Fluid Simulation for Visual Effects

Magnus Wrenninge[1]

April 17, 2003

[1]magwr867@student.liu.se

Abstract

This thesis describes a system for dealing with free surface fluid simulations, and the components needed in order to construct such a system. It builds upon recent research, but in a computer graphics context the amount of available literature is limited and difficult to implement. Because of this, the text aims at providing a solid foundation of the mathematics needed, at explaining in greater detail the steps needed to solve the problem, and lastly at improving some aspects of the animation process as it has been described in earlier works.

The aim of the system itself is to provide visually plausible renditions of animated fluids in three dimensions in a manner that allows it to be usable in a visual effects production context.

The novel features described include a generalized interaction layer providing greater control to artists, a new way of dealing with moving objects that interact with the fluid and a method for adding source and drain capabilities.

Contents

Chapter 1

Overview

Water is familiar to everyone. We know how it moves, how it flows, how it splashes and how it settles. For computer graphics, modeling the behaviour of fluids has long been a challenge. Relying heavily on direct simulation of three-dimensional domains, the computational cost is a big problem which also manifests itself in the difficulty for artists to operate such a system. The number of iterations that can be performed during production is limited and thus the amount of refinement that can go into it.

Recently, new literature has been published describing how to simulate fluids in three dimensions, most notably Stam [17] and Fedkiw et al ([10], [5] and [4]). Although they show how to solve the problem, the prerequisite knowledge is great, and implementing a system based on their papers is still quite difficult.

This thesis starts out by reviewing the mathematics involved. The Navier-Stokes equations are described and equations for conservation of mass are derived. These are non-linear partial differential equations and are difficult (if not impossible) to solve analytically, so numerical methods are required. The finite difference method is described and discrete Poisson equations reviewed, as well as how to treat the different boundary conditions required in modeling fluid flow. Methods for tracking and propagating surfaces is discussed in a section on level set methods. Given the prerequisite mathematics, an overview of Stam's and Fedkiw et al's contributions show methods of solving fluid dynamics problems for gas and liquids.

An overview of a solver for liquids is then given, and is described in somewhat greater detail than in earlier works. An overview of each step required in the simulation is given, and some novel features are introduced. These include a generalized method for handling user interaction with the fluid, driven both kinematically and dynamically. A new way of handling stationary and moving objects and environments are described, which both increases efficiency and simplicitly when implementing the system. Also, a method for implementing regions acting as sources and sinks (areas where fluid is intentionally gained or lost) is shown. Lastly some results are shown and discussed.

Chapter 2

Mathematical overview

2.1 Partial differential equations

A partial differential equation (PDE) is one that involves functions and their partial derivatives. For example, the following is a PDE

$$\frac{\partial x}{\partial t} = 0 \tag{2.1}$$

since it involves partial derivatives of x.

Such equations can sometimes be solved analytically, but the ones we will come across need to be solved numerically, in our case using finite difference methods.

2.2 Fluid mechanics

2.2.1 Equations of motion

The equations of motion for a fluid were derived by Navier and Stokes in the late eighteen-hundreds. Essentially, they are the extension of Newton's laws for momentum, $\vec{F} = m\vec{a}$ as it is applied to a fluid element. In condensed form, the Navier-Stokes equation is as follows

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} + \nu\nabla^2\vec{u} - \frac{\nabla p}{\rho} + \vec{f} \tag{2.2}$$

where $\vec{u}$ is the velocity, ν is the viscosity coefficient, ρ is the density, p is the pressure and $\vec{f}$ is the body force.

The different terms can be refered to as the *advection* term ($(\vec{u} \cdot \nabla)\vec{u}$), the *diffusion* term ($\nu\nabla^2\vec{u}$), the *pressure* term ($\frac{\nabla p}{\rho}$) and the *body force* term ($\vec{f}$).

2.2.2 Conservation of mass

For an incompressible liquid, eq. 2.2 needs to be coupled to an equation stating that no mass can be gained or lost.

If $\vec{F}$ is a vector-valued function with continuous partial derivatives on the region W, then the divergence theorem (Gauss' theorem) states that for W,

bounded by the surface S, with $\vec{n}$ as the outward normal of unit length to S, the following holds

$$\iint_S \vec{F} \cdot \vec{n} \, dS = \iiint_W \nabla \cdot \vec{F} \, dV \tag{2.3}$$

The rate of change of mass in W is

$$\frac{d}{dt} m(W,t) = \frac{d}{dt} \int_W \rho(\vec{x},t) dV = \int_W \frac{\partial \rho}{\partial t}(\vec{x},t) dV \tag{2.4}$$

The rate of change of mass must be equal to the rate at which mass crosses ∂W, i.e.

$$\frac{d}{dt} \int_W \rho \, dV = - \int_{\partial W} \rho \vec{u} \cdot \vec{n} \, dA \tag{2.5}$$

By the divergence theorem, we may rewrite the RHS of eq. 2.5 from a surface integral into a volume integral, simplifying the expression into

$$\int_W \left(\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \vec{u} \right) dV = 0 \tag{2.6}$$

Since this holds for all of W, we have the differential

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \vec{u} = 0 \tag{2.7}$$

For an incompressible fluid we have $\frac{\partial \rho}{\partial t} = 0$, which yields

$$\nabla \cdot \vec{u} = 0 \tag{2.8}$$

2.3 Finite difference method

2.3.1 Discrete gradient

The ∇ operator (*nabla*, or *del*) is defined as the vector of the partial derivatives. On $\mathbf{R}^3$, it has the following form

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \tag{2.9}$$

Given a discrete scalar field $F_{i,j,k}$, we can define the gradient ∇F using eq. 2.9 as

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right) = (F_x, F_y, F_z) \tag{2.10}$$

Which may be discretized using standard central differencing to form

$$\nabla F_{i,j,k} = \left(\frac{F_{i+1,j,k} - F_{i-1,j,k}}{2h}, \frac{F_{i,j+1,k} - F_{i,j-1,k}}{2h}, \frac{F_{i,j,k+1} - F_{i,j,k-1}}{2h} \right) \tag{2.11}$$

2.3.2 Discrete divergence

Given a discrete vector field $\vec{F}_{i,j,k} = (u, v, w)_{i,j,k}$, the divergence is defined as the scalar product

$$\nabla \cdot \vec{F} = \left(\frac{\partial \vec{F}}{\partial x} + \frac{\partial \vec{F}}{\partial y} + \frac{\partial \vec{F}}{\partial z} \right) = \left(\vec{F}_x + \vec{F}_y + \vec{F}_z \right) \tag{2.12}$$

Which can be discretized as

$$\begin{aligned} \nabla \cdot \vec{F}_{i,j,k} &= \frac{(u_{i+1,j,k} - u_{i-1,j,k})}{2h} + \\ & \frac{(v_{i,j+1,k} - v_{i,j-1,k})}{2h} + \\ & \frac{(w_{i,j,k+1} - w_{i,j,k-1})}{2h} \end{aligned}$$

2.3.3 Discrete Laplacian

The Laplace operator, $\nabla \cdot \nabla$ or ∇^2, is defined as the sum of the second partial derivatives. It can be thought of as the divergence of the gradient.

$$\begin{aligned} \nabla^2 = \nabla \cdot \nabla &= \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \cdot \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \\ &= \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \end{aligned}$$

Thus, for a scalar field F, we have

$$\nabla^2 F = \frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} + \frac{\partial^2 F}{\partial z^2} = F_{xx} + F_{yy} + F_{zz} \tag{2.13}$$

The equation can be discretized, forming

$$\begin{aligned} \nabla^2 F_{i,j,k} &= \frac{F_{i+1,j,k} - 2F_{i,j,k} + F_{i-1,j,k}}{(\Delta x)^2} + \\ & \frac{F_{i,j+1,k} - 2F_{i,j,k} + F_{i,j-1,k}}{(\Delta y)^2} + \\ & \frac{F_{i,j,k+1} - 2F_{i,j,k} + F_{i,j,k-1}}{(\Delta z)^2} \end{aligned}$$

2.3.4 Discrete Poisson equations

Poisson's equation has the basic form $\nabla^2 a = x$. On a scalar field F, it has the form $\nabla^2 F = G$. The Poisson equation can also be constructed on a vector field $\vec{F} = (u, v, w)$, such that $\nabla^2 \vec{F} = \vec{G}$, in which case each component is treated separately as $(\nabla^2 u = \vec{G}_1, \nabla^2 v = \vec{G}_2, \nabla^2 w = \vec{G}_3)$. In order to solve the basic Poisson equation on a discrete grid, we use eq. 2.13 to form one (or three, in the case of a vector field) linear system $A\vec{x} = \vec{b}$, where A is the matrix of coefficients from the discrete version of eq. 2.13, $\vec{x}$ is the sought solution and $\vec{b}$ is the RHS of each equation.

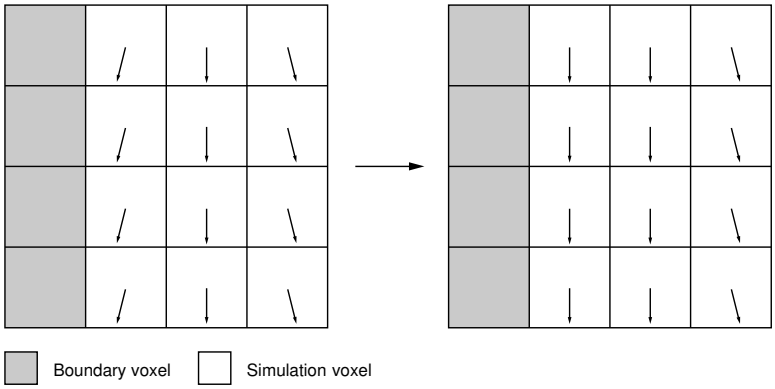

Figure 2.1: The Dirichlet boundary condition $\vec{u} \cdot \hat{n} = 0$ applied to a (2D) vector field

The discrete Poisson equation is simplified greatly if the grid is uniform, e.g. $\Delta x = \Delta y = \Delta z$. We call the grid size $\Delta\tau$ and find the simplification

$$\nabla^2 F_{i,j,k} = \frac{F_{i+1,j,k} + F_{i-1,j,k} + F_{i,j+1,k} + F_{i,j-1,k} + F_{i,j,k+1} + F_{i,j,k-1} - 6F_{i,j,k}}{(\Delta\tau)^2} \tag{2.14}$$

The linear system yielded from this equation is sparse and positive-definite and can be solved efficiently using the conjugate gradient method. Using Gauss-elimination or any other technique that does not take advantage of the sparse nature of the matrix is impossible as the matrix quickly becomes very large. A voxel space that is $10 \times 10 \times 10$ will produce a matrix that is $1,000 \times 1,000$, but for a space that is $100 \times 100 \times 100$, the matrix will be $1,000,000 \times 1,000,000$ ($100^3 \times 100^3$).

2.4 Boundary conditions

The equations in section 2.3.4 hold for all voxels in the grid where neighbouring voxels are also within the solution domain. But once we reach the edge of the voxel grid, or if we allow internal cells to be excluded from the solution, boundary conditions are often needed (depending on the problem modeled).

2.4.1 Dirichlet boundary conditions

When solving a problem on a grid of vectors, the Dirichlet boundary conditions specifies the behaviour of the function along the boundary edge, in our case that the velocity along the normal of a face shared by a simulation and a non-simulation cell must be zero. Intuitively, this means that there must be no flow into a cell that is not regarded as part of the solution domain. Mathematically, it can be stated as

$$\vec{u} \cdot \hat{n} = 0 \tag{2.15}$$

2.4.2 Neumann boundary conditions

Similar to the Dirichlet boundary condition, the Neumann condition deals with flow in or out of the solution domain. Whereas the Dirichlet directly enforces the flow to be zero, the Neumann states that the change of flow along the normal must also be zero. This may seems superfluous given the Dirichlet condition, but it has impacts on the forming of systems of equations as is discussed next. Mathematically, it is defined as

$$\frac{\partial \vec{u}}{\partial \hat{n}} = 0 \tag{2.16}$$

2.4.3 Boundary conditions on discrete Poisson equations

When solving the Poisson equation for a discrete vector field, which is a common operation in our fluid dynamics system, we must take into account the fact that some cells are not part of the simulation domain. This is due to the fact that we have internal obstacles and that the edge of the simulation is treated as a solid wall. Enforcing a Dirichlet condition is easy, we may simply alter each cell to satisfy it. The Neumann condition is slightly more difficult, as it must be incorporated into the Poisson equation itself.

If we look at just the x-dimension of the equation, we have the following

$$\nabla^2 F_{i,j,k} = \frac{F_{i+1,j,k} + F_{i-1,j,k} - 2F_{i,j,k}}{(\Delta\tau)^2} \tag{2.17}$$

If we now regard $F_{i+1,j,k}$ as out of the simulation domain, we want to enforce a Neumann boundary condition on it. Stating that the normal derivative (or the derivative along the normal) is zero implies

$$F_{i+1,j,k} - F_{i,j,k} = 0 \tag{2.18}$$

Plugging this into eq. 2.17 yields

$$\nabla^2 F_{i,j,k} = \frac{F_{i+1,j,k} + F_{i-1,j,k} - 2F_{i,j,k}}{(\Delta\tau)^2} = \frac{F_{i-1,j,k} - F_{i,j,k}}{(\Delta\tau)^2} \tag{2.19}$$

The effect this has on the linear system is that we must remove the row and column corresponding to cell $F_{i+1,j,k}$ and change the coefficient at the central cell ($F_{i,j,k}$) to reflect this change.

Why don't we just rely on the Dirichlet condition? Since it is enforced explicitly after the solution of the Poisson equation, it is a local operation. Using the Neumann condition as we solve the Poisson will make sure that the change in one cell will affect the other cells correctly. If we limit the flow in one cell, the neighbours of it will have to be altered, and so on. The Dirichlet is still needed to provide the correct initial value, as the Neumann only controls the *change* of flow.

2.5 Level sets

Level sets were developed by J. A. Sethian and S. Osher as a way of tracking propagating interfaces. They are based on a shift in philosophy regarding in-

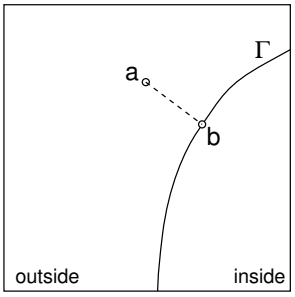

Figure 2.2: Characteristics of a level set. The interface Γ is the zero level set ($\phi = 0$). Furthermore, we have $\phi(b) = 0$ as it is on the surface, and given that b is the closest point on Γ from a we have $\phi(a) = |a - b|$. This implies that the vector $a - b$ is orthogonal to Γ.

terfaces, in that they replace the Lagrangian, geometrical methodology in favor for a Eularian, based on partial differential equations.

Level sets are intuitively similar to iso-surfaces. Given a scalar-valued function ϕ, a level set α is the set of points $\vec{p}$ for which $\phi(\vec{p}) = \alpha$. Thus, a level set can define a surface Γ implicitly through a scalar distance function ϕ. At any point $\vec{p}$ in space, $\phi(\vec{p})$ is the distance to the closest point on the surface Γ.

Implicit surfaces can be used to describe topologically complex surfaces that are hard to parameterize. While parametric surfaces require a basis function which will limit the number of geometries that can be modeled, a level set can describe any given geometry in continuous space, and is only limited by resolution in discrete space. Furthermore, level sets allow for arbitrary deformations and propagations of Γ.

2.5.1 Mathematical properties

The surface Γ is described by the zero level set such that

$$\Gamma(t) = \{\vec{x} | \phi(\vec{x}, t) = 0\}$$

where ϕ is a smooth signed distance function.

$\Gamma(t)$ describes one or many closed surfaces which bound the region Ω. Thus ϕ has the following properties:

$$\begin{aligned} \phi(\vec{x}, t) &< 0 \qquad \text{for } \vec{x} \in \Omega \\ \phi(\vec{x}, t) &> 0 \qquad \text{for } \vec{x} \ni \Omega \\ \phi(\vec{x}, t) &= 0 \qquad \text{for } \vec{x} \in \partial\Omega \end{aligned}$$

The outward unit normal $\vec{N}$ of $\Gamma(t)$ is defined as

$$\vec{N} = \frac{\nabla\phi}{|\nabla\phi|} \tag{2.20}$$

The mean curvature κ is given by

$$\kappa = \nabla \cdot \left(\frac{\nabla\phi}{|\nabla\phi|} \right) \tag{2.21}$$

2.5.2 Propagation of the surface

A level set is propagated by the velocity field $\vec{u}$ according to the partial differential equation

$$\frac{\partial \phi}{\partial t} = -\vec{u} \cdot \nabla \phi \tag{2.22}$$

Since this equation is impossible to solve analytically, it must be discretized. Although simple low-order discretizations could be used, level sets require highly accurate computations to be useful. Thus an upwinded Hamilton-Jacobi ENO (essentially non-oscillatory) or WENO (weighted essentially non-oscillatory), or a TVD (total variation diminishing) Runge-Kutta method is usually employed. [15] provides good details for implementing either of these solvers.

2.5.3 Reinitialization

As numerical errors are introduced during repeated movement of the surface, the value of ϕ may no longer be an accurate measure of the distance to the closest surface. When this occurs, the level set needs to be *reinitialized*. During this, each grid point away from the zero level set is recomputed so that ϕ is once again accurate.

Reinitialization is most efficiently performed using the *fast marching method*. Details can be found in a range of different publications, for example [16].

2.6 Particle level sets

Despite level sets' ability to capture and treat rarefaction and shock (in the case of water, splashes separating themselves from a body of water and the joining of said splash when it comes back onto the surface), some detail may still be lost in this process. In [10], Foster and Fedkiw discuss an improvement to the level set method called the *particle level set method* (PLS method). Whereas the old method relies solely on the old value of ϕ and a velocity field in order to compute a new surface location, the PLS method places marker particles along the zero level set (the surface). By advecting these particles along with the surface, any errors in the level set method become evident as the particles will no longer be located along the $\phi = 0$ interface. In these cases, the values of grid points neighbouring on the particle are corrected by measuring the distance from the particle to the point. Intuitively, if a particle said to be located on the surface is advected and ends up away from the surface, then that surface was not advected properly. Further details are found in [15].

Chapter 3

Related Work

This section describes related work that this thesis builds upon. In the areas of fluid mechanics and computational fluid dynamics, thousands of papers and books have been published. Here, we will only be concerned with those having a direct relation to fluid dynamics in a computer graphics context. Focus will be mainly on visual results as opposed to strict physical accuracy.

The number of publications dealing with fluid dynamics for computer graphics is very limited, with pioneering work taking place only in the last five years. In 1995 Foster and Metaxas [11] published one of the first papers on how to create a system for animating fluids. Stam [17] showed how to solve the Navier-Stokes equation in an unconditionally stable manner. Fedkiw and Foster [10] and Enright, Marschner and Fedkiw [5] built upon both these papers and introduced level sets as a method of tracking a free surface in the simulation.

These next sections describe the work in [17], [10] and [5]. Although [11] is also important, this thesis is not based directly upon it and a detailed review is therefor omitted.

3.1 Stable fluids

In 1999, Stam showed how to solve the Navier-Stokes equations using a semi-lagrangian scheme that guaranteed stable solutions, regardless of the size of the time step [17]. By using the *method of characteristics*, the advection term of eq. 2.2 is solved in a stable manner. Stam also shows how to solve the mass-conservation step using an implicit scheme, as opposed to the explicit one used by Foster and Metaxas in [11].

The following sections outline how Stam's solver works. It should be noted that he only treats one-phase fluids, the fluid occupies the entire simulation domain and no internal object boundaries exist.

3.1.1 Solver overview

To compute the liquid motion, the simulation domain is discretized to form a velocity field $\vec{u}$. Given an initial state $\vec{u}_t = \vec{u}(\vec{x}, t)$, the solution $\vec{u}_{t+\Delta t}$ is solved in four steps. We let $\vec{w}_0 = \vec{u}_t$. The Navier-Stokes equation (eq. 2.2) is then

solved term-by-term as

$$\begin{aligned}
\vec{w}_0(\vec{x}) &\rightarrow \vec{w}_1(\vec{x}) \quad \text{force} \\
\vec{w}_1(\vec{x}) &\rightarrow \vec{w}_2(\vec{x}) \quad \text{advect} \\
\vec{w}_2(\vec{x}) &\rightarrow \vec{w}_3(\vec{x}) \quad \text{diffuse} \\
\vec{w}_3(\vec{x}) &\rightarrow \vec{w}_4(\vec{x}) \quad \text{project}
\end{aligned}$$

Force term

The force term is trivial to solve. We merely integrate the force $\vec{F}$ over the time step Δt.

$$\vec{w}_1(\vec{x}) = \vec{w}_0(\vec{x}) + \Delta t \vec{F}(\vec{x}, t) \tag{3.1}$$

Advection term

The advection term is basically the 'transport of velocity'. A disturbance in the velocity field $\vec{u}$ propagates according to $(\vec{u} \cdot \nabla)\vec{u}$. This is a non-linear term which may be solved in different manners. A finite difference discretization of the ∇ operator can be used, but it imposes limitations on the size of the time steps which can be taken.

Stam proposes an semi-lagrangian technique which had been used in meteorological simulations, but never before in computer graphics. It is based on the *method of characteristics*, a technique that is used to solve partial differential equations (see section 2.1). When applied to the advective derivative $(\vec{u} \cdot \nabla)\vec{u}$, the method may be understood as tracing a particle backwards in the velocity field $\vec{u}$ over the time step Δt, and finding the velocity vector at that point. We define the tracing function $\vec{p}(\vec{x}, dt)$ as the position that a particle currently at $\vec{x}$ had at time $t + dt$. In this way, we can solve the advection term as

$$\vec{w}_2(\vec{x}) = \vec{w}_1(\vec{p}(\vec{x}, -\Delta t)) \tag{3.2}$$

This method has the advantages of being easy to implement (only a particle tracer and a vector field interpolator is needed), as well as being unconditionally stable.

Diffusion term

The diffusion of the fluid is due to viscosity. A grid cell will be pulled along with the velocities of its neighbours, with the kinematic viscosity constant ν determining the scale of the effect.

This diffusion has the form of a standard diffusion equation.

$$\frac{\partial \vec{w}_2}{\partial t} = \nu \nabla^2 \vec{w}_2 \tag{3.3}$$

In [11], Foster and Metaxas used a finite difference scheme and explicitly solve this equation for every grid cell. While this is easy to implement, it has problems with stability as the viscosity becomes large. Instead, Stam solves the term implicitly by forming

$$\frac{\vec{w}_3 - \vec{w}_2}{\Delta t} = \nu \nabla^2 \vec{w}_3 \tag{3.4}$$

$$\vec{w}_3 - \nu\Delta t\nabla^2\vec{w}_3 = \vec{w}_2 \tag{3.5}$$
$$\left(\mathbf{I} - \nu\Delta t\nabla^2\right)\vec{w}_3 = \vec{w}_2 \tag{3.6}$$

After discretizing the ∇^2 operator, this equation leads to a sparse linear system which can be solved very efficiently using the *conjugate gradient method*. The underlying mathematics are described in section 2.3.4.

Pressure term

In order to solve the pressure term, Stam uses a projection method. The *Helmholtz-Hodge theorem* states that any vector field $\vec{w}$ may be decomposed into a divergence-free field and a gradient (conservative) field.

$$\vec{w} = \vec{u} + \nabla q \quad \text{where} \quad \nabla \cdot \vec{u} = 0 \tag{3.7}$$

We define the projection function $\vec{P}$ as the operator that projects a field onto its divergence-free part, thus

$$\vec{u} = \vec{P}(\vec{w}) = \vec{w} - \nabla q \tag{3.8}$$

This may be applied to the basic Navier-Stokes equation (2.2) to yield

$$\frac{\partial\vec{u}}{\partial t} = \vec{P}(-(\vec{u}\cdot\nabla)\vec{u} + \nu\nabla^2\vec{u} + \vec{f}) \tag{3.9}$$

We note that $\vec{P}(\vec{u}) = \vec{u}$ since we are assuming that fields be divergence-free, and that $\vec{P}(\nabla p) = 0$ since ∇p is a conservative field.

In order to make the field $\vec{w}_3$ divergence free, we need to find the scalar field q of eq. 3.8. By multiplying both sides of the equation by ∇ we have

$$\nabla \cdot \vec{w}_3 = \nabla^2 q \tag{3.10}$$

After discretizing the ∇^2 operator, a sparse linear system similar to that of the diffusion step is found and can be solved in the same manner. To find the resulting field $\vec{w}_4$, we simply take

$$\vec{w}_4 = \vec{w}_3 - \nabla q \tag{3.11}$$

and compute the resulting field using a finite difference discretization of the gradient operator.

Section 2.4.3 discusses how to solve this linear system and how to treat cells lying at the edge of the voxel grid.

3.2 Fluids on irregular domains

While Stam showed how to solve the Navier-Stokes equations for a gas, in [10], [5] and [4] Fedkiw et al showed how to compute solutions for fluids with surfaces and how to accurately track the surface between air and liquid. The papers built upon the earlier work by Stam and Foster/Metaxas for simulating the fluid motion, and on work by Sethian and Osher for surface-capturing. The most important differences are described below.

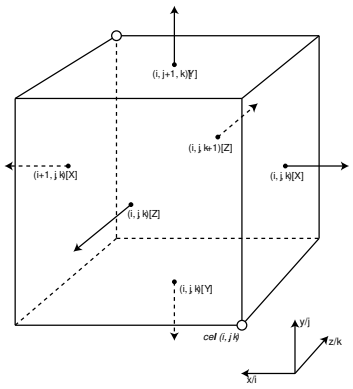

Figure 3.1: MAC grid conventions.

3.2.1 Velocity field

Where Stam used a regular grid, i. e. all pressures, densities and velocities are defined at the center of each voxel, the velocities are instead defined at the center of each face of a voxel. This is called a 'staggered' or MAC grid. Figure 3.1 illustrates the principle.

3.2.2 Surface representation

Although water simulations are similar to gas simulations, some additional steps must be taken. The surface must be tracked, and the domain must be altered as the surface captures a different region in each time step. In [10], the method of using level sets to track the fluid surface is introduced. Level sets fulfill both the need for a robust treatment of moving fronts as well as a way of determining which voxels are part of the solution domain.

But since level sets have a tendency of gaining and/or losing mass after repeated propagation, an improvement on level sets called *particle level sets* (see section 2.6) is introduced. By placing large numbers of marker particles along the surface and advecting these using the same velocity field as for the level set, any numerical errors can be corrected by examining the value of a distance function at each grid point.

In [10], marker particles are placed on the inside of the surface, which prevents the level set from falling back on itself, but still allows the level set to gain mass. This method is refined in [5] by placing marker particles on both sides of the surface. This makes the solution symmetric, but requires careful treatment where particles on the outside of the surface move to the inside while at the same time particles on the inside move to the outside.

3.2.3 Boundary

All stationary objects are converted to a voxel representation where each voxel is regarded either as in or out of the boundary. Moving objects are kept as polygons, which requires a search through all polygons in order to determine if

a voxel is within a moving boundary. Also, this limits the number of polygons that may lie within a single voxel to just one.

3.2.4 Updating the velocity field

The force and advection term of the Navier-Stokes equation is solved exactly as described in Stam's solver above. Since only water-like fluids are regarded, the viscosity term is solved explicitly using finite differences. This is acceptable, as instabilities only occur for large viscosities. Last, the velocity field is made to conserve mass (e. g. be divergence free) along the same lines as Stam, but since a staggered grid is used the result is more accurate.

In all above steps, care must be taken to only apply forces and enforce mass-conservation on those voxels that are actually inside the fluid. The sign of ϕ of the surface level set provides a simple method of checking this.

Chapter 4

System overview

This section describes the methods and techniques used in the system. The problem we are solving is the motion of an interface Γ under the influence of a velocity field $\vec{U}$ on a domain D.

4.1 Novel Features

While the papers reviewed in section 3 are the foundation of the system described here, we will also introduce some new features.

4.1.1 Generalized representation of user forces and velocities

Previous papers have discussed ways of letting the user influence the motion of the water, but none of them propose a general method of integrating interaction elements into the solver.

In [10], Foster and Fedkiw propose using sets of 3D splines to control the motion of the fluid and describe a method of setting the values in the simulation velocity field to accomodate the inputs. In our system, interaction elements can be of any kind imaginable, as long as it can be 'rendered' into a 3D vector field. Thus for 3D splines, we would set the values of voxels neighbouring the spline along the lines of the method in [10].

By creating a layer of abstraction to the interaction inputs we can allow the user to write custom interaction elements, as long as they can be represented as a vector field.

4.1.2 Sources and drains

Sources and drains are places in the simulation where water is intentionally gained or lost. A source is a group of voxels that are always populated by water, a drain one that is always populated by air. Both are effective methods of giving the user greater control as well as reducing simulation time. Instead of using a large tank of water with a hole for the water to exit through in order to inject water into the a simulation, we can choose to model only the hole where water is exiting. Correspondingly, instead of having water exit through a hole

in the floor and having to simulate the container it spills into, we can choose to model only the drain hole.

4.1.3 Level set representation of obstacles

In [10], Foster and Fedkiw describe a method of dealing both with stationary and moving obstacles. Stationary ones use a voxel representation and moving ones are made up of polygons. While it is fine to represent stationary obstacles as voxels, the polygon approach to moving obstacles require that a search be executed every time one wants to find the polygon intersecting a given voxel. Inside/outside tests may also be difficult.

By creating a level set for the static obstacles and a level set per frame for moving obstacles, we can use CSG (Constructive Solid Geometry) operators to merge the two. If we associate a velocity field with the obstacles, we can easily decide whether or not a voxel is stationary or moving and treat it accordingly.

Having a volume representation of the moving obstacles allows for constant-time queries of whether a given voxel is inside or outside a moving obstacle, and what its velocity is.

4.2 Data structures

A number of data structures are required for the solver to work. Here we will describe the functionality each one must offer as well as some details of their implementations.

4.2.1 3d fields

In order to store data in a volume, a 3d field is used. The field is discrete, with regularly spaced sample points (i. e. a regular grid). In order to access an element, we have $F_{i,j,k} = value$, where $i \in [0, width)$, $j \in [0, height)$, $k \in [0, depth)$. The bounding box for the field is defined by the points at $F_{0,0,0}$ and $F_{width-1,height-1,depth-1}$.

The values stored in a 3d field must all be of the same datatype, but different fields may have different datatypes. The C++ *template* construct is used to accommodate this. Some classes that inherit from the 3d field template introduce new method calls that apply only to that class. They are described below.

Scalar fields

Scalar fields allow the storage of any scalar data, such as integers and floating point values. They are used in the simulator for storing density values (floating point) as well as for book keeping (integers). Appropriate mathematical functions are available, such as the gradient operator (∇).

Vector fields

Vector fields are mainly used to store velocity data, and uses any 3d vector as its datatype. The divergence operator ($\nabla \cdot$) is available.

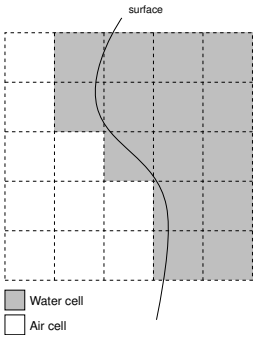

Figure 4.1: Classifying water cells versus air cells

Level sets

A level set is a sub-class of scalar field, since the data stored is principally equivalent. The methods available are different however, and include reinitializing (see section 2.5.3) and moving the level set (see section 2.5.2) as well as performing CSG operators.

Particle level sets

Particle level sets subclass level sets but add storage of particles. Since particles will be created and deleted continuously, but no random access is required, a double-linked list is used.

4.3 Variables

The following variables need to be kept

Surface The surface is represented using a particle level set. All voxels that have $\phi \leq 0.5\Delta\tau$ (computed at the center of the voxel) are regarded as being in water and thus in the solution domain. Figure 4.1 shows the principle.

Velocity field The velocity field is a vector field which will keep track of the flow throughout the simulation space.

Boundaries The boundaries define where water cannot enter. It is represented using a normal level set, since we will never be using level set operations to move it. Instead, if it is animated, a sequence of pre-calculated level sets are read in. Voxels with a $\phi \leq 0$ are regarded as inside an obstacle – see figure 4.2

Boundary velocity field In order to allow for the boundary to be animated, we keep a separate velocity field with values for each boundary voxel. Stationary objects are treated no different from moving ones, so there is no need to use a separate field for specifying which voxels move.

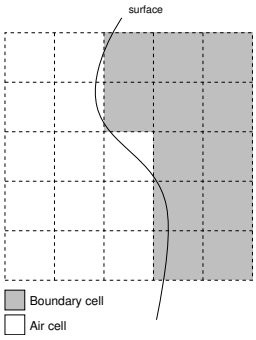

Figure 4.2: Classifying boundary cells versus air cells

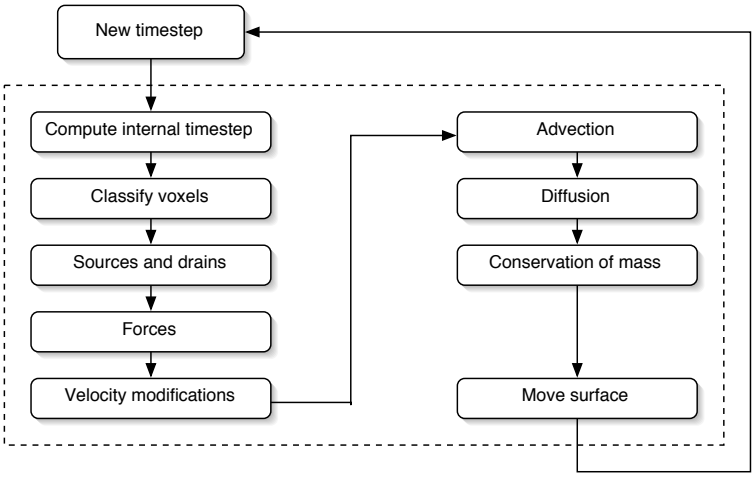

Figure 4.3: Overview of the simulation

4.4 Computing the solution

This section describes the steps taken to solve the equations of fluid motion for a single time step. Figure 4.3 shows an outline of each time step.

4.4.1 Time step

The time step of the simulation is controlled by the user, most often set to $\frac{1}{24}$ seconds, which is the frame rate of film. However, this time step may be large enough to make the simulator inaccurate. In order to allow the user to specify an arbitrarily large time step while still being in control of the simulation, the time step is subdivided into smaller ones internally.

The CFL condition specifies that the time step must be chosen so that the maximum velocity on the simulation domain will move a massless particle at most one voxel. While our simulator is stable for arbitrarily large time steps, its accuracy will be reduced as the velocities grow bigger. The user may therefor

alter a variable called *CFLmultiplier* to control the CFL condition. From this multiplier, the number of necessary subdivisions needed can be computed as

$$subdivisions = C\frac{\vec{u}_{max} \cdot dt}{\Delta\tau}$$

Where $\vec{u}_{max}$ is the maximum velocity in the velocity field $\vec{u}$, dt is the user-specified time step, C is the scaling factor (*CFLmultiplier*) for controlling the fidelity of the simulation (lower is less accurate), and $\Delta\tau$ is the size of one voxel.

4.4.2 Book keeping

Throughout the following steps we will need to access parts of the velocity field based on their state (in water, in air, in obstacle, in source etc.). Also, we need to be able to quickly traverse any set of voxels.

Voxel state

By traversing the entire level set field once we build a buffer of the state for each voxel. This is represented with flags for each state, since a voxel may both be in water and in a source, both in air and in a drain. The available states are:

- Air, the voxel being completely filled with air.
- Water, the voxel being completely or partially filled with water.
- Obstacle, the voxel being at least half filled with an obstacle.
- Source, the voxel being completely within a source.
- Drain, the voxel being completely within a drain.

Traversal lists

In order to quickly traverse all voxels of a given state (e. g. all water voxels) the coordinates for each voxel is added to a list during the creation of the voxel state (see above). This ensures that we only visit those voxels that are relevant during each step.

4.4.3 Sources and drains

In order to efficiently be able to inject and remove fluid from the simulation, sources and drains are available. Sources as well as drains are represented by ordinary level sets. For the source Sc and the simulation domain D such that $Sc \in D$, we have

$$\phi(\vec{x}, t) \; < \; 0 \tag{4.1}$$

for all $\vec{x} \in Sc$ and all t. For a drain $Dr \in D$ we have

$$\phi(\vec{x}, t) \; > \; 0 \tag{4.2}$$

for all $\vec{x} \in Dr$ and all t.

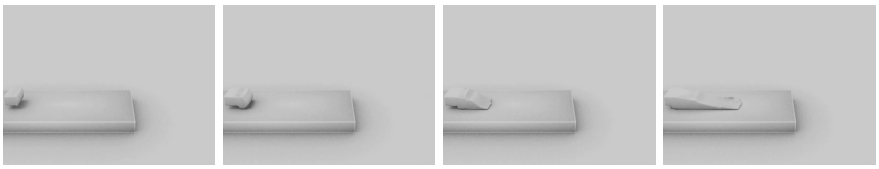

Figure 4.4: A source at the left injects fluid into the simulation domain

A velocity field $\vec{V}$ (which may be animated) is used to provide initial velocities at each frame. For all voxels within a source or drain, we first set the value of ϕ so that eq. 4.1 and 4.2 are satisfied. This is achieved by performing a CSG union operator on sources and a CSG difference operator on drains.

Then, for any voxel within a source or drain, we set the velocity to be that of the supplied source/drain velocity. Thus for the simulation velocity field $\vec{U}$, we have

$$\vec{U}(\vec{x}) = \vec{V}(\vec{x}) \tag{4.3}$$

for all $\vec{x} \in Sc$. Figure 4.4.3 shows an active source.

4.4.4 Forces

Forces are used to drive the simulation dynamically (as opposed to kinematically, see section 4.4.5). In order to add the force term to the simulation velocity field $\vec{U}$ we compute the following for each cell

$$\vec{U}(\vec{x}, t + \Delta t) = \vec{U}(\vec{x}, t) + \vec{F}(\vec{x}, t)\Delta t \tag{4.4}$$

where $\vec{F}$ is the force field composed of gravity plus any user-specified forces.

4.4.5 Velocity modifications

Velocity modifications are used to drive the simulation kinematically. The user specifies the velocity at each point in the field as well as a blend factor field. The blend factor is used to facilitate ease-in/ease-out, where the velocities are modified gradually over time.

Using a blend factor to control velocities may seem similar to using a force, but there is a difference in that a velocity modifier will never reach a value higher than that specified in the velocity field.

Computing the modified velocity of the simulation velocity field is done at each cell through

$$\vec{U}(\vec{x}, t) = (1 - \alpha(\vec{x}, t))\vec{U}(\vec{x}, t) + \alpha(\vec{x}, t)\vec{V}(\vec{x}, t) \tag{4.5}$$

where $\vec{U}$ is the original velocity field, $\vec{V}$ is the velocity modification field and α is the blend factor at the given cell. Figure 4.5 shows the effect of the blend factor.

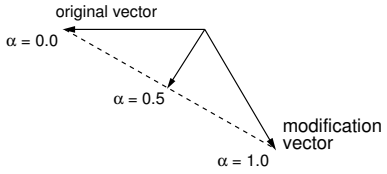

Figure 4.5: The effect of α when modifying velocities

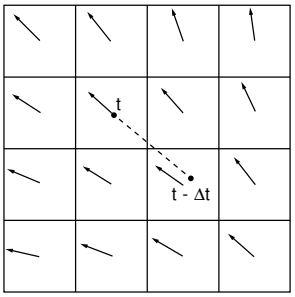

Figure 4.6: A particle at time t is backtraced through a velocity field. The result is interpolated from the four neighboring vectors closest to the point at time $t - \Delta t$

4.4.6 Advection

The advective term of eq. 2.2 is solved exactly along the lines of Stam (see [17] and section 3.1).

For every cell in the simulation, we are trying to find the advective derivative of the velocity field $\vec{U}_{i,j,k}$. By using a particle tracer, we march backwards in time Δt time units along the flow. The new value for $\vec{U}_{i,j,k}$ is the velocity found at the location of the back-traced particle. The particle-tracing is implemented as an Euler- or Runge Kutta-type differential equation solver in three dimensions. Figure 4.6 shows an Euler particle tracer.

4.4.7 Diffusion

The diffusion term is solved along the lines of Stam [17]. The linear system is constructed as described in 2.3.4 and is solved using a pre-conditioned conjugate method solver. If the viscosity is small (as it is when modelling water), no boundary conditions are needed except for setting the velocity equal to zero for any voxels inside obstacles or outside the solution domain. For high-viscosity liquids and liquids where the viscosity varies on the simulation domain, special methods are required. Henrik Fält describes a system for dealing with these cases in [7].

Once the linear system is solved, the vector b contains the new velocity for each voxel.

4.4.8 Conserving mass

In order to keep the the velocity field mass-conserving (i.e. divergence free), we need to compute the pressure for each cell such that when removing the pressure term from eq. 2.2, a divergence free field is obtained. The method for doing this is described well in [10] and the underlying mathematics in sections 2.3.4 and 2.4.3. In essence, a linear system is constructed where each row is the following equation

$$\sum_{n=i,j,k} (p_{n+1} + p_{n-1}) - 6p = 2\rho\frac{\Delta\tau}{\Delta t} \sum_{n=i,j,k} (u_{n+1} - u_{n-1}) \tag{4.6}$$

The sigma notation is simply a shorthand for doing differencing in three dimensions. The LHS contains only the unknown variable p and this is where the boundary conditions are applied. In constructing the A matrix (the coefficients of the p terms each have a corresponding column in the A matrix) for the Poisson equation two types of boundary conditions are required. The first is the liquid-obstacle boundary, which is described in section 2.4.3. These cases do not affect the RHS of the equation, since the pressure of the an obstacle cell is zero and this only cancels a term on the LHS.

The second boundary condition is the liquid-surface boundary. Thanks to the *ghost fluid method* (see [8]), this boundary condition is modelled by explicitly setting the known pressure values for those cells. Thus, any neighbour cell that is defined as being in air is removed from the A matrix and the atmospherical pressure is subtracted from the RHS of 4.6.

In effect, all voxels that do not require a special boundary condition will have the center coefficient of the A matrix equal to -6 and six other coefficients in the row equal to 1. All voxels that have boundary conditions will have a center coeffient equal to minus the number of neighbour cells defined as 'in liquid', and the corresponding columns having a coefficient of 1. Any cells being inside obstacles or air are removed altogether by deleting the corresponding row and column of the A matrix.

When computing the divergence present in each cell (the second sum of eq. 4.6) in order to produce the RHS for the Poisson equation, it is important to take into account any moving objects. A neighbour voxel that is inside a boundary should not be treated simply as having zero velocity, instead it is necessary to look up its true velocity in the boundary velocity field (see section 4.3).

Once the linear system is solved the new velocity for each cell is computed from

$$\vec{U}_{i,j,k}^{t+\Delta t} = \vec{U}_{i,j,k}^{t} - \frac{\Delta t}{2\rho\Delta\tau} \left((p_{i+1} - p_{i-1}), (p_{j+1} - p_{j-1}), (p_{k+1} - p_{k-1})\right) \tag{4.7}$$

which is essentially the equation $u = w - \nabla p$ (see 3.1.1).

4.4.9 Moving the surface

Since our surface is represented by a particle level set, there are two steps that need to be performed. First, as described in section 2.5.2, the surface Γ is moved by solving the following equation for all grid points

$$\frac{\partial \phi}{\partial t} = -\vec{U} \cdot \nabla \phi \tag{4.8}$$

After this step, each particle in the particle level set is moved using Euler or Runge Kutta integration along the flow lines of the velocity field. The particles are then used to correct the value of ϕ along the surface (see section 2.6).

4.5 Output

The output of the simulator is two three-dimensional fields: the level set surface Γ and the velocity field $\vec{U}$. The animation is a frame-by-frame sequence of these fields.

Although only the level set surface is needed in order to render the animation, the velocity field offers allows easy motion-blurring since it contains the velocity at each point on the surface. Also, it can provide particles with initial velocities, for example when accentuating splashes.

Chapter 5

Results

5.1 Example 1

The example in figure 5.1 shows a round ball having been plunged into a pool of water. The first and subsequent frames show the effect on the water after the ball has come to rest. It displays the following components in action

Moving objects The moving ball is decribed by one level set for each animation frame. The level set need only capture the region the ball is actually occupying, not the entire simulation space. If more objects were involved in the animation, separate level set files would be created. The boundary velocity field is calculated at the center of each voxel and the vector field is recorded for each frame. All these computations are separate from the simulation engine.

Surface tracking After impact, a thin sheet of water develops. If ordinary level sets where to be used, the sheet would most likely disappear from numerical errors in the level set propagation equation. However, the particle level set used accurately maintains a coherent surface.

The simulation time for the $140 \times 91 \times 111$ domain was approximately 10 minutes per frame.

5.2 Example 2

In figure 5.2 a $20 \times 10 \times 20$ voxel region at the left end of the experiment is defined as a source. The simulation is 10 m long and the fluid is injected at 3 m/s. The images show the fluid hitting a wall and accurately exhibiting a wave crashing back on itself. Several secondary waves are visible in the third and fourth frames.

This behaviour is entirely caused by the Navier-Stokes equations with boundary conditions, no special cases are regarded in order to treat falling droplets, waves building, water splashing or the rising of the water surface.

The simulation time for the $200 \times 80 \times 31$ domain was approximately 2 minutes per frame.

Figure 5.1: A sphere is moved rapidly into a pool of water. The thin splash is only possible to capture using a particle level set – ordinary level sets would fail to keep track of the fine detail.

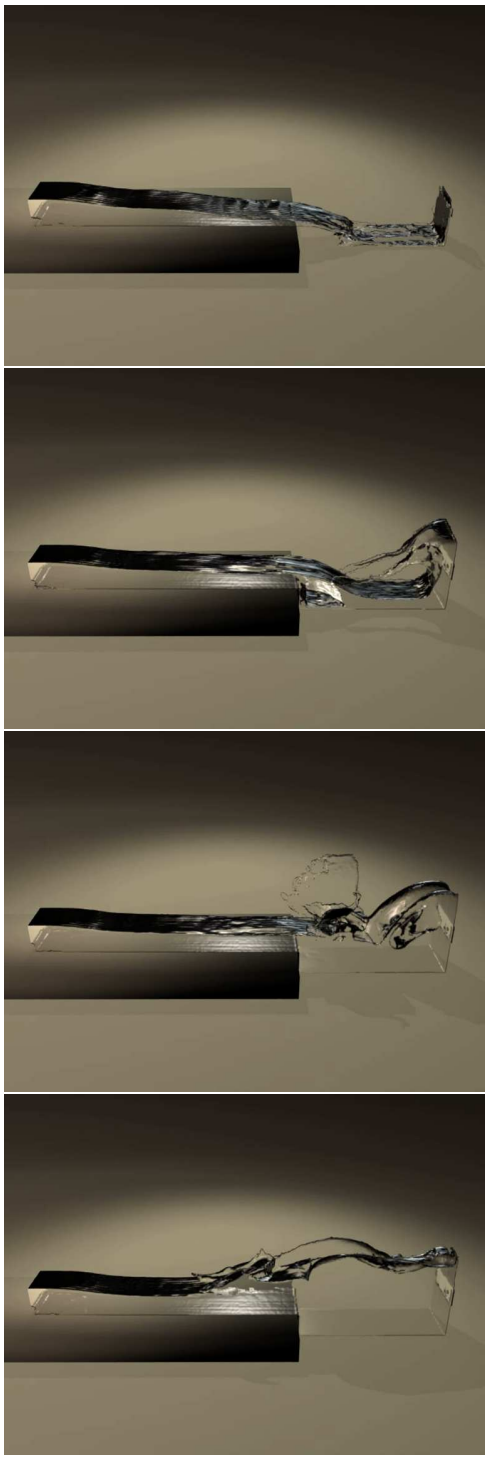

Figure 5.2: A source at the left edge of the experiment injects water at a high speed. After it reflects off of the opposing wall, the appropriate crashing wave is formed.

5.3 Example 3

The images in figure 5.3 show a moving obstacle which contains a pool of water (the example in 5.1 is a solid ball which is only *driving* the water in front of it). As the container is rotated, the water pours out of it and into the larger pool of water underneath. We note that throughout the rotation no water leaks into the container. This movement is modelled by the user in exactly the same way as in 5.1 but can be used to model completely different behaviours.

5.4 Example 4

The two images in figure 5.4 are meant to illustrate the behaviour of the simulator as the resolution changes. Since units are treated correctly throughout, the resolution may be altered without changing the rate at which waves propagate, the speed at which a source ejects liquid, etc. However, as the images show, a high resolution is necessary not necessarily to capture the correct motion of the water, but to provide the fine detail needed to convince a viewer that what is seen is really similar to water.

Figure 5.3: A moving container filled with liquid is emptied into a still pool of water.

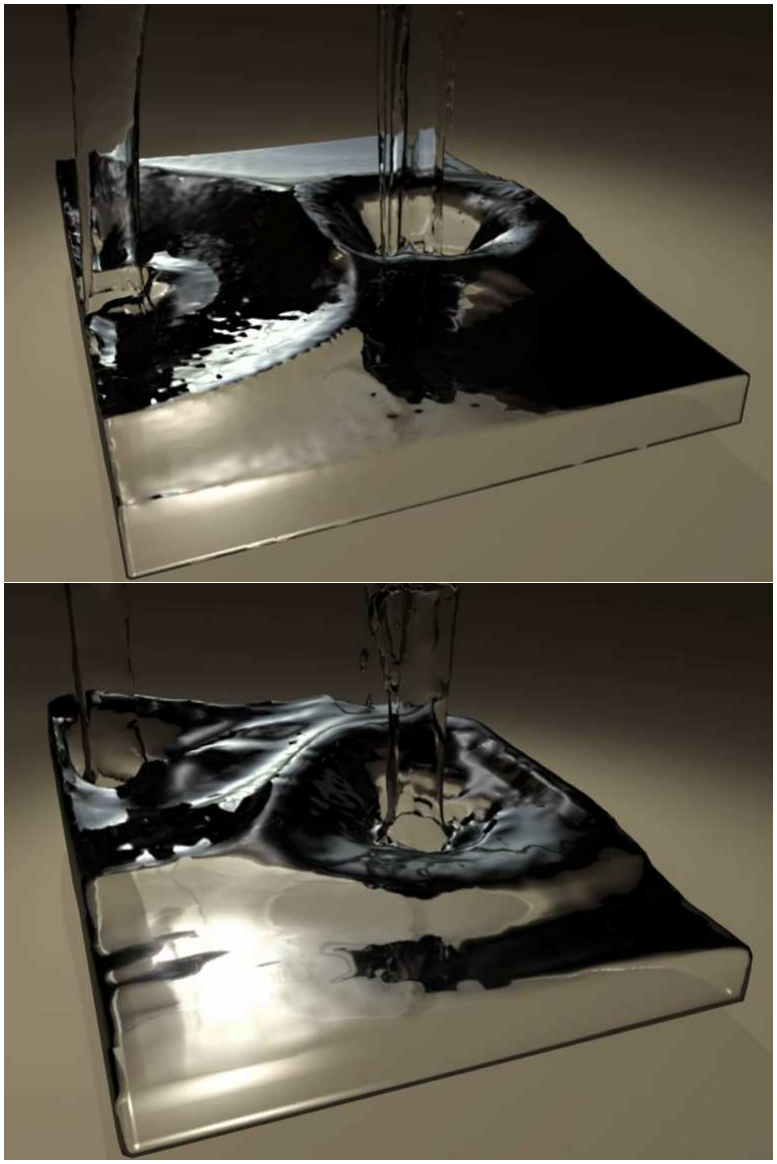

Figure 5.4: Two similar experiments showing a higher resolution (top, $80 \times 77 \times 113$) and lower resolution (bottom, $40 \times 36 \times 56$). Although both accurately describe the motion of the water, the low resolution version lacks the finer detail necessary to produce convincing surfaces

Chapter 6

Future work

Future work can be divided into two sections: that dealing with computing the motion of the water, and that dealing with tracking the surface of the water.

In dealing with the motion, the voxel-based finite difference method (FDM) could be substituted for a more general finite element method (FEM). This would have huge benefits in the quality and speed of the simulation engine, but is much more difficult to implement. Also, extending the physical properties modelled to include friction and other phenomena is an interesting topic.

In the second, an extension to the particle level set would be interesting. The method is quite expensive computation-wise, since upwards of 64 particles are placed in each voxel close to the surface. Approximating the surface through each voxel parametically using planes or curved surfaces instead of particles might help, but requires research as the particle primitive is an essential assumption to the particle level set method.

And finally, involving both surface tracking and liquid motion, improvements could be made in the treatment of static and moving obstacles. Although a level set is used to describe these, each voxel is still treated as completely inside or completely outside of the obstacle. A method using the exact value of ϕ would help keep the liquid flowing perfectly flush along boundaries. Also, the normal of obstacles is currently treated as axis-aligned. Changing this would require specifying new boundary conditions (the ones described in section 2.4.3 all assume axis-aligned normals), but would certainly help in the case of obstacles whose surfaces are at angles close to 45 degrees of the grid.

Chapter 7

Acknowledgements

I would like to thank the following: My supervisor at Digital Domain, Doug Roble. My academic supervisors, Mark Ollila and Anders Ynnerman. My family, Claes, Monica and Jonas Wrenninge. All of the very talented people involved in the FluidSim project at Digital Domain, in particular Robert Underwood, Jason Iversen, Ryo Sakaguchi and Sean Cunningham.

Bibliography

[1] D. Adalsteinsson, J. A. Sethian, *The Fast Construction of Extension Velocities in Level Set Methods* J. Comp. Phys. 148, pp. 2–22 (1999).

[2] J. D. Anderson, Jr., *Computational Fluid Dynamics*, McGraw-Hill (1995).

[3] A. J. Chorin, J. E. Marsden, *A Mathematical Introduction to Fluid Mechanics*, 3rd ed., Springer-Verlag (1990).

[4] D. Enright, R. Fedkiw, *Robust Treatment of Interfaces for Fluid Flows and Computer Graphics.*

[5] D. Enright, S. Marschner, R. Fedkiw, *Animation and Rendering of Complex Water Surfaces*, ACM Trans. on Graphics (SIGGRAPH 2002 Proceedings) 21, pp. 736–744 (2002).

[6] D. Enright, R. Fedkiw, J. Ferziger, I. Mitchell, *A Hybrid Level Set Method for Improved Interface Capturing*, J. Comp. Phys. (2002).

[7] H. Fält, *Fluid Solver For Fluid With High Viscosity Using Level Set*, 2003.

[8] R. Fedkiw, T. Aslam, B. Merriman, S. Osher, *A Non-Oscillatory Eularian Approach to Interfaces in Multimaterial Flows (The Ghost Fluid Method)*, J. Comp. Phys. 152, pp. 457–492 (1999).

[9] R. Fedkiw, J. Stam, H. W. Jensen, *Visual Simulation of Smoke*, Proceedings of SIGGRAPH 2001, pp. 15–22 (2001).

[10] N. Foster, R. Fedkiw, *Practical Animation of Liquids*, Proceedings of SIGGRAPH 2001, pp. 23–30 (2001).

[11] N. Foster, D. Metaxas, *Realistic Animation of Liquids.*

[12] D. Q. Nguyen, R. Fedkiw, H. W. Jensen, *Physically Based Modeling and Animation of Fire*, ACM Trans. on Graphics (SIGGRAPH 2002 Proceedings) 21, pp. 721–728 (2002).

[13] Eric Weisstein's World of Mathematics, http://mathworld.wolfram.com/

[14] S. Osher, R. Fedkiw, *Level Set Methods: An Overview and Some Recent Results.*

[15] S. Osher, R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*, Springer-Verlag. (2002)

[16] J. A. Sethian, *Level Set Methods and Fast Marching Methods*, Cambridge University Press, 2nd edition (1999)

[17] J. Stam, *Stable Fluids*, ACM SIGGRAPH 99, pp. 121–128. (1999)

[18] R. T. Whitaker, *A Level-Set Approach to 3D Reconstruction from Range Data*, The International Journal of Computer Vision, 29(3), October 1998, pp 203-231

[19] R. T. Whitaker, *Isosurfaces and Level-Set Surface Models*.