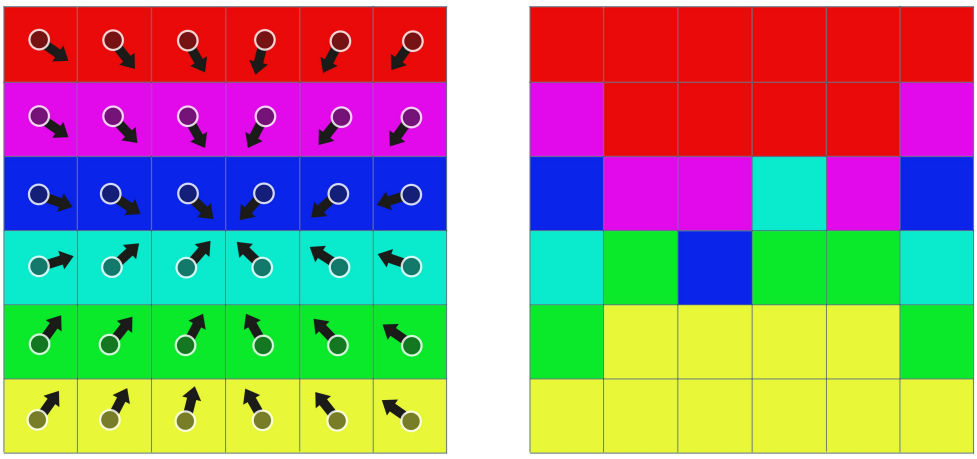

Figure 4.2: An example of texture advection where each grid cell represents a texel, or unit of area, in a texture. The arrows on the left hand image represent the input parameters that affect the output results, and the right image shows the resultant image after being advected. In our case the arrows can be thought of as the velocity components of the underlying fluid simulator, and the circles represent the advection particles.

simply projecting a texture onto a flat surface, it also simplifies the calculations for environmental reflections and refractions, self-shadowing, and other fluid visualization techniques designed to enhance the visual fidelity of the simulation. The downside of this approach is that it requires a large number of polygons in comparison, however we do attempt to address this with level of detail partitioning. Also, since our simulation is a mixture of being both CPU bound (by the fluid solvers) and geometrically bound (by the surface polygon data) it was deemed that this would be the best trade off between quality and efficiency.

In the alternate case (using a 2D colour-mapped surface representation rather than 3D), we could animate the planar texture such that it would appear similar to the 3D one, but without using many polygons. In fact, our method could easily be converted to produce this type of scene, which may be desirable for some applications where the number of polygons must be restricted, but the 3D method produces more realistic visuals, and is also more amenable to direct interaction with the fluid surface. For example, consider a scene where an object is floating in a river. In this situation the extra polygon data used to represent the fluid's height will be beneficial in convincing viewers that the object is actually influencing and being influenced by the river's fluid flow. The ability to see water waves and ripples hitting and reflecting off the object will provide a much more realistic

visualization than the 2D variant which would be unable to produce such effects.

In order to transport our 3D surface representation across the fluid system's velocity field there are two basic problems that need to be solved:

- The resolution of the volumetric grid is significantly lower than our synthesis / advection grid (since we want to synthesize a high resolution animated texture in areas where simulation information in unavailable).
- We also need to know how and where to advect the colour data to (i.e., where does the texture get advected to in one frame in the animation to the next) [26, p. 7].

The actual task of advecting pixel data is quite simple. We use a technique very similar to the same procedure that the Navier-Stokes advection solver uses in order to transport colour from one location to the next. A back-trace is performed, in a manner very similar to that which is performed during the NS advection step, on each point in the volumetric grid to its source location in the previous time step:

$$p\left(\mathbf{x}, t + \triangle t\right) = p\left(\mathbf{x} - \mathbf{u}\left(\mathbf{x}, t\right) \triangle t, t\right), \tag{4.1}$$

where, $\mathbf{u}\left(\mathbf{x}, t\right)$ is the particle's current position, and $-\mathbf{u}\left(\mathbf{x}, t\right) \triangle t$ is the vector that we use to translate the particle back through time by the amount specified in $\triangle t$. Note that, as with the NS advection, this step will likely leave the particle somewhere in between four grid cells, so we perform the same bilinear interpolation from the four neighbouring grid cells to compute the end result (see Figure 3.1).

Once the location is computed the colour is re-evaluated by interpolating from the neighbouring grid points.

4.1 Procedural Wave Generation

Now that we can advect textures we still need to determine how and where the texture values are going to be advected throughout the fluid volume, but before advecting the texture we need an actual texture to advect. For the purpose of generating a visually plausible river surface we use an animated texture that is generated procedurally while the simulation runs.

We experimented with a number of non-animated textures prior to deciding an animated texture was required. The results showed promise but with most of these textures it was clear that a static texture was being advected since the resultant simulation lacked a lot of

the chaotic details present in an actual river surface (see Figure 6.4 for a comparison). We then looked at a number of animated texturing techniques including random noise, Gerstner Waves [4], and an FFT based ocean wave method. Random noise looked too random and simple, but the various ocean wave techniques proved more effective, and we therefore chose to adapt the method presented by Mitchell from ATI Research [35] that is directly based upon Mastin et al. [34] and Tessendorf [49].

Although we have not yet discussed the Advection Particles that we will be using to transport the texture through the fluid simulation and create the river's heightfield, it is worth briefly mentioning the underlying principle so a full understanding of our choice of algorithms is possible. The idea is to advect the waves created by the procedural wave generator in a way that allows individual wave fronts to approximately travel together, yet still respond appropriately to the fluid simulation information. For example, a wave front should flow around bends in a river as dictated by the fluid solver.

The procedural wave generation method presented by Mitchell is flexible enough to allow a number of parameters that can be set to provide different types of waves. This may be important for some rivers (i.e., some are more turbulent than others, etc.), however since we are mostly relying on the underlying fluid simulation to create the proper turbulence, and merely using the procedural technique as an advectee for added surface definition, we were primarily interested in finding an algorithm that produced large quantities of realistic waves with low computational cost.

The underlying principle behind Mitchell's technique is taken directly from Tessendorf's method of summing a series of sinusoids with time-dependent changes in amplitude. The fluid surface is defined as a heightfield where the height of any grid cell, x at a given point in time, t, is:

$$h\left(\mathbf{x},t\right)=\sum_{k}\tilde{h}\left(\mathbf{k},t\right)e^{i\mathbf{kx}}, \tag{4.2}$$

where h is the height field, $\mathbf{k}$ is a 2D vector (k_x,k_y) such that $k_x=\frac{2\pi n}{L_x}, k_y=\frac{2\pi m}{L_y}$, and n,m are integers with bounds $\frac{-N}{2}\leq n<\frac{N}{2}$ and $\frac{-M}{2}\leq m<\frac{M}{2}$.

There are further refinements to the technique outlined by Mitchell and Tessendorf, largely with regards to parallelizing the FFT based algorithm, and adding other details such as directionally controlling the waves via a parameter that attempts to simulate wind. We do not make use of these parameters in our implementation as we want the generated waves to work well in a variety of situations. We primarily utilize the fluid simulation and advection

particle mechanics to produce the desired visual results for the river rather than relying on the procedural algorithm for parameters like turbulence, wind, etc., which are all difficult to set algorithmically under a wide variety of conditions. That being said, there are additional parameters available that can be modified which would provide additional creative control that we did not completely explore.

The output from the wave generation algorithm produces a height field, but as we already discussed, we are advecting textures in 2D space, not surfaces in 3D. This is because our 3D wave generation method is being colour-mapped to a 2D surface for the purpose of texture advection. After the 2D texture advection completes we take the 2D texture results and map them back to a 3D heightfield using a traditional heightmap transformation technique, where colour ranges directly translate to changes in height. In order to avoid two $O(N)$ conversion operations while performing these tasks we converted the texture advection routine to use floating point textures, so there is no conversion required other than two pointer casts.

Now that a 2D texture generation method is defined we can proceed with advecting the animated texture throughout the fluid volume.

4.2 Advection Particles

Our aim is to efficiently, and as realistically as possible, simulate the types of rivers seen in Figure 4.1. The characteristic features include: intricate and chaotic ripple details that can be seen travelling with the various ebbs and flows present in the fluid system, sections of opposing flows, and even calm sections. We have defined methods for calculating the fluid simulation, performing texture advection, and for procedurally generating an animated ocean wave texture. Now we need to supply the texture advection algorithm with both the texture and the parameters for which it should be advected.

The problem at this stage is how do we advect the texture so it appears as if the texture is flowing naturally with the river. As a simplistic example, think of moving a carpet around a curved track, where the carpet is roughly the same width as the track. In order to move the carpet around a bend it will need to contract on the side that is turning into the bend. In sections where the track might be wider or thinner than the carpet other similar issues arise.

Our situation is actually much more complex because rather than moving a carpet (or

texture) around a track we need to simultaneously move every section of the carpet in a different direction, and at a different speed, yet we still want the sections of the surface of the carpet to appear to be travelling together, and to completely cover all parts of the track underneath it at all times.

The procedural wave generation algorithm provides the fine details that we require, but no method by which we can tie them to the fluid simulation. A simplistic approach could, as per the prior example, attempt to simply drag the animated texture along the river path and distort it as necessary. This would produce a scenario where the animated texture would move with the contour of the river at a uniform rate; there would be no portions of the river that are moving at different speeds than others, and it would not be possible to capture details such as vortices.

We require a way to couple the colour advection step to the fluid simulation. Kwatra, et al. used a concept they called "texture energy" to advect a texture through their fluid simulation [26, p. 1], however their technique does not run in real-time. They require anywhere from 60-200 seconds per frame using much smaller fluid volumes than we used [26, p. 10]. The also perform a number of expensive steps that we did not wish to calculate, due to having real-time operation as a requirement for our system.

Once the texture has been generated it is advected using what we call "advection particles." Essentially the texture advection step consists of these advection particles being propagated through the fluid simulation using the results of the velocity, pressure, and density information from the Navier-Stokes simulation (which, in turn, has been affected by the HSP columns).

The particles are advected through the simulation using Texture Advection as outlined above (see Equation 4.1). The Advection Particles are used merely to track data important to both the advection and the propagation of the particles' progress through the simulation. Each advection particle has the following properties:

- Location in the input textures
- Birth location
- Age
- Current location in the river
- Age of death

An advection particle pertains to a specific location in the input textures which does not

change. For our application we are using an animated texture comprised of a number of frames of individual textures. The location in the input textures refers to the same location in each texture.

This particle is then introduced into the fluid simulation and affected by the fluid simulation's velocity and pressure fields so that it moves through the river much like a massless particle of dust would. As it moves it affects the resultant texture by adding its value from the input textures to the output texture. For example, if a the location of particle n in the input texture was $(0, 0)$ and the RGB value of the input texture at index $(0, 0)$ was $(255, 255, 255)$ then the particle would add the value $(255, 255, 255)$ to its current location in the output texture.

Each location in the output texture can be seen as the average of all particles currently occupying that location:

$$O_{xy} = \frac{\sum_{i=0}^{n} f\left(P_i\right)}{n} \tag{4.3}$$

where $f = (P_i)$ is a function that returns the RGB value of the particle's combined input textures, or a transparency value if the particle at position (x, y) does not occupy that location.

Particles are spawned at specific intervals and have limited life-spans; they do not travel infinitely far from their birth location because after travelling a certain distance they become un-grouped from their neighbours and start to resemble noise rather than wave fronts moving in unison. The average life-span of the particles must be set manually with a value that will differ depending on the parameters of the river (how turbulent and quickly it is moving). Our tool provides a graphical particle display (see Figure 4.3) that helps in the choosing of this value. By representing each particle as a colour determined by its birth location it is easy to see how far the particles remain travelling in groups of similar colour. We should also note that we found there to be a large range of values for particle life-span that work effectively, meaning that the system is not very dependent on tuning of this particular parameter. Setting the parameter to a value of up to half that of the maximum value had little impact on the simulation since new particles are born where old particles die, and thus if the old particles are still travelling in groups, the new particle births will approximate the old groupings.

We retain the particles' ages and times of death because these parameters determine

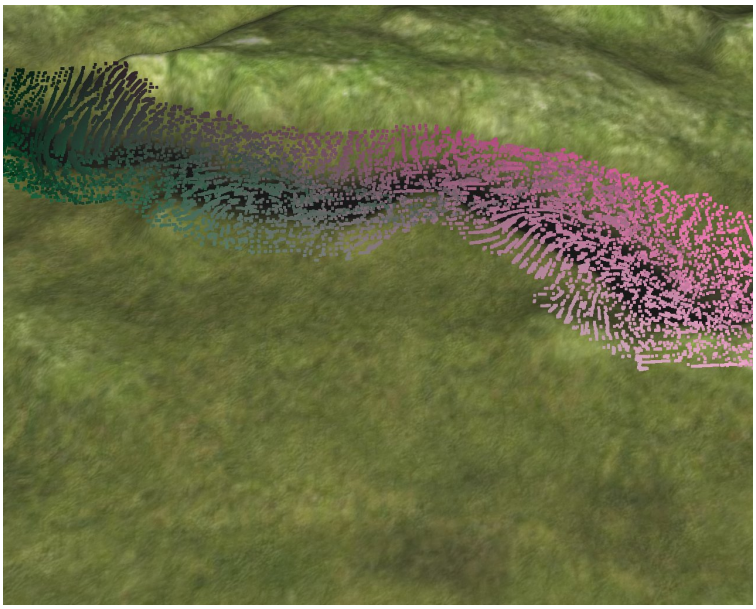

Figure 4.3: Advection Particles being visualized (note: the number of particles has been reduced significantly to improve readability of the figure). The particle colours denote birth location; particles born in areas close to one another have a similar colour. We can see from the visualization that similar coloured particles tend to move together, and even from this informationally reduced diagram we can see some elements of flow among particles.

how strongly the output texture is affected by each particle. For example, if a particle was recently spawned we want it to fade in, and as a particle nears its time of death we want it to fade out until it is invisible right before it dies, thus removing any visual "popping" artifacts from the output. The particle birth time and current location are kept for similar reasons: the farther a particle strays from its birth location the less of an impact it has on the colour summation. We therefore define the particle colour update function to be:

$$f(P) = \frac{\frac{a_c - a_b}{A} + \frac{\|l_c - l_b\|}{L}}{2}, \tag{4.4}$$

where a_c is the current age of the particle and a_b is its birth age. Similarly, l_c, and l_b denote the particle's current and birth locations respectively. A is a user defined constant that is the maximum age of a particle, and L is another constant the determines the maximum distance a particle can be from its birth location without being transparent.

For our simulation we chose A to be 1.5 seconds, and L to be 5% of the total distance of the river. These values were chosen through experimentation with the system which provides a tool for graphically watching particles in order to best choose these values (see Figure 4.3). As previously mentioned we found there to be significant leeway in choosing

good settings for these parameters. Finding a single "optimum" setting for these two user defined constants is not required. Many values will work, but shorter values tend to be better than longer ones.

Note that we use a pseudo-random function that produces values in the range of $[0.5 - 2.0]$ to alter each particle's life span. The aim behind adding randomness to the particle ages is to ensure that the particle updates do not all happen at common intervals and avoids cases where many particles are dying or spawning simultaneously. Without the introduction of some randomness these situations do occur and are visibly apparent while the simulation is running.

As the particles are advected through the simulation their position is affected, at each step along the way, by the results from the fluid simulation, which has, in turn, been affected by the 3D information obtained from the HSP columns. The particles can be seen moving in varying directions just like dust or sediment would inside a real river (except in two dimensions only).

The particles can also be seen travelling together in groupings where particles instantiated in similar locations are consistently near to one another. In situations where no particle exists in a cell, a new one is faded into existence and is an approximation of all the particles around it (its birth location is set to an average of its neighbours and then rounded to an integer value). This removes the situation where a section of a river may be blank, and also eliminates any visual popping that might occur at the introduction of new particles into the system.

4.2.1 Bootstrapping and Initial Particle States

In Section 3.5 we discussed the bootstrapping procedure for the impulse driven Navier-Stokes algorithm, but mentioned that bootstrapping was not yet complete. The only task remaining at this point is to populate the advection particle list with an initial set and then bootstrap the system so that the viewer is not presented with an uninitialized simulation state.

In order to accomplish bootstrapping we first assign an advection particle to every cell in the grid. We then run the simulation for some duration deemed long enough for the simulation to reach a stable state. Rather than define some predetermined amount of time we run the simulation until every one of the initial advection particles has died and respawned

at least once.

4.3 Summary

Results of the Texture Advection step add the desired realism to the fluid surface and the coupling between the texture advection and fluid simulation appears natural. The algorithm produces a highly detailed surface construction that approximates the chaotic and complex relationship that river fluid volumes have with their surface, while remaining fast and efficient to produce and render. The particles can clearly be seen flowing with the simulation results (see Figure 4.3), and when the final output is being rendered, the original texture can be seen being advected through the fluid simulation resulting in the finely detailed fluid surfaces that can be observed in real rivers (see Figure 6.5).

Chapter 5

Rendering

Our primary goal is to achieve better results than existing real-time river simulations, and while rendering was not our main focus, we still wanted to provide a reasonable visual approximation that would both showcase the fluid simulator, and prove that the method would be efficient enough to be coupled with a real world rendering architecture suitable for use in games or other commercial applications.

There were a number of issues that we had to solve regarding rendering, including the fluid surface representation or carpet construction (see Figure 5.1), environmental reflections, river particulate and sediment (represented by volumetric fog), terrain generation and rendering, and Level of Detail (LOD) issues for both the fluid simulation and the terrain.

5.1 Fluid Surface Carpet Construction

We designed the fluid solver such that all the individual components use the same grid during the different simulation steps. For example, the Navier-Stokes solver uses the same grid and LOD tessellation to represent the velocity and density vector fields that the HSP solver uses to build the individual pressure columns from. This keeps the simulation largely homogeneous in regard to the conversion from simulation space to 3D space, however there are still a few issues that are worth noting.

The approach we took to building the fluid surface geometry was to convert the simulator grid directly to a 3D heightfield. The heightfield method does have a number of drawbacks, the most severe of which is the inability to represent fluid phenomena that require one portion of the geometry to fold over top of itself (such as cresting waves). For us this was a reasonable compromise to make since our fluid simulation approach does not allow for cresting waves. Moreover, such a representation implies that any LOD or rigid-body physics algorithms we built for the terrain could also be applied to the fluid surface (the terrain is also represented using a height field). The heightfield method is also vastly more efficient than other more complex methods that allow arbitrary geometry as surface

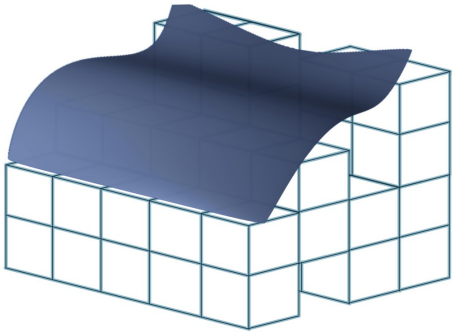

Figure 5.1: Fluid surface being constructed from simulation grid data. Adapted from: [19, p. 5].

representations (such as Marching Cubes).

All of the carpet construction issues essentially revolve around the same thing: building polygons for the fluid surface. Since we needed a highly detailed fluid surface this means we can not waste any polygons for the sake of algorithmic simplicity. Even on modern graphics cards our method has the ability to push the polygon count to the upper limit of what the cards can process in real-time, even without adding rendering improvements that require GPU intensive operations such as shaders and multi-pass rendering. However, we have also included the ability to tune the algorithm so that the polygon count can be tailored to suit a compromise between quality and speed.

When building the surface geometry we took great care in only representing as much surface area as necessary. However, being strict about this presents a number of issues, the first one being the generation of surface normals. In areas around obstacles (such as rocks and sand bars, etc.) and at river bank edges the normals must be constructed from polygons that do not exist. For example, think of the boundary condition case where the boundary is not a smooth or straight line. Calculating the normals along these edges often means changing the normals calculation algorithm so that it picks different triangle vertices under different conditions – this in and of itself is not a serious problem. The real issue occurs while smoothing the normals. We use an 8-way smoothing in order to give the river surface a soft and fluid appearance, however in the boundary condition case there will not be 8 neighbours available. Usually what is done in such cases is to represent the surface in simulation space but not in 3D space, so that the information is available for geometry computations, yet no polygons are wasted. In our case adding this information to the simulation step means increasing the number of simulation grid cells to the point where

it could have an impact on frame rates (depending on the river shape and grid resolution). We therefore create approximated geometry based on an average of the existing geometry (see Figure 5.2).

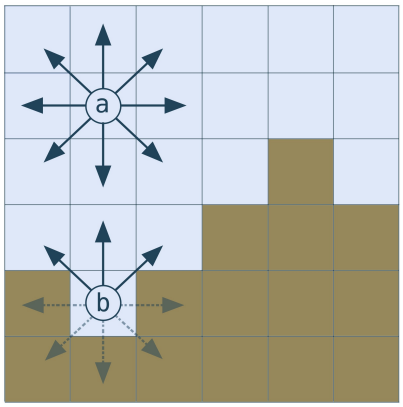

Figure 5.2: Construction of smoothed surface normals where geometry is not available. The blue cells are wet cells, and the brown cells indicate dry cells. The cell labelled 'a' has all eight neighbours available to produce an averaged surface normal from, whereas cell 'b' only has three neighbours available. In the case of cell 'b', we simply set its surface normal to one computed from an average of the three available neighbours. This is not as accurate as alternative approaches, but is more efficient.

In order to accomplish this we use a simple interpolation scheme to introduce geometry points based on those those that do exist and then generate the normals from those points. We start from the first available grid cell and perform a clockwise operation to fill in the missing geometry. Each missing grid cell is interpolated from all existing or already approximated neighbours (up to eight). The result is that the generated normals are not completely accurate, but the results are not easily perceptible and the gains in efficiency are considerable when compared to representing the necessary data in simulation space.

The next problem is ensuring that there were no gaps in the river surface and the terrain. Many terrain engines use dynamic LOD in order to save polygons at large viewing distances, however using dynamic LOD on the terrain means that there might be situations where the river fluid surface no longer matches the terrain (small holes, slits, or gaps can form). In this situation there are a number of solutions, the most ideal of which would be to set the terrain LOD algorithm to treat the entire scene domain (both terrain and river surface) as a single object. This would enable geometry LOD support for both terrain and

river simultaneously, and to take advantage of the terrain LOD algorithm's ability to ensure terrain features remain attached to one another.

In our case, we did not focus on devising improvements to geometry level of detail. Much research has already been done in the area and it is outside the scope of this project. We instead focus our efforts on LOD support for the fluid simulation algorithm itself. That being said we still need to use at least a simple geometry LOD algorithm, and for this we chose a standard quadtree style LOD scheme for the terrain, where the terrain is broken up into recursively smaller patches and further away sections are rendered with less detail. We also chose to clamp the terrain to the river surface so that any gaps would be filled and the river surface height would not be modified by the terrain clamping. We were not concerned if the terrain surface adjacent to the river did not completely match the input digital elevation data, because our primary concern was the fluid simulation, and having minor differences in the terrain adjacent to the river does not impact the river itself (at the point at which the terrain is clamped to the river, the river's path has already been determined). However, there are situations where it would be important to retain complete accuracy in the terrain geometry and in these instances some other more advanced geometry LOD technique will be necessary.

5.2 Level of Detail

Level of Detail support at the fluid simulation level means the LOD algorithm needs to interface with the different parts of the fluid simulation (Navier-Stokes, and Texture Advection) separately. Since our simulation combines a number of methods they will all need to be addressed separately in order to achieve maximum gains from Level of Detail transitioning. Despite this we still wish to simplify the LOD support so that, as much as possible, a single method could be used to at least control the LOD settings and transitions.

Looking at the fluid simulation pipeline it can be seen that all of the individual components share the same simulation grid. We implement this grid as an quadtree, such that at any given time every piece of the simulation has access to the same information regarding which sections of the grid are on screen and how far they are from the viewer. From there it is up to the individual components to alter their settings based this data.

5.2.1 Hydrostatic Pressure Columns

In our simulation we do not dynamically update the HSP columns (they are computed once at startup), we therefore did not implement an LOD scheme for them. However, as we note in the future work section it should not be overly difficult to add this support, and if the HSP columns were to run in conjunction with the rest of the fluid simulation, it would definitely be advantageous to add dynamic LOD support.

5.2.2 Navier-Stokes

Our approach to reducing the cost of the Navier-Stokes computations was similar to what we propose for the HSP columns. Since we have the quadtree structure available to use we simply treat the distant quadrants as single NS cells. Rather than compute all three solver steps (advection, diffusion, density) for each iteration and for each cell, we average the individual cell values according to the quadtree structure. For example, if one quadtree node contains 4 cells, those cells are summed (for all their parts including velocity, pressure, etc.), an average is produced, and these larger cells are then used in the NS computations rather than the individual cells themselves.

There are a few issues with this method, the major one being that the NS method doesn't lend itself easily to irregularly sized grids. For example, what is the correct output when a large cell neighbours a series of smaller cells? Since complete accuracy is not our primary concern we simply choose to treat all cells the same. If a large cell points to an quadtree node that contains a myriad of smaller cells we sum the cells and perform the calculation as if we were computing two larger cells. In the opposite case where we are computing a small cell that points to an quadtree node containing a large cell we simply treat the large cell as if it were a small cell. This means that cell will be computed n times, where n is the number of cells that neighbour it, but these wasteful cases are small in number and only happen on border transitions between quadtree levels. Also, the situations where this occurs is generally far off into the viewing distance where irregularities in the simulation are not easily seen.

The major drawbacks to this method are in simulation instances where a viewer quickly moves from an area where a certain portion of the river is under high simulation detail, to an area under low detail. In this case the simulation would be in a significantly different state if it were operating under a high level of detail. However, for our purposes this is not

a concern. The simulation itself still looks just as detailed, because the underlying fluid simulation does not impact the surface detail (which is done by the texture advection step). The different simulation state only appears as a different series of surface deformations. Obviously this would be inadequate in cases where an accurate fluid simulation is required, but our aim is to produce efficient and visually convincing results, so this does not present a major problem.

The other area where we implement LOD support is directly within the NS solver algorithm itself. During each iteration the linear solver runs a number of times in order to increase its solving accuracy. Our simulation defaults to 20 steps (the same that Stam [47] used in his implementation), and we determined this parameter would be the easiest and most cost-effective method of reducing simulation runtime. As the quadtree depth decreases we also reduce the solver iterations for that quadrant, reducing the number to a low value of 1 at very distant edges of the simulation.

We define the falloff rate for NS solver iterations as a linear correlation between the frustrum's maximum view distance and the viewing position. For example if the maximum view distance is 100 units, and the simulation section being considered for LOD is 50 distance units from the viewing position then its solver iterations will be halfway between maximum and minimum (10 iterations in our case). Any river section that is at a distance of greater than or equal to the maximum viewing distance will be given the minimum number of solver iterations.

Even for off-screen sections of river it is still important to perform the NS advection step. This is because the application of the impulses that drive the river's constant flow are necessary to keep the entire river moving at a constant and stable rate.

5.2.3 Procedural Texture Generation

While it should be possible to add LOD support to the procedural texture generation algorithm, it is not a desirable candidate for LOD. We compute a single high resolution texture and use it across the river, so adding LOD support would mean computing several different textures at different resolutions. Since the highest resolution texture would nearly always be required for the closest river sections the gains in efficiency would be minimal, and in fact could be detrimental in terms of performance (the highest detail level would always

need to be calculated, and adding more detail levels would require additional computations). Perhaps some scheme could be implemented using dynamic procedural patching (i.e., create multiple texture patches with differing resolutions to signify the different LOD levels), however they would need to be matched at the edges so the different patches could not be detected. We did not feel it would be worthwhile to implement such a method, because the texture generation step, while not free, is one of the lower cost elements of the overall algorithm.

5.2.4 Texture Advection

There are a number of inherent problems in switching between detail levels with regards to our use of advection particles, all of which revolve around ensuring that no visual artifacts are introduced into the simulation during LOD transitions. The particles must flow from areas one detail level to another, and when the number of particles change at a LOD transition, the wave fronts being carried by those particles should not immediately fade or blink out of existence.

The most computationally expensive parts of the algorithm are in updating the particles, and advecting the texture based on the particles, both of which are tied to the number of particles in the system, so our efforts in LOD optimization centered around reducing the number of active particles. In order to vary the particle numbers without introducing any visual popping or other artifacts we simply vary the particle birth rate based on the current node's quadtree depth. As mentioned in Chapter 4 we spawn a new advection particle if there is a cell that has no other particles occupying it. In order to accommodate the texture advection LOD, rather than directly querying the simulation grid to determine cell occupation we query the quadtree node, thus effectively combining cells and reducing the particle birth rate by an amount directly proportional to the number of cells contained within the quadtree node. For example, if a node contains 4 cells it will be 4 times less likely have no particles in it, and 4 times less likely to spawn a new particle.

Furthermore, this technique has the added advantage of slowly reducing the number of particles over time. When a section of river changes LOD the excess particles are not immediately killed off, but rather they live out their predetermined lifetime and only after all the existing particles in a node are dead is a new one spawned. This is simpler to implement than other methods which might aim to use different particle sets, or enforce

strict adherence to specific particle counts based on the current LOD level, and we believe the results would be comparable.

5.3 Terrain

Rendering the terrain became a larger issue than initially expected. There were a number of reasons for this, but primarily they were self-imposed constraints. We wish to test the simulation under a variety of conditions that most closely approximated a real world river simulation situation, so we endeavoured to build the terrain engine with the ability to load (and render) various types of datasets commonly used in the Geographic Information Systems (GIS) community, which included several types of Digital Elevation Maps (DEMs). The goal was to study the fluid simulation results and be able to compare them to real rivers so that lessons could be learnt from watching how rivers flow (and are formed) in nature.

Since this thesis concerns river rendering, plausible terrain rendering is not just a simple aesthetic aim. Even small differences in the terrain have relatively dramatic consequences to the resulting river path and fluid surface, because the simulation dynamically builds the flow volume directly from the terrain, and the terrain also continues to affect the simulation even after the river's path is determined. For example, a large rock could be represented as a cube, or as some more complex polygon shape, each of which will produce different results in the fluid surface.

5.3.1 DEM Loading

It is not important for us to discuss the actual parsing and loading of the various DEM formats here as they are all well documented, however we ran into some real-world implementation issues that are worth mentioning. DEM models are produced from various means including satellite imagery, topographical map conversion, radar and sonar data, etc. The freely available DEM maps are of varying quality and resolution and since our aim was to produce a simulator that could easily be tested on a variety of DEMs we needed a method that would allow us to render and simulate any map without much intervention from the user. To this aim we added a filter (bilinear) to the DEM loader that smoothed out incongruities in the maps.

Another important factor in loading real-world data is with regard to computing the

flow path and simulating the flow. There are actually cases in the real world where rivers flow up hill for what we might consider lengthy distances (for more on this see Section 3.3.1). There are also other factors involved in river pathways that are difficult to model in a real-time scenario, such as ground water, underground streams, etc., that can change the path of a river or how it appears on the surface.

Despite these issues, even when comparing our algorithm with real world maps, very clear similarities can be seen between the real river and our simulated river, at both the large and small scale. Comparison screenshots are available in Chapter 6.

5.3.2 Procedural Texturing

Another part of the terrain that became of interest to us was the generation of textured terrain tiles. The terrain texturing that can be seen in the screen shots in Chapter 6 is procedurally generated from a series of input textures that range from images of dirt and gravel to grass, rock, and snow. The algorithm that textures the terrain computes the terrain height range, and then assigns base textures to different height ranges [11, 12, 13]. The idea being that dirt and grass might more commonly cover flat lands, whereas rock and snow would be more likely to cover mountainous regions. The regions are smoothed into and out of each other by varying the opacity of the textures in those locations, random pieces of the textures are transplanted in small spots across the terrain (and smoothed again) in order to supply some more interesting ground features. As is common in real-time graphics, detail textures are also tiled over the map when the viewer is close to the terrain in order to give the texture the appearance of being higher resolution than it is.

5.4 Water Rendering

Although further enhancements could be applied to the rendering of the water, we did implement a few rendering features to make the simulation more visually realistic. Environmental reflections are accomplished via environment cube maps that are generated each frame [10]. The water also has varied levels of transparency based on the depth of the river which was an attempt to simulate particulate or sediment in the river. The strength of this parameter can be varied based on the desired properties of the river. For our simulation we chose to simply make the river bottom begin to fade out at a depth of ten meters, and be

completely dark at a depth of 15 meters. The effect was implemented using volumetric fog, a technique commonly used for clouds and other gaseous phenomena [43].

Much work has already been done in the area, and there are several improvements that could be made which we would like to pursue in the future, including low-pass Fresnel based refractions and reflections, and shadows. For a more detailed discussion on future work with regards to rendering improvements see Chapter 7.1.

Chapter 6

Results

The results we achieved with our method for simulating and rendering rivers in real-time are more visually convincing than existing real-time methods at comparable frame-rates [19, 24, 44, 52]. Minute details in the river flow can be seen as a result of terrain features and other complex interactions between fluid and terrain, and the fluid itself. These complex interactions are a direct result of combining HSP columns with 2D Navier-Stokes. The Texture Advection step allows for a highly detailed fluid surface construction where the water can be seen interacting with the underlying terrain in ways typically reserved for the fully 3D class of fluid solver. Water can be seen speeding up over shallow sections and slowing down over deep sections, as well as becoming turbulent in areas with large underwater obstacles, etc. For examples of this see Figure 6.2.

Also, the waves that are produced from the procedural wave generator can easily be seen flowing with the river, sometimes getting caught in nooks, or eddies, and flow around bends, etc. The addition of the texture advection method sets our technique apart from other real-time river simulations, producing additional detail to the river surface that would not be possible otherwise.

The Level of Detail scheme provides notable performance improvements, and as can be seen by Table 6.1. Even the modest LOD optimizations we have implemented make a significant difference to the frame rate, or polygons we can rasterize per second, in a 960,000 polygon test scene, and suggest that further work with level of detail enhancements could yield additional gains in the area.

Level of Detail	**Frames per Second**	**Poly's per Second**
Fully Disabled	63	61 million
Only Texture Advection	74	71 million
Only Navier-Stokes	111	107 million
Fully Enabled	120	115 million

Table 6.1: Comparison of the simulation running with different amounts of LOD enabled.

There is little difference in static imagery of the same scene with Level of Detail turned on and off (when being viewed from the same angle), as can be seen by Figure 6.3. As previously mentioned, differences can be seen when jumping directly from the current camera location to a distant location in the scene, and the simulation is more accurate with LOD turned off. However, we think the compromise is worthwhile, and are of the opinion that even more LOD work on the algorithm would be beneficial. For more information on our proposed future work with level of detail see Section 7.1.

We feel that the technique could be used, as is, in a game or other interactive application, and that the results would be positive in terms of both visual plausibility and algorithmic efficiency. The success we have achieved with this method indicates to us that further research is warranted, and with more work being done on both the algorithm and the rendering architecture even greater visual results and efficiency could be obtained.

Figure 6.1 looks at the difference between a real river, and a simulated river constructed from DEM data from the same region. In the map view (taken from the Government of Canada's public topographical map data [40]), a northern branch of the river can be seen that is not captured by our river path finding algorithm. The reason for this is that particular section of river flows downward and starts at a height which is greater than the water level which was specified in the scene construction parameters. Since our algorithm does not handle non-planar river surfaces this effect can not be captured without modifying the algorithm.

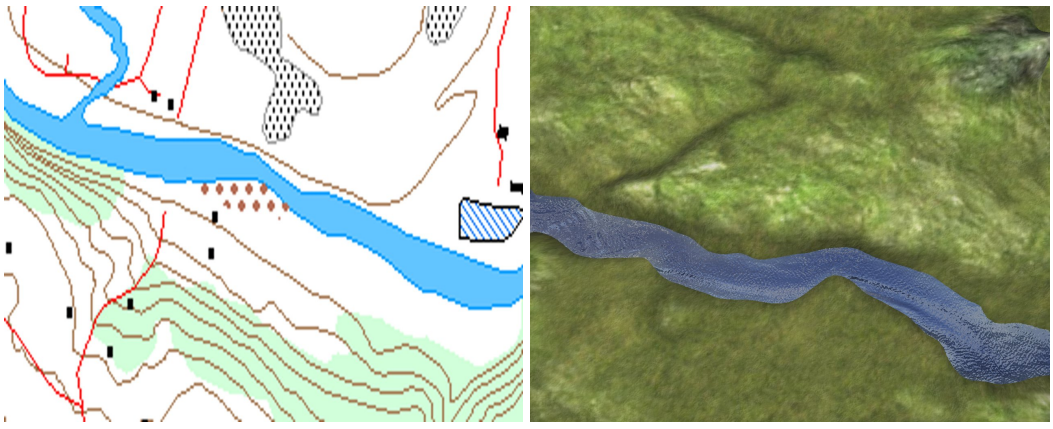

Figure 6.1: Comparison between a map of a real part of Alberta, Canada (left), and the resultant river flow path as calculated by our river path finding algorithm (right). The river offshoot in the top left quadrant is not modelled in our simulation because its surface would not be planar and this is not currently handled by our application.

Figure 6.2 is a comparison between the simulation running with the HSP columns turned on and off. In case with HSP columns turned off the simulation is purely using the 2D Navier-Stokes for simulation results, and any terrain to fluid (and vice versa) interactions are only at the shoreline at the very surface of the fluid. Figures 6.5 through 6.8 are more images of renderings produced by the system.

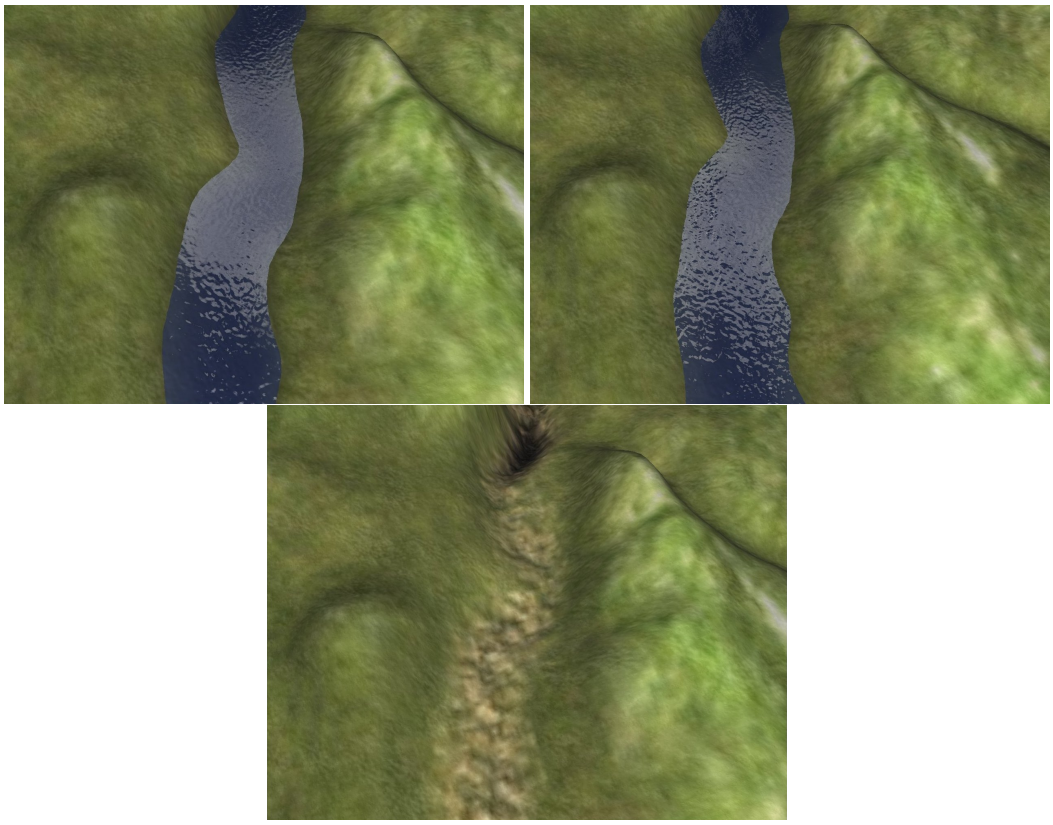

Figure 6.2: Comparison between the simulation running with the Hydrostatic Pressure Columns disabled (top-left), and enabled (top-right). Although difficult to see via still image (non-animated) note the smoother surface in the middle of the screen shot on the left, and rougher surface on the right. These disturbances on the right are caused by a submerged terrain feature under the surface of the water (as seen in the bottom image that has the fluid surface rendering disabled).

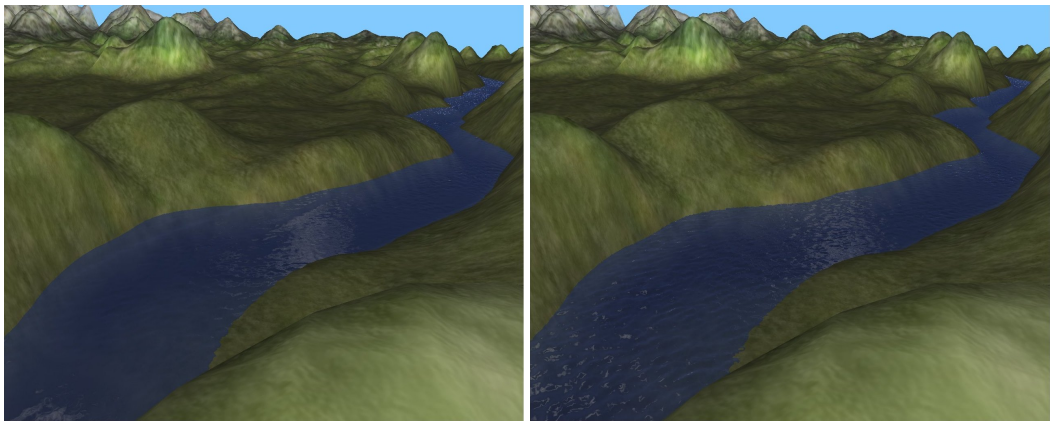

Figure 6.3: Comparison between the simulation running with the Level of Detail disabled (left), and enabled (right). Difference in the images can be seen, but it is subtle, and largely due to the river snapshot being taken at different time in the simulation state. Note that due to the random nature of the algorithm, the results would still be different even between two simulation runs with the exact same settings.

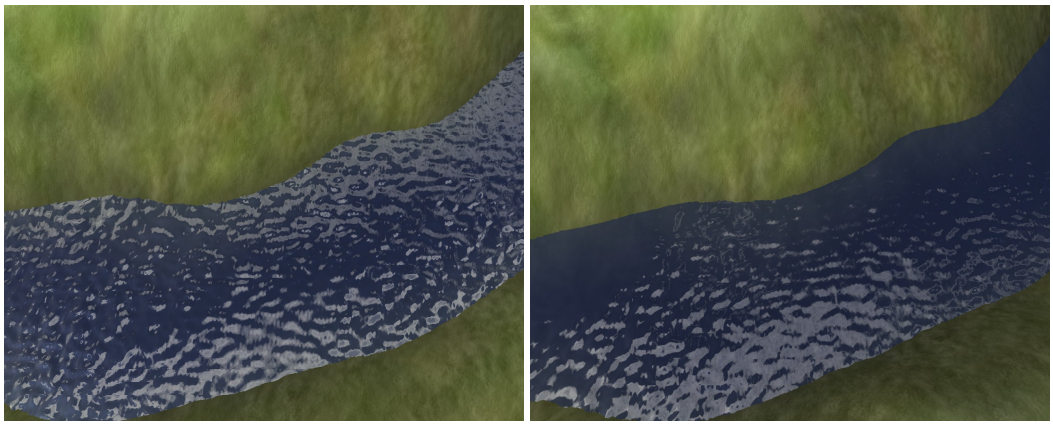

Figure 6.4: The left image shows the system running with Texture Advection using procedurally generated waves, while the image on the right shows the system running with the Texture Advection on using a static advection texture of water. The difference is difficult to perceive through the use of an image alone, however notice the more chaotic ripples on the left and the more uniform ripples on the right.

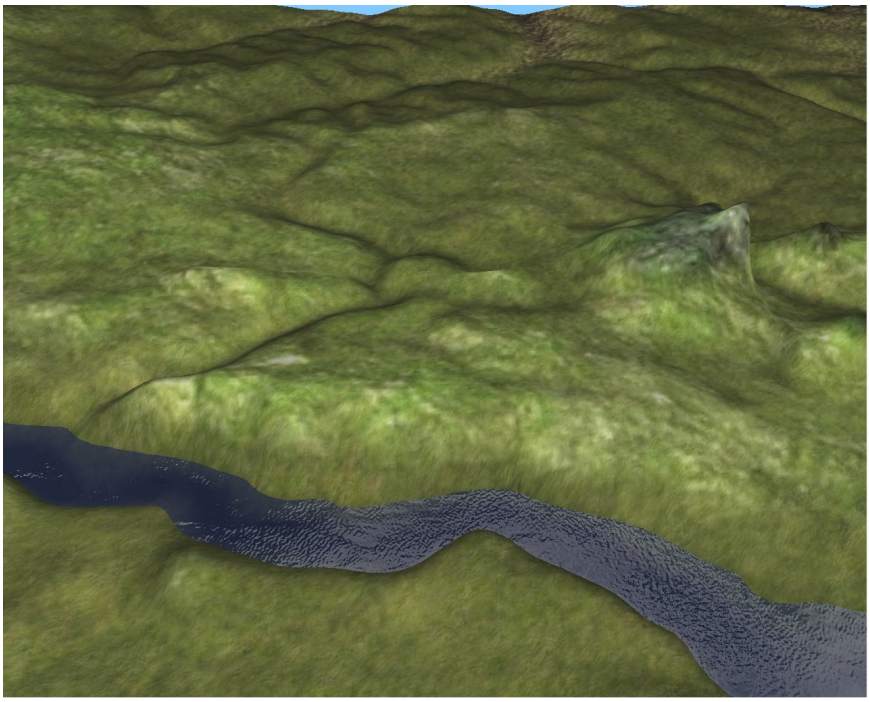

Figure 6.5: Final render showing off all of the pieces of the fluid simulator, procedural terrain, and rendering engine.

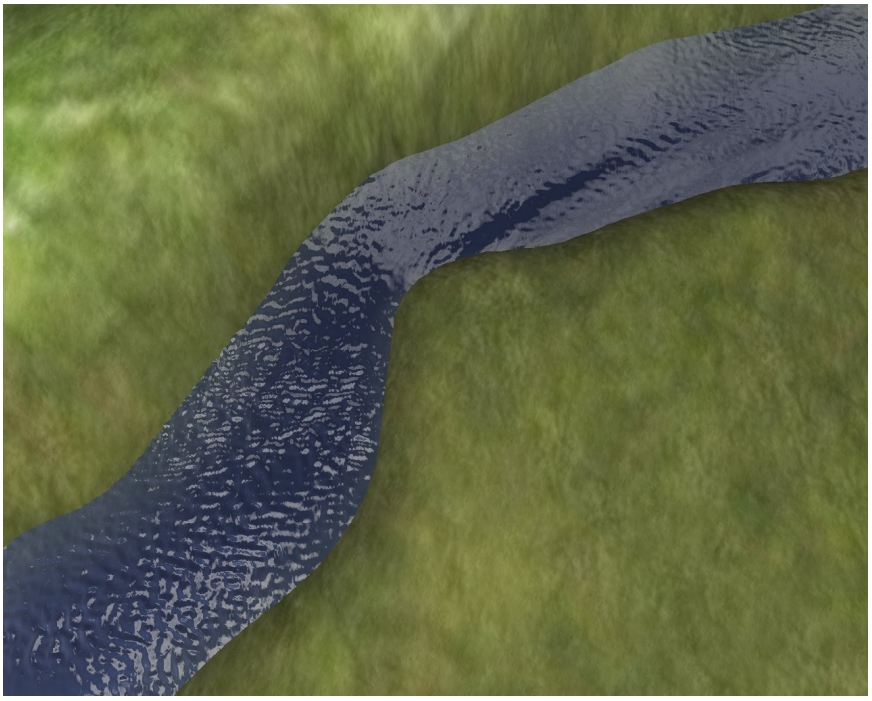

Figure 6.6: Final render showing a close up view of the river surface.

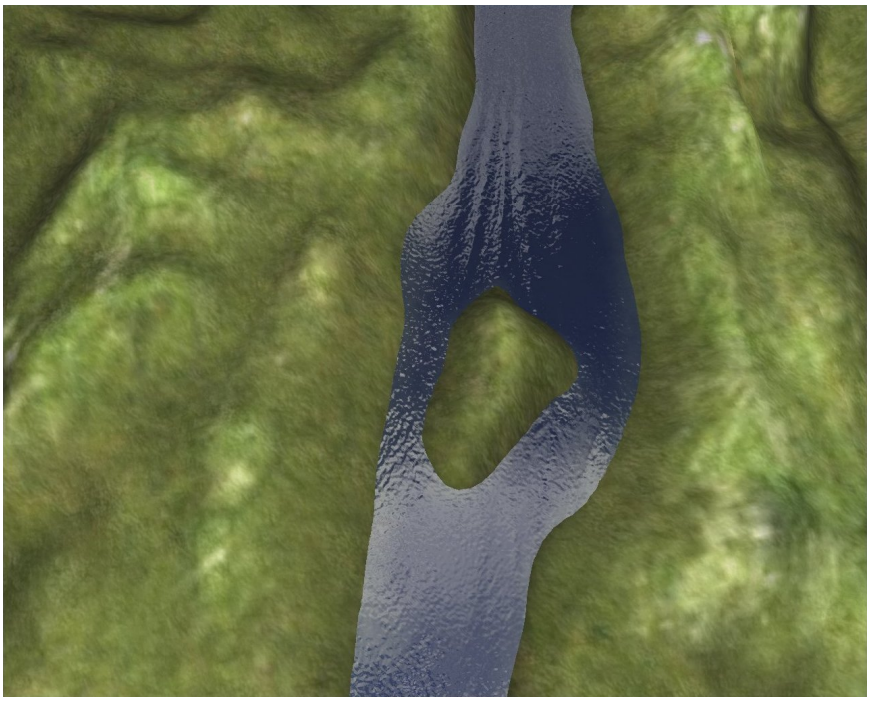

Figure 6.7: Final render showing a river with an obstacle in the middle.

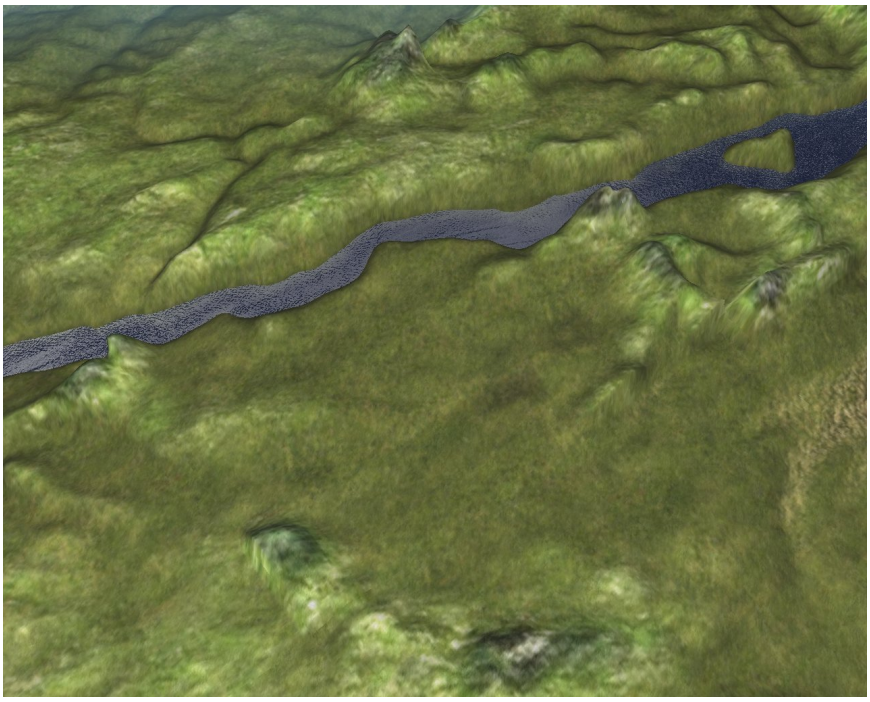

Figure 6.8: Another section of river.

All screenshots were produced on an off-the-shelf dual-core Athlon XP 3800+ computer with an nVidia 8600GT graphics card, and 4GB RAM. The images were captured while the system was running at between 60 and 120 frames per second. None of the code has been parallelized or GPU optimized, and the results should be reproducible on any computer with equivalent or faster hardware.

Chapter 7

Conclusion

7.1 Future Work

As previously discussed, we argue that the technique presented in this thesis for simulating and rendering rivers in real-time is an effective method for visualizing large scale rivers. Despite this there is a lengthy list of items that could be implemented to either improve the existing system, or that could be investigated as interesting further research in the area.

One enhancement to implement would be two-way rigid-body to fluid interactions. We designed the system with this in mind, and built the API from the ground up to allow individual scene objects to have full access to the fluid simulator's velocity and advection information. Rigid-body physics engines are fairly complex and non-trivial to implement by themselves and if one were to be included in our test application it would need to support fluid physics for object buoyancy and other related features which are not currently available in any of the physics engines we had access to at the time of writing. As a result we decided to leave rigid-body physics for future work and focus on the fluid simulation itself.

Another feature we did not have time to experiment with was non-planar fluid surfaces. Other fluid simulators, in the real-time class, generally use some alternate method of representing the non-planar sections (i.e., some use SPH and Marching Cubes for spray, waterfalls, etc.) and this is a possibility with our technique since pressure information is available at every simulation cell. We also considered allowing the surface representation to exist in three dimensions, however the height field based method does not lend itself well to non-planar surfaces (one grid cell can not be over top of another) so any results would be limited by that fact. It would be better to use some other method specifically designed to handle the non-planar aspects.

Another avenue of interesting work could be done in the area of the procedural texture / wave generation methods. We mostly assumed animated waves would be best to advect through the fluid simulation (although we did try random noise as well), but perhaps

there are other better or more interesting functions that could be used as colour advection schemes. Since we have not done enough research to definitively answer that question we cannot say for certain whether or not that is the case, but given our experience and results with this method we feel strongly that procedural waves are the right advection candidate.

There is still plenty of work that could be done in this area. The choice of wave generation algorithm is important, so there could be a suite of algorithms that could be used under different circumstances, that might even be able to be chosen programmatically. Multiple textures with different properties (such as wave frequency) could be combined and advected to produce even more interesting and detailed results. There is also the possibility of tweaking the Tessendorf algorithm parameters in different ways to produce different results. We did experiment with the algorithm settings quite a bit and we believe we produced an optimal setting for our scenes but the parameters are many and the algorithm can be made to produce a wide range of effects.

It would also be desirable to conduct more work with the terrain and fluid rendering engine. The most immediate rendering improvements we would like to make would be Fresnel refraction and reflections. Other improvements would include foam, above and below water foliage, shadows, outdoor lighting (sun glow, god rays, night / dark transitions), skyboxing, clouds, etc.). The list of rendering improvements is lengthy, so our goal was to present the viewers with something that would show the potential of the algorithm if plugged into a mature rendering engine. We also did not want to implement too many rendering enhancements for fear of limiting the detail on the river surface itself.

The other primary area where there are many opportunities for future work is in Level of Detail. We performed only the most basic geometry LOD on the terrain, and no geometric LOD on the river sections (the river sections are split up into LOD patches, however this is only for the purpose of changing detail levels within the fluid simulation algorithm). Terrain pieces are split into sections and the more distant sections are drawn with low detail. Sections that contain fluid or that are close to the user are drawn in full detail. LOD schemes are non-trivial to implement and are still a topic of ongoing research, especially in regard to GPU based schemes like geo-clipmapping [42, 48]. We feel that adding a geometry LOD algorithm into the system would be extremely beneficial, allowing for the simulation and rendering of more and larger sections of river at any given time, and at higher levels of detail.

There is still more work that could be done in terms of simulation LOD as well. We did not run the HSP simulation in parallel with the rest of the simulation, and if that were done it may need parallelizing in order to not become a bottleneck to the simulator. We did not test or perform an experiments to validate this so we are uncertain what the results would be. Our hypothesis is that the HSP columns would become too costly if a large enough section of river were being simulated, so further work in this area would be worthwhile.

The primary factor affecting the computational cost of performing the HSP update is the number of pipes involved in the simulation: the more pipes the higher the cost. Since we already have the quadtree information available, and it already provides a recursive structure that could be used to determine which columns point to which other columns, adding LOD support would be a simple matter of replacing the current pipe set with a new one based on the quadtree structure at each camera movement update. Since all this data is pre-existing and LOD changes would only affect the current quadtree depth, no rebuilding or data structure modification would be required as a result of LOD switches. The algorithm would not necessarily need to be aware that an LOD switch had occurred. For example, at each simulation update the current cell would perform outflow calculations based on the pipes (quadtree node pointers) that are currently enabled via the LOD quadtree structure. We have not performed experiments to prove whether or not this will work, so this is currently a hypothesis. There could be unknown issues that arise from changing pipe sizes constantly, which might lead to simulation instability or visual artifacting.

7.2 Closing

As our results show, we have developed an efficient approach to rendering large scale fluid flows over arbitrary terrains with a relatively high level of physical and visual realism. By combining Impulse Driven 2D Navier-Stokes with multi-tier hydrostatic pressure columns we have created a low computational-cost fluid solver that provides enough 3D information to simulate a river in real-time. We then used a procedural wave generation technique to produce an animated texture which is advected through the fluid simulation results and visualized in order to present users with a highly detailed fluid surface representation that exhibits many of the visual elements that are characteristic of rivers.

Our technique is primarily applicable to real-time and interactive simulation scenarios and has been designed with interfacing between fluid and rigid-body physics objects in

mind. Our implementation does not provide for non-planar fluid surfaces, although the method could be augmented with another fluid simulation technique more suited to this task, as pressure information is available directly from the simulation that could be used to automate creation of waterfalls and large sprays (rapids, etc.), and other natural river features as well as for easy integration with a foam / particulate engine.

The gap that this fluid simulation method fills is by no means complete as a result of this research, but we feel that it is a major step forward in the area of real-time river rendering for interactive applications. The area has not had much prior work done simply because it is a difficult field – large scale physically and visually realistic fluid flows do not easily lend themselves to real-time use.

Our method of combining several existing fluid simulation techniques with our novel approach of driving the Navier-Stokes solver with impulses and using Advection Particles to transport an animated advection texture through the simulation produces results that are characteristic of large rivers. The technique succeeds in increasing the visual detail and realism of what can be produced in real-time, and not only produces the results we hoped to achieve prior to beginning research on the topic, but surpasses what we thought we could achieve in a real-time application. We believe the method is worthy of continued study and hope that our research may influence or inspire others to proceed with their own work in the difficult but rewarding area of real-time river rendering.

Bibliography

[1] Adam W. Bargteil, Tolga G. Goktekin, James F. O'brien, and John A. Strain. A semi-Lagrangian contouring method for fluid simulation. *ACM Transactions on Graphics*, 25(1):19–38, January 2006.

[2] Robert Bridson, Ronald Fedkiw, and Matthias Muller-Fischer. Fluid simulation: Siggraph 2006 course notes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 1–87, New York, NY, USA, 2006. ACM.

[3] Robert Bridson, Jim Houriham, and Marcus Nordenstam. Curl-noise for procedural fluid flow. *ACM Trans. Graph*, 26(3):46, 2007.

[4] Simon Brown and Rhett Collier. Flooding ice age: The meltdown using wavesynth and point based froth. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 21, New York, NY, USA, 2006. ACM.

[5] Jim X. Chen and Niels da Vitoria Lobo. Toward interactive-rate simulation of fluids with moving obstacles using navier-stokes equations. *CVGIP: Graphical Model and Image Processing*, 57(2):107–116, 1995.

[6] Yung-Feng Chiu and Chun-Fa Chang. GPU-based ocean rendering. In *IEEE International Conference on Multimedia and Expo*, pages 2125–2128. IEEE, 2006.

[7] A. J. Chorin and J. E. Marsden. *A mathematical introduction to fluid mechanics*. Springer-Verlag, 3rd edition, 1993.

[8] Simon Clavet, Philippe Beaudoin, and Pierre Poulin y. Particle-based viscoelastic fluid simulation. In Demetri Terzopoulos and Victor Zordan, editors, *ACM SIGGRAPH /Eurographics Symposium on Computer Animation*, pages 219–228, Los Angeles, California, 2005. Eurographics Association.

[9] Hilko Cords. Moving with the flow: Wave particles in flowing liquids. *Journals of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'08)*, 2008.

[10] NVIDIA Corporation. Perfect reflections and specular lighting effects with cube environment mapping. Webpage at `http://developer.nvidia.com/object/Cube_Mapping_Paper.html`, January 2000.

[11] Jürgen Döllner, Konstantin Baumann, and Klaus Hinrichs. Texturing techniques for terrain visualization. In *IEEE Visualization*, pages 227–234, 2000.

[12] Jürgen Döllner, Konstantin Baumman, and Klaus Hinrichs. Texturing techniques for terrain visualization. In Thomas Ertl, Bernd Hamann, and Amitabh Varshney, editors, *Proceedings of the Conference on Visualization 2000 (VIS-00)*, pages 227–234, Piscataway, NJ, October 8–13 2000. IEEE Computer Society.

[13] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994.

[14] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, Annual Conference Series, pages 15–22. ACM Press / ACM SIGGRAPH, 2001.

[15] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[16] R.A. Gingold and J.J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. 181:375–389, 1977.

[17] I. Ginzburg and K. Steiner. Free surface lattice-boltzmann method to model the filling of expanding cavities by bingham fluids. Technical report, Universität Kaiserslautern / Mathematik, January 01 2004.

[18] Simon Green. Particle-based fluid simulations for games. Webpage at `http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_ParticleFluids.pdf`, February 2008.

[19] Nathan Holmberg and Burkhard Wünsche. Efficient modeling and rendering of turbulent water over natural terrain. In Yong Tsui Lee, Stephen N. Spencer, Alan Chalmers, and Seah Hock Soon, editors, *Proceedings of the 2nd International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia 2004, Singapore, June 15-18, 2004*, pages 15–22. ACM, 2004.

[20] Geoffrey Irving, Eran Guendelman, Frank Losasso, and Ronald Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Transactions on Graphics*, 25(3):805–811, July 2006.

[21] Claes Johanson. Real-time water rendering - introducing the projected grid concept. Master's thesis, Lund University, Lund University, Box 117 S-221 00, Lund, Sweden, March 2004.

[22] Roland Sweet John Adams, Paul Swarztrauber. Fishpak. Webpage at `http://http://www.cisl.ucar.edu/softlib/FISHPAK.html`, 1998.

[23] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 49–57, August 1990.

[24] Peter Kipfer and Rüdiger Westermann. Realistic and interactive simulation of rivers. In Carl Gutwin and Stephen Mann, editors, *Proceedings of the Graphics Interface 2006 Conference, June 7-9, 2006, Quebec, Canada*, pages 41–48. Canadian Human-Computer Communications Society, 2006.

[25] T. Klein, M. Eissele, D. Weiskopf, and T. Ertl. Simulation, modelling and rendering of incompressible fluids in real time. In *Workshop on Vision, Modelling, and Visualization VMV '03, 2003.*, Artikel in Tagungsband, pages 365–373. infix, November 2003.

[26] Vivek Kwatra, David Adalsteinsson, Theodore Kim, Nipun Kwatra, Mark Carlson, and Ming C. Lin. Texturing fluids. *IEEE Trans. Vis. Comput. Graph*, 13(5):939–952, 2007.

[27] Anita T. Layton and Michiel van de Panne. A numerically efficient and stable algorithm for animating water waves. In *The Visual Computer*, volume 18(1), pages 41–53. Springer, 2002.

[28] Richard Lee and Carol O'Sullivan. A fast and compact solver for the shallow water equations. *Virtual Reality Interactions and Physical Simulation*, 1(1):51–58, 2007.

[29] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algoritm. *Computer graphics*, 21(4):163–168, July 1987.

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

[31] Frank Losasso, Jerry Talton, Nipun Kwatra, and Ronald Fedkiw. Two-way coupled SPH and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804, July/August 2008.

[32] Leon B. Lucy. A Numerical Approach to Testing the Fission Hypothesis, December 1977.

[33] Marcelo M. Maes, Tadahiro Fujimoto, and Norishige Chiba. Efficient animation of water flow on irregular terrains. In Y. T. Lee, Siti Mariyam Hj. Shamsuddin, Diego Gutierrez, and Norhaida Mohd Suaib, editors, *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia 2006, Kuala Lumpur, Malaysia, November 29 - December 2, 2006*, pages 107–115. ACM, 2006.

[34] G. A. Mastin, P. A. Watterberg, and J. F. Mareda. Fourier synthesis of ocean scenes. *IEEE Computer Graphics and Applications*, 7(3):16–23, March 1987.

[35] Jason L. Mitchell. Real-time synthesis and rendering of ocean water. Technical report, Marlboro, MA, 2004.

[36] David Mould and Yee-Hong Yang. Modeling water for computer graphics. *Computers & Graphics*, 21(6):801–814, November 1997. ISSN 0097-8493.

[37] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In D. Breen and M. Lin, editors, *Eurographics/SIGGRAPH Symposium on Computer Animation*, pages 154–159, San Diego, California, 2003. Eurographics Association.

[38] Simon Schirm Nils Thürey, Matthias Müller-Fischer and Markus Gross. Real-time breaking waves for shallow water simulations. *Proceedings of the Pacific Conference on Computer Graphics and Applications 2007*, page 8, October 2007.

[39] J. F. O'Brien and J. K. Hodgins. Dynamic simulation of splashing fluids. In *Computer Animation '95*, pages 198–205, April 1995.

[40] Government of Canada. The atlas of canada - toporama. September 2008.

[41] Ken Perlin and Eric M. Hoffert. Hypertexture. *Computer Graphics*, 23(3):253–262, July 1989.

[42] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[43] Stefan Roettger. *Volumetric methods for the real time display of natural gaseous phenomena*. Dissertation, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, October 20, 2004.

[44] Zhaoxia Dong Songxin Shi, Xiuzi Ye and Yin Zhang. Real-time simulation of large-scale dynamic river water. *Simulation Modeling Practice and Theory*, 15(6):635–646, 2007.

[45] Jos Stam. Stable fluids. In *SIGGRAPH*, pages 121–128, 1999.

[46] Jos Stam. A simple fluid solver based on the FFT. *Journal of Graphics Tools: JGT*, 6(2):43–52, 2001.

[47] Jos Stam. Real-time fluid dynamics for games. GDC 2003. Webpage at `http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf`, April 05 2003.

[48] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: A virtual mipmap. In *SIGGRAPH*, pages 151–158, 1998.

[49] Jerry Tessendorf. Simulating ocean surfaces. *SIGGRAPH Course Notes 2004*.

[50] Jerry Tessendorf. Simulating ocean water. *SIGGRAPH Course Notes 2001*.

[51] Jerry Tessendorf. Radiative transfer on curved surfaces. *Journal of Mathematical Physics*, 31(4):1010–1019, April 1990.

[52] Sébastien Thon and Djamchid Ghazanfarpour. A semi-physical model of running waters. In *Proceedings of the 2001 International Eurographics Conference*, pages 53–59. ACM, 2001.

[53] Sébastien Thon and Djamchid Ghazanfarpour. Ocean waves synthesis and animation using real world information. *Computers and Graphics*, 26(1):99–108, February 2002.

[54] Patrick Witting. Computational fluid dynamics in a traditional animation environment. In Alyn Rockwood., editor, *Siggraph 1999, Computer Graphics Proceedings*, Annual Conference Series, pages 129–136, Los Angeles, 1999. ACM Siggraph, Addison Wesley Longman.

[55] Cem Yuksel, Donald H. House, and John Keyser. Wave particles. *ACM Transactions on Graphics*, 26(3):99:1–99:12, July 2007.

[56] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Transactions on Graphics*, 24(3):965–972, July 2005.

Appendix A

Source Code Listings

A.1 Apply Flow Hint

```
void applyFlow(float DeltaTime) {
    // make sure the user has specified a flow speed
    if (mFlowSpeed <= 0.0f)
        return;

    // foreach CELL
    for (uint idx = 0; idx < mActiveCellsLength; idx ++) {
        Cell * cell = &mActiveCells[idx];
        Vector2f Velocity = Vector2f(mVelocityX[cell.x][cell.y],
           mVelocityY[cell.x][cell.y]);
        float Length = Velocity.length;

        // do density
        float Density = mDensity[cell.x][cell.y];
        float DensityPrev = mDensityPrev[cell.x][cell.y];
        float DensityFactor = mFlowSpeed;
        float ConstFactor = 2.0f;
        if (DensityPrev < DensityLowValue)
            mDensityPrev[cell.x][cell.y] += DensityAdd * DensityFactor *
                ConstFactor;
        else
            mDensityPrev[cell.x][cell.y] -= DensitySub * DensityFactor *
                ConstFactor;
        if (Density < 0.2f)
            mDensity[cell.x][cell.y] += DensityAdd * DensityFactor *
               ConstFactor;
        else if (Density < 0.3f)
            mDensity[cell.x][cell.y] += DensityAdd * 2.0f *
               DensityFactor * ConstFactor;
```

```
        // do velocity
        Vector2f FlowHint = mFlowHint[cell.x][cell.y];
        float FlowSpeed = mFlowSpeed;
        if (getFlowSpeedPercentile !is null)
            FlowSpeed *= getFlowSpeedPercentile(cell.x, cell.y);
        if (mFlowSpeedRandom)
            FlowSpeed += random(-mFlowSpeedRandom, mFlowSpeedRandom);
        if (Length > FlowSpeed)
            Velocity.scale(VelocityMaxScale);
        else {
            if (Length < VelocityMin)
                Velocity += FlowHint * FlowSpeed;
            Velocity.scale(VelocityMinScale);
        }

        // do hinting
        if (FlowHint.length > 0.0f) {
            float AngleDiff = abs((atan2(Velocity.y, Velocity.x) - atan2
                (FlowHint.y, FlowHint.x)) * rad2deg);
            const float DiffTrigger = 15.0f;
            const float DiffTriggerOverTwo = DiffTrigger / 2.0f;
            if (AngleDiff >= 180.0f - DiffTriggerOverTwo && AngleDiff <=
                 180.0f + DiffTriggerOverTwo) {
                FlowHint.scale(Velocity.length);
                Velocity = FlowHint;
            }
        }

        // set it back
        mVelocityX[cell.x][cell.y] = Velocity.x;
        mVelocityY[cell.x][cell.y] = Velocity.y;
    }
}
```

A.2 Flow Hint Bootstrapping

```
void calculateFlowHinting(Vector2f InitialFlowDirection) {
    // initialize
    clear2DArray(mFlowHint, mGridWidth + 2);

    // apply initial flow hints
    for (uint x = 0; x < LastCol; x ++)
        for (uint y = 0; y < mGridWidth; y ++)
            if (active(x, y))
                mFlowHint[x][y] = InitialFlowDirection;

    // calculate flow hints
    uint Unhinted = mActiveCellsLength;
    uint LastUnhinted;
    bool StuckMode = false;
    int CurPercent = 0;
    do {
        // simulate
        constantMult = true;
        simulate(1.0f / 60.0f);

        // calculate flow hints
        Unhinted = 0;
        for (uint x = 0; x < mGridWidth; x ++) {
            for (uint y = 0; y < mGridWidth; y ++) {
                if (active(x, y)) {
                    Vector2f FlowHint = mFlowHint[x][y];
                    if (FlowHint.length != 0.0f)
                        continue;

                    // no flow hint yet, let's see if we can get one
                    Vector2f Velocity = velocity(x, y) +
                        velocity(x - 1, y    ) +
                        velocity(x + 1, y    ) +
                        velocity(x    , y - 1) +
                        velocity(x    , y + 1);

                    if (Velocity.length == 0.0f) {
```

```
                        if (StuckMode) {
                            mFlowHint[x][y] = Vector2f.normalize(
                                InitialFlowDirection);
                            continue;
                        } else {
                            // nope
                            Unhinted ++;
                            continue;
                        }
                    }

                    // Yes we can grab a flow hint!
                    mFlowHint[x][y] = Vector2f.normalize(Vector2f.
                        normalize(Velocity) + (Vector2f.normalize(
                        InitialFlowDirection) * 1.0f));
                }
            }
        }

        if (Unhinted == LastUnhinted)
            StuckMode = true;
        else
            StuckMode = false;
        LastUnhinted = Unhinted;
    } while (Unhinted);
}
```