The Simulation of Liquids

Mario Ausseloos
mario.ausseloos@hotmail.com

In partial fulfillment
of the requirements for the degree
MSc Computer Animation

National Centre for Computer Animation
Bournemouth University
September 2006

Contents

List of symbols

$\mathbf{A}$	the matrix of a linear system, p. 20.
$\mathbf{b}$	the right-hand side of a linear system, p. 20.
c	the isocontour value of a level set, p. 24.
$\mathbf{f}$	the total body force, p. 7.
$\mathbf{f}_{\text{conf}}$	the confinement force, p. 21.
F	the speed of a dynamic implicit surface in a direction normal to itself, p. 56.
i	the index of the simulation grid in the x-direction, p. 10.
j	the index of the simulation grid in the y-direction, p. 10.
k	the index of the simulation grid in the z-direction, p. 10.
$\mathbf{M}$	the normalized gradient of the magnitude of the vorticity, p. 21.
N_x	the total number of grid cells in the x-direction, p. 10.
N_y	the total number of grid cells in the y-direction, p. 10.
N_z	the total number of grid cells in the z-direction, p. 10.
$\mathbf{N}$	the (outward) surface normal, p. 25.
O	the time cost of an algorithm, p. 27.
p	the pressure, p. 7.
$p_{i,j,k}$	the pressure value stored as the center of cell (i, j, k), p. 10.
$\mathbf{p}$	a point in space, p. 56.
P	a set of points in $\mathbb{R}^3$, p. 53.

r	the radius of all massless marker particle (used to initialize the level set), p. 30.
r_p	the radius of a massless marker particle (used in the particle level set method), p. 31.
$r_{\min}$	the minimum radius of a massless marker particle, p. 31.
$r_{\max}$	the maximum radius of a massless marker particle, p. 31.
s_p	the sign of a massless marker particle, p. 31.
$\mathbb{R}$	the vector space of the real numbers, p. 24.
$\mathbb{R}^3$	the vector space $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$, p. 24.
t	the current time, p. 7.
t_{end}	the final time of the simulation, p. 11.
$\triangle t$	the current time step in the simulation, p. 11.
T	the arrival time function, p. 55.
u	the x-component of the velocity , p. 10.
$u_{i,j,k}$	the value of the x-component of the velocity stored at the left face of cell (i, j, k) , p. 10.
$(u_0)_{i,j,k}$	the value of the x-component of the preliminary velocity field stored at the left face of cell (i, j, k), p. 13.
$\mathbf{u} = (u, v, w)^T$	the velocity, p. 7.
$\mathbf{u}_{i,j,k} = (u_{i,j,k}, v_{i,j,k}, w_{i,j,k})^T$	the velocity value associated with cell (i, j, k), p. 10.
$\mathbf{u}_p$	the velocity of a massless marker particle, p. 11.
$\mathbf{u}^t$	the velocity field obtained at time t in the simulation, p. 11.
$\mathbf{u}_0 = (u_0, v_0, w_0)^T$	the preliminary velocity computed in the simulation, p. 11.
v	the y-component of the velocity , p. 10.
$v_{i,j,k}$	the value of the y-component of the velocity stored at the lower face of cell (i, j, k) , p. 10.
w	the z-component of the velocity , p. 10.
$w_{i,j,k}$	the value of the z-component of the velocity stored at the back face of cell (i, j, k) , p. 10.

$\mathbf{x} = (x, y, z)^T$	the position (Cartesian coordinates) of a point in $\mathbb{R}^3$, p. 24.
$\mathbf{x}_p$	the position of a massless marker particle, p. 11.
$\mathbf{x}_p^t$	the position of a massless marker particle at time t in the simulation, p. 31.
$\mathbf{y}$	the unknown vector in a linear system, p. 20.
ν	the kinematic viscosity, p. 7.
ρ	the density, p. 7.
ϕ	a level set function, p. 24.
ϕ_p	the level set function associated with a massless marker particle, p. 30.
ψ	a scalar field, p. 49.
$\boldsymbol{\omega}$	the vorticity field, p. 21.
$\triangle\tau$	the grid resolution, p. 10.
$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)^T$	the 'del' or 'nabla' operator, p. 7.
$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$	the Laplacian operator, p. 7.
$\nabla \times \mathbf{u}$	the curl of the velocity field, p. 21.

Introduction

Falling rain drops, pouring a glass of milk, crashing ocean waves, lava flows, a plummeting waterfall, stirring a cup of coffee, taking a shower, a massive flood: liquid flows are everywhere. Hence, it is not surprising one often wants to include such liquid phenomena in feature films, computer games and commercials.

Until the late 1990s, the only way to obtain believable liquid effects was to recreate the liquid effect on the stage. However, this approach has its limitations. First, one often wants to exaggerate the reality or simply create effects which are physically not possible. Second, recreating the liquid effect on the stage may not always be possible because of safety reasons. Third, it is a very expensive solution for large-scale effects. Admittedly, one has tried to reduce the costs by building miniatures, but there is a trade-off between a convincing look and the cost. Indeed, because water does not behave scaled, believable water effects can only be created, in practice, if the scale factor is not less than about a quarter (Deusen et al. 2004).

Because of these reasons, there is a huge demand for computer generated liquid effects. This is a non-trivial task because liquids move and interact in complex ways due to the interplay between various phenomena such as convection, diffusion, turbulence, and surface tension (Müller et al. 2003). Liquids exhibit a wide range of behaviours such as forming waves, creating foam, merging together, breaking up into droplets, etc. Moreover, we are so familiar with how liquids behave and look, that even an almost accurate liquid animation may not be good enough to fool an audience.

There exist two categories of techniques to generate digital liquid effects: emulation and simulation techniques. Emulation methods attempt to reproduce the behaviour and look of liquid flow without modelling the underlying physics. One example is the use of particle systems combined with textures of poured salt to emulate waterfalls as used e.g. in the film The Chronicles of Narnia (2005). Although emulation techniques can be very effective for certain effects, they generally give the user too much freedom so that it is very hard to capture the dynamics of liquids in that way. Another disadvantage is that their application is generally restricted to one particular effect and that they do not generalize easily.

A more natural way to model liquids is to use the Navier-Stokes equations, which are the physical equations that describe their motion (see section 1.2). Such simulation techniques can create very realistic and complex effects and provide a general liquid model. However, they are generally very time-consuming and it is harder to control the liquid flow because it is dictated by the underlying physics.

Liquid simulation techniques can be subdivided in two categories: Eulerian and Lagrangian methods. Both methods solve the Navier-Stokes equations to obtain the velocity of the liquid at different points in space. Eulerian (*grid-based*) methods calculate the liquid velocity at fixed points in space while Lagrangian (*particle-based*) methods calculate the liquid velocity at points which move along with the liquid.

Eulerian methods can only handle small-scaled problems with the currently available computer power and they suffer from "numerical dissipation" which results in mass loss during the simulation. Lagrangian methods don't have such mass loss issues and they can handle medium-scaled problems but, unlike Eulerian methods, they have the disadvantage that it is difficult to obtain a smooth liquid surface. Moreover, much more research has been carried out on Eulerian methods (see e.g. Foster & Metaxas (1996), Stam (1999), Foster & Fedkiw (2001), Enright et al. (2002b), Carlson (2004) and many others) than on Lagrangian methods (see e.g. Müller et al. (2003), Premoze et al. (2003), and Clavet et al. (2005)) and Eulerian methods have yielded the most realistic liquid animations so far, so they are the most popular in the visual effects industry at the moment.

These research efforts have resulted in computer generated liquid effects which are convincing enough to be used in feature films. *Waterworld (1996)* was the first film with realistic CG water. In the first years, the digitally created water effects were limited to calm ocean surfaces (see e.g. *The Fifth Element (1997)*, *Titanic (1998)*, *Pearl Harbor (2001)*, and many others).

Liquid simulation was taken to the next level in *The Perfect Storm (2000)*. Live footage of real water was combined with a CG ocean which could simulate several realistic phenomena and special effects such as swells that form waves and curl forward, the interaction of water, wind and solid objects, foam and water splashing effects, etc. Basic fluid dynamics was applied to simulate the body of the ocean while particle systems were used to generate additional effects such as crests on the waves, mist, foam, water droplets, etc. These particle systems were built into the ocean simulation and created millions of particles each time a wave collided with another object (e.g. a boat, a person or another wave). Without the CG water, it would have been impossible to suggest the violence of *The Perfect Storm*. The film received a visual effects Oscar nomination for its pioneering work on liquid simulation.

Liquid simulation has become a very active research topic in the last few years, resulting in increasingly more realistic-looking and more spectacular CG liquid simulations

(see e.g. *Shrek (2001)*, *Shrek 2 (2004)*, *Poseidon (2006)*, and *Cars (2006)*). The aim of this work is to

- research the liquid simulation techniques which are currently very popular in the visual effects industry,
- implement them to create a C++ library for state-of-the-art liquid animations.

There are many choices one must make when implementing a liquid solver. I decided to focus on the design and development of an Eulerian solver which allows to create liquid simulations in two and three dimensions. This liquid solver should be able to handle free boundary value problems in which the shape of the domain occupied by the liquid changes over time.

This Master thesis is organised as follows: the first three chapters provide the reader with the scientific framework which is required to understand the author's Master project. Chapter 1 introduces (1) the mathematical equations which govern the behaviour of liquids and (2) the liquid simulation method proposed by Foster & Metaxas (1996). This method was already implemented by the author in two dimensions in the third term. It is briefly discussed here as the author's Master Project is an extension of this previous work.[1]

Chapters 2 and 3 provide the technical background which is directly relevant to the author's Master project. This consists of two main parts:

- **Part I: Extending a liquid solver from 2D to 3D** (chapter 2)

  The first part involves the extension of my previous work to an efficient liquid solver in three dimensions. Substantially more effort is required to treat free boundary value problems in 3D than 2D. Moreover, more efficient techniques have to be implemented to solve the Navier-Stokes equations in three dimensions.

- **Part II: The Liquid Surface** (chapter 3)

  The second part is concerned with the representation and visualization of the liquid surface. Implicit surfaces are presented as an alternative to parametric surfaces for geometry modelling. An introduction is given to dynamic implicit surfaces and level set methods which allow to track evolving interfaces such as the liquid surface.

A detailed description of my Master project is presented in Chapter 4. The actual implementation of the C++ library is described. Problems encountered during the implementation and the adopted solution strategies are discussed.

[1] The reader is also advised to read the reports of the author's Major Animation and Computer Animation Principles & Practice projects as they provide more details.

Finally, this thesis ends with a conclusion and an overview of future work.

Part I

Theoretical framework

Chapter 1

Solving the 2D Navier-Stokes equations

1.1 Introduction

This chapter provides the reader with the computational fluid dynamics (CFD) background which is necessary to understand the following chapters. First, the equations which describe the motion of liquids, the *Navier-Stokes equations*, are given. The physical meaning of the different terms present in these equations is explained.

After discussing the limitations of the adopted Navier-Stokes equations, we present several popular techniques used to solve them in the visual effects industry: the *staggered MAC grid*, *operator splitting* and *pressure projection*. Finally, we describe the solution method proposed by Foster & Metaxas (1996) as they were the first to introduce the (fully 3D) Navier-Stokes equations to computer animation.

1.2 The Navier-Stokes equations

Mathematically, the state of a liquid at a given instant of time is modeled by means of a velocity vector field **u** and a pressure field p . The Navier-Stokes equations describe how these two fields are coupled and how they change over time.

$$\nabla \cdot \mathbf{u} = 0 \tag{1.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} - \frac{\nabla p}{\rho} + \mathbf{f} \tag{1.2}$$

with ν the kinematic viscosity, ρ the density and **f** the total body force, which are assumed to be known. These equations were developed by Claude Navier and George Stokes in

the first half of the 19th century. Equation (1.1) and (1.2) are mathematical statements of basic conservation laws of physics: conservation of mass and Newton's second law, respectively. We refer the reader to Anderson (1995) for a detailed derivation of the Navier-Stokes equations from these physical principles.

Equation (1.2) states that the instantaneous change in liquid velocity at a given position is the result of four processes which are briefly described below.

$-(\mathbf{u}\cdot\nabla)\mathbf{u}$ (self-)advection term

expresses that "the velocity should move along itself".[1] Take the example of a fast river in which the time-variation of the water velocity is dominated by advection: any small amount of water poured into the river will quickly be swept away with the current.

$\nu\nabla^2\mathbf{u}$ diffusion term

The higher the viscosity constant ν, the faster the liquid damps out spatial variations in the velocity, which results in a "thicker", more viscous liquid.

$-\frac{\nabla p}{\rho}$ pressure term

The gradient of a scalar field is a vector field which points in the direction of the greatest rate of increase of the scalar field. Hence, the pressure term describes how liquid flows in a direction from high to low pressure.

$\mathbf{f}$ body force term

accounts for the external forces (per unit mass) that act globally on the liquid such as wind or gravity.

1.3 Limitations of the adopted Navier-Stokes equations

One should realize that the adopted liquid model as given by equations (1.1) and (1.2) has its limitations.

1. The liquid density is assumed to be constant in space and time or, in other words, one assumes an incompressible liquid. In reality no liquid is ever really incompress-

[1] This statement will become more clear in appendix A which presents the derivation of the advection equation.

ible, but incompressibility is assumed for liquids moving at low speeds because their compression is negligible (Carlson 2004).

2. The viscosity of the liquid is supposed to be constant in space. The diffusion term $\nu\nabla^2\mathbf{u}$ in equation (1.2) should be replaced with the more general term $\nabla.(\nu\nabla\mathbf{u})$ if one wishes to model variable viscosity liquids.

3. Equation (1.2) does not take surface tension forces into account. Surface tension originates from the fact that the intermolecular forces are unbalanced at the free surface. These forces become important for highly curved liquid surfaces in which case they tend to minimize the surface curvature. Surface tension effects can be modelled by adding an extra term to equation (1.2).

4. The liquid is assumed to be laminar. This means one does not model turbulent flow, the flow regime characterized by chaotic, stochastic property changes. Turbulence can be observed, for instance, in the rapids of streams when the structure of the flow is no longer visible, and only foam and bubbles can be seen (Griebel et al. 1998).

Being aware of these limitations, we adopt equations (1.1) and (1.2) as our liquid model and continue to describe how these equations can be solved in the next section.

1.4 Solving the Navier-Stokes equations

1.4.1 Introduction

The Navier-Stokes equations are very hard to solve because they are non-linear due to the presence of the (self-)advection term in equation (1.2). An analytical solution which provides the liquid velocity and pressure at all (infinitely many) points in space and time has not been found yet. Instead, numerical simulation techniques are applied which discretize the Navier-Stokes equations, i.e. consider them at only a finite number of selected points.

A lot of research has been carried out in CFD in the last few decades. Indeed, liquid simulation is very important in physical sciences and engineering as a research or design tool. One example is the study of the potentially devastating effects of a tsunami. Liquid simulation in combination with digital prototypes are also used to improve new designs of submarines before actually having to built the real submarine.

In the visual effects industry, on the other hand, liquid simulations are relatively new. Unfortunately, the tools developed in physical sciences and engineering to solve

the Navier-Stokes equations cannot be directly applied in computer animation. Indeed, liquid simulation in physics and engineering has very different goals. The primary aim in those disciplines is to obtain physically accurate results while it is all about the look in computer animation. An engineer generally does not know the results in advance while an animator wants to create a particular effect. These fundamental differences make that solution methods in computer animation must satisfy different criteria.

First, the specification of the initial and boundary conditions should happen automatically so that an animator does not need to have a clear understanding of the underlying equations. Second, control structures have to be implemented which allow the animator to control the liquid flow to produce a desired effect. Note that this is a challenging task because one still wishes realistic liquid flow as if it was only governed by the underlying physics. Because there are no easy mechanisms for controlling the simulation, time becomes an important issue in computer animation. Indeed, several trials may be necessary to obtain a certain effect so from a practical point of view, one cannot afford that one run takes hours.

To conclude, the liquid simulation techniques developed in physical sciences and engineering cannot be directly applied to computer animation and a lot of research is needed to come up with new methods which are useful in the visual effects industry. In the following sections, we describe several techniques which have been frequently used in computer animation to solve the Navier-Stokes equations.

1.4.2 The MAC method

As mentioned in the previous section, numerical simulation methods consider equation (1.1) and (1.2) at only a finite number of selected points. The *Marker-And-Cell (MAC) method*, which was originally described by Harlow & Welch (1965), defines these points by means of a MAC grid. This is a fixed rectangular grid which subdivides the environment in cubes (called *cells* or *voxels*). The grid resolution is designated as $\triangle\tau$ and the number of voxels in the x, y and z-direction as N_x, N_y and N_z, respectively. The symbols i, j and k denote the grid index in the x, y and z-direction, respectively.

All scalar flow-field variables such as the pressure p are stored at the cell centers, sometimes referred to as the *cell nodes*. Vector values such as the velocity $\mathbf{u} = (u, v, w)^T$ are stored in a *staggered grid* formation. The latter means that the three components of the vector are stored at the center of the left, lower, and back face of the cell respectively. These computational nodes are sometimes referred to as *face nodes*. Such a staggered grid formation is more complex to implement but it is well known from the CFD literature that it yields better results (see e.g. Anderson (1995)). The discretized pressure and velocity values associated with cell (i, j, k) are denoted as $p_{i,j,k}$ and $\mathbf{u}_{i,j,k} = (u_{i,j,k}, v_{i,j,k}, w_{i,j,k})^T$,

respectively.

Assuming the initial values for the velocity $\mathbf{u}$ (and pressure p) at time $t = 0$ are given, time is incremented by $\triangle t$ in each step of an outer loop until the final time t_{end} is reached. In each time-stepping loop the velocity (and pressure) field is updated: the Navier-Stokes equations are solved to find the values $\mathbf{u}^{t+\Delta t}$ at time $t+\Delta t$ based on the values $\mathbf{u}^t$ obtained in the previous time step.

Furthermore, the MAC method uses a large collection of *massless marker particles* to handle free boundary value problems; by passively advecting the particles with the liquid flow, they can be used to determine the domain occupied by the fluid at a particular time. In practice, this proceeds as follows: at the beginning of each simulation step, all non-obstacle cells are marked empty. Subsequently, all empty cells which contain at least one particle, are flagged as a fluid cell. Finally, all fluid cells bordering at one or more empty cells, are marked as surface cells. At the end of each simulation step, the position of the massless marker particles is updated according to the inertialess equation

$$\frac{d\mathbf{x}_p}{dt} = \mathbf{u}_p \tag{1.3}$$

with $\mathbf{x}_p$ the particle position and $\mathbf{u}_p$ the fluid velocity at $\mathbf{x}_p$. The velocity $\mathbf{u}_p$ is calculated by interpolating the fluid velocity at the neighbouring computational grid nodes.

1.4.3 Operator splitting

The idea of operator splitting is to separate the right-hand components of a partial differential equation (PDE) such as equation (1.2) into multiple terms, and to calculate these terms in sequence, independently of one another. The obvious advantage of this technique is that each term can be solved with a different algorithm so that the most efficient algorithm can be selected for solving each term.

Using partial operator splitting, one can separate out the pressure term in the Navier-Stokes equations and solve equation (1.2) as follows:

- First, solve
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + \mathbf{f}, \qquad \mathbf{u}(t) = \mathbf{u}^t \tag{1.4}$$
to obtain an intermediate velocity field $\mathbf{u}_0$ and subsequently,
- solve
$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{\nabla p}{\rho}, \qquad \mathbf{u}(t) = \mathbf{u}_0 \tag{1.5}$$

to obtain the updated velocity field $\mathbf{u}^{t+\Delta t}$ at time $t + \Delta t$.

Note that equation (1.5) is calculated based on the intermediate value $\mathbf{u}_0$ of the velocity instead of the original velocity $\mathbf{u}^t$ at time t.

The updated velocity field $\mathbf{u}^{t+\Delta t}$ should also satisfy the mass conservation principle (equation 1.1). This is the subject of the following section.

1.4.4 Pressure Projection

Using Euler's method to discretize the time derivative in equation (1.5),

$$\frac{\partial \mathbf{u}}{\partial t} = \frac{\mathbf{u}^{t+\Delta t} - \mathbf{u}_0}{\Delta t},$$

we can rewrite this equation as follows

$$\mathbf{u}^{t+\Delta t} = \mathbf{u}_0 - \Delta t \frac{\nabla p}{\rho}. \tag{1.6}$$

Enforcing that the updated velocity field $\mathbf{u}^{t+\Delta t}$ satisfies equation (1.1)

$$0 = \nabla \cdot \mathbf{u}^{t+\Delta t} = \nabla \cdot \mathbf{u}_0 - \Delta t \frac{\nabla^2 p}{\rho},$$

yields a Poisson equation for the pressure

$$\nabla^2 p = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u}_0. \tag{1.7}$$

In summary, computing the values $\mathbf{u}^{t+\Delta t}$ at time $t+\Delta t$ based on the values $\mathbf{u}^t$ obtained in the previous time step consists of the following three steps:

1. Compute a preliminary velocity field $\mathbf{u}_0$ according to equation (1.4) from the velocity field $\mathbf{u}^t$ obtained in the previous time step.
2. Solve the Poisson equation (equation 1.7) for the pressure using the preliminary velocity field $\mathbf{u}_0$ obtained in the first step.
3. Compute the updated velocity field $\mathbf{u}^{t+\Delta t}$ using equation (1.6) with the velocity field $\mathbf{u}_0$ obtained in the first step and the pressure values obtained in the second step.

This approach corresponds to the Chorin projection method developed by Chorin (Chorin 1968) and Temam (Temam 1969).

1.4.5 An explicit solution method

The MAC grid (section 1.4.2), operator splitting (section 1.4.3), and pressure projection (section 1.4.4) have been popular choices to solve the Navier-Stokes equations in computer animation (see e.g. Foster & Metaxas (1996), Stam (1999), Foster & Fedkiw (2001), Carlson et al. (2002), and many others). The methods described in these papers differ in the algorithms used to compute the preliminary velocity field (equation 1.4) and the pressure field (equation 1.7). This section discusses the approach taken by Foster & Metaxas (1996).

Computation of the preliminary velocity field

Foster & Metaxas (1996) use a numerical technique known as finite differences to discretize equation (1.4). The resulting discretized version of the u-component of equation (1.4) is given as an example:

$$\begin{aligned}(u_0)_{i,j,k} = u_{i,j,k} &+ \frac{\triangle t}{4\triangle \tau}\Big[(u_{i-1,j,k}+u_{i,j,k})^2 - (u_{i,j,k}+u_{i+1,j,k})^2 \\ &+ (u_{i,j-1,k}+u_{i,j,k})(v_{i-1,j,k}+v_{i,j,k}) \\ &- (u_{i,j,k}+u_{i,j+1,k})(v_{i-1,j+1,k}+v_{i,j+1,k}) \\ &+ (u_{i,j,k-1}+u_{i,j,k})(w_{i-1,j,k}+w_{i,j,k}) \\ &- (u_{i,j,k}+u_{i,j,k+1})(w_{i-1,j,k+1}+w_{i,j,k+1})\Big] \\ &+ \frac{\nu\triangle t}{(\triangle\tau)^2}\Big[u_{i-1,j,k}+u_{i,j-1,k}+u_{i,j,k-1} \\ &- 6u_{i,j,k}+u_{i+1,j,k}+u_{i,j+1,k}+u_{i,j,k+1}\Big] \\ &- \frac{\triangle t}{\rho\triangle\tau}(p_{i,j,k}-p_{i-1,j,k}) \\ &+ f_x\triangle t \end{aligned} \tag{1.8}$$

Analogous expressions can be derived for the v- and w-component of equation (1.4). Griebel et al. (1998) explain the finite differences technique and derive the discretized version of the Navier-Stokes equations.

Equation (1.8) indicates that the x-component of the preliminary velocity $\mathbf{u}_0$ at a particular cell face can be obtained based on the velocity values obtained at neighbouring cell faces at the previous time step. This implies the need of boundary conditions at the border of the simulation grid and at the interface between liquid cells on the one hand, and obstacle and empty (air) cells on the other hand.

In order to account for the boundary conditions at the border of the simulation grid, an artificial boundary strip of grid cells is added internally. Hence, the computational grid has dimensions

$$(N_x + 2) \times (N_y + 2) \times (N_z + 2)$$

internally. The extra layer of cells around the simulation grid are considered obstacle cells and treated the same as the internal obstacle cells with regard to the application of boundary conditions. Therefore, only two different types of boundary conditions have to be considered:

- Boundary conditions between fluid cells and obstacle cells

  These boundary conditions can be set in different ways depending on the desired liquid behaviour:

  - *no-slip conditions* which model frictional forces between the liquid and surrounding obstacles,
  - *free-slip conditions* which model no frictional forces between the liquid and surrounding obstacles,
  - *inflow conditions* which allow liquid to move freely in the computational domain, and
  - *outflow conditions* which allow liquid to move freely out the computational domain.

  The reader is referred to Foster & Metaxas (1997) for a more detailed discussion of this type of boundary conditions and ways how they can be enforced.

- Boundary conditions between fluid cells and empty cells

  Intuitively, boundary conditions have to be set at the liquid surface so that air does not mix with or inhibit the liquid motion, while allowing it to flow freely into empty cells (Foster & Fedkiw 2001). This is done by explicitly enforcing incompressibility within each surface cell. More details will be given in section 2.2.

The implementation of these boundary conditions is very important. Indeed, given that the behaviour of "all" viscous incompressible liquid is governed by the Navier-Stokes equations, the boundary conditions are the real driver for any particular solution.

It should also be noted that some velocity values in obstacle cells or empty cells neighbouring liquid cells have to be set although they are not needed for the solution of the

Navier-Stokes equations. Indeed, as discussed in section 1.4.2, the position of the massless marker particles is updated by interpolating the velocity values at neighbouring cells. For particles at the surface, this sometimes means that velocity values outside the liquid are needed. Griebel et al. (1998) describes how these velocity values can be set in two dimensions. The extension to three dimensions is given in section 2.2.

Once all the boundary conditions are set, one can solve equation (1.8). This equation may seem complex at first, but all values at the right-hand side are obtained in the previous simulation step. Hence, it is relatively straightforward to implement this technique and obtain the preliminary velocity $\mathbf{u}_0$.

Computation of the pressure field

Using finite differences, one can transform equation (1.7) in a very large, sparse linear system of equations (see e.g. Griebel et al. (1998) for more details). Solving such linear systems involve setting boundary conditions for the pressure and applying an appropriate linear equation solver.

Foster & Metaxas (1996) set the pressure in obstacle cells equal to the pressure in the adjacent fluid cell to prevent fluid flowing across the boundary. The pressure in a surface cell is set to the atmospheric pressure.

Solving very large, sparse linear systems by means of direct methods such as Gaussian elimination are too costly in terms of computer time and storage. Instead, iterative solution methods are generally applied. Classic examples of such iterative techniques are Jacobi and Gauss-Seidel (Golub & Van Loan 1983). Gauss-Seidel, for example, starts from an initial approximation and successively processes each cell (i, j, k) once in every iteration step by modifying its pressure value in such a way that equation (1.7) is satisfied exactly. Foster & Metaxas (1996) used a Successive Over Relaxation (SOR) method, which is an improved variant of the Gauss-Seidel iteration scheme.

Chapter 2

Solving the 3D Navier-Stokes equations

2.1 Introduction

This is first of two chapters which describe the theoretical background which is directly relevant to the author's Master project. The representation and tracking of the liquid surface in 3D is the subject of the next chapter. This chapter discusses the extension of the liquid solver from 2D to 3D.

From a mathematical-numerical standpoint, there are no fundamental difficulties in extending the basic two-dimensional code to the three-dimensional case. However, solving free boundary value problems involves setting the boundary conditions at the surface, which requires substantially more effort in 3D as will be explained in section 2.2.

After these adaptations of the liquid solver, one can, in principle, simulate liquids in 3D. However, a straightforward implementation of the method of Foster & Metaxas (1996) (section 1.4.5) is too slow to be useful in a production environment. Therefore, more efficient methods were implemented to compute

- the advection term in equation (1.4) [section 2.3],
- the diffusion term in equation (1.4) [section 2.4], and
- the pressure equation (equation 1.7) [section 2.5].

Section 2.6 introduces the vorticity confinement method which is used to counteract the non-physical dampening of the liquid motion which is due to the coarse grids which are typically adopted in computer animation.

2.2 Boundary conditions at the liquid surface

Each fluid cell in two dimensions has four neighbouring grid cells which can be either empty (filled with air) or not. This results in $2^4 = 16$ different cases. In the case that all four neighbouring cells are not empty, no surface boundary conditions have to be set. Therefore 15 different cases have to be considered for the surface boundary conditions in two dimensions. These cases are enumerated in Foster & Metaxas (1996) and Griebel et al. (1998).

In three dimensions, however, each fluid cell has 8 neighbouring cells, resulting in $2^6 - 1 = 63$ different cases which have to be examined. Carlson (2004) briefly describes how the surface boundary conditions can be set in these 63 cases.

As mentioned in section 1.4.5, velocity values outside the liquid are sometimes needed for the computation of the massless marker particles' position. Enright et al. (2002b) proposed a method which extrapolates the velocity along the normal of the liquid surface. But we do not have normal information at the surface available at this point. Alternatively, one can adopt the method suggested by Griebel et al. (1998). These authors only describe the two dimensional case. The three-dimensional case, however, is significantly more complex. Given the lack of information available in the literature on setting the surface boundary conditions in 3D in this way, section 4.3.1 will be devoted to this subject.

2.3 Solving the advection term

Remember from section 1.4.5 that Foster & Metaxas (1996) solved equation (1.4) by means of a finite difference technique. The explicit nature of this method imposes restrictions on the time step with which the velocity field can be updated.

The advection term $-(\mathbf{u} \cdot \nabla)\mathbf{u}$ in equation (1.4) leads to a stability restriction known as the Courant-Friedrichs-Lewy (CFL) condition. The CFL condition states that the time step must be small enough to make sure information does not travel across more than one cell at a time:

$$\Delta t < \frac{\Delta\tau}{\max(|u|, |v|, |w|)} \tag{2.1}$$

where $\Delta\tau$ is the size of the grid cells used in the simulation.

This condition can be overly restrictive for liquids flowing at high speeds or in case a fine simulation grid is desired. This is a well known disadvantage of Eulerian advection schemes. Lagrangian advection schemes can often use much larger time steps than Eulerian ones but they have the disadvantage that an initially regularly spaced set of par-

ticles will generally evolve to a highly irregularly spaced set at later times, and important features of the flow may consequently not be well represented (Staniforth & Cote 1991). The idea behind semi-Lagrangian methods is to try to get the best of both worlds: the regular resolution of Eulerian schemes and the enhanced stability of Lagrangian ones. This technique was first invented in 1952 by Courant, Rees and Isaacson and has been rediscovered by many researchers in different fields. It was introduced to computer graphics to solve the self-advection term of the Navier-Stokes equations by Stam (1999). We refer the reader to appendix B for a description of a first-order semi-Lagrangian method.

The subject of the author's Computer Animation Principles and Practice Project was to replace the finite difference scheme with such a semi-Lagrangian method (Stam 1999) to compute the advection term. This work resulted in significant performance improvements (typically a reduction of a factor 5 in simulation time) at the cost of increased rotational damping. Section 2.6 describes the vorticity confinement method which re-injects the lost energy back into the simulation.

2.4 Solving the diffusion term

The previous section explained that solving the advection term of the Navier-Stokes equations with the finite difference technique leads to a restriction on the time step with which the liquid velocities can be updated. Similarly, adopting an explicit method to solve the viscosity diffusion term $\nu\nabla^2\mathbf{u}$ in equation (1.4) imposes another stability restriction. This restriction is given by the following equation in three dimensions:

$$\Delta t < \frac{(\Delta\tau)^2}{6\nu}. \tag{2.2}$$

This stability criterion becomes more stringent than the CFL condition at high viscosities. Therefore, Carlson et al. (2002) proposed an implicit numerical method to solve the diffusion term in equation (1.4).

Using operator splitting to separate out the diffusion term, similarly as was done for the pressure term in section 1.4.3, we obtain

$$\mathbf{u}^{t+\Delta t} = \mathbf{u}_0 + \Delta t\nu\nabla^2\mathbf{u}_0. \tag{2.3}$$

This is an explicit viscous diffusion formulation. If we replace this equation with the implicit formulation using implicit backwards Euler integration,

$$\mathbf{u}^{t+\Delta t} = \mathbf{u}_0 + \Delta t\nu\nabla^2\mathbf{u}^{t+\Delta t}, \tag{2.4}$$

we are able to make the diffusion step stable even with large time steps for high-viscosity liquids.

Equation (2.4) can be rewritten as follows

$$(1 - \Delta t \nu \nabla^2)\mathbf{u}^{t+\Delta t} = \mathbf{u}_0,$$

or in matrix notation,

$$\mathbf{A}\mathbf{y} = \mathbf{b} \tag{2.5}$$

with

- $\mathbf{A} \equiv (1 - \Delta t \nu \nabla^2)$ a positive definite matrix,
- $\mathbf{y} \equiv \mathbf{u}^{t+\Delta t}$ the unknown velocity values, and
- $\mathbf{b} \equiv \mathbf{u}_0$ the known velocities obtained in the previous step.

The most naive way to solve this linear system would be to find $\mathbf{A}^{-1}$ directly and then set $\mathbf{y} = \mathbf{A}^{-1}\mathbf{b}$. Indeed, finding the inverse of the matrix $\mathbf{A}$ is an expensive operation which we can not carry out reliably when the size of the matrix is large, because of numerical error.

A good alternative is the conjugate gradient method. This is a linear system solver which is based on the fact that finding

$$\mathbf{y} = \mathbf{A}^{-1}\mathbf{b}$$

is equivalent to minimizing

$$\frac{1}{2}\mathbf{y}^T\mathbf{A}\mathbf{y} - \mathbf{y}^T\mathbf{b}.$$

2.5 Solving the pressure term

The Poisson equation for the pressure

$$\nabla^2 p = \frac{\rho}{\Delta t}\nabla \cdot \mathbf{u}_0. \tag{2.6}$$

was derived in section 1.4.4. This equation can be recast in a linear system $\mathbf{A}\mathbf{y} = \mathbf{b}$ by letting

$$\mathbf{A} \equiv \nabla^2, \qquad \mathbf{y} \equiv p, \qquad \mathbf{b} \equiv \frac{\rho}{\Delta t}\nabla \cdot \mathbf{u}_0.$$

Recall that Foster & Metaxas (1996) solved this linear system with a Successive Over Relaxation (SOR) method. Although this method is straightforward to implement, it is well known that there exist more efficient linear systems solvers such as the conjugate gradient technique. Indeed, the pressure projection matrix is very similar to the implicit diffusion matrix (equation 2.4) so that both linear systems can be solved with the same numerical technique.

2.6 The Vorticity Confinement method

Both the use of coarse grids and a semi-Lagrangian method to compute the advection term result in "numerical dissipation": the fluid dampens faster than in reality. In the case of the semi-Lagrangian method, this is because to enforce stability additional damping is added to the flow.

Steinhoff & Underhill (1994) invented a technique called "vorticity confinement" for the numerical computation of complex turbulent flow fields around helicopters where it is not possible to add enough grid points to accurately resolve the flow. The basic idea is to re-inject the energy lost due to dissipation back into the fluid, through a clever force field which encourages the generation of the small scale details. In incompressible flows this small scale structure is provided by the vorticity field which is defined by the curl of the velocity

$$\boldsymbol{\omega} = \nabla \times \mathbf{u}.$$

It is a measure of how much the flow rotates. First, the normalized gradient of the magnitude of the vorticity is calculated

$$\mathbf{M} = \frac{\nabla|\boldsymbol{\omega}|}{|\nabla|\boldsymbol{\omega}||},$$

which results in a vector field pointing from lower to higher vorticity concentrations and hence, pointing inwards to the vortex center. In a second step, a confinement force

$$\mathbf{f}_{\text{conf}} = \varepsilon \Delta\tau (\mathbf{M} \times \boldsymbol{\omega})$$

perpendicular to this gradient field is calculated which keep the vortices alive and well localized. The parameter ε controls the amount of small scale detail added back into the flow field and the dependence on the voxel size $\Delta\tau$ guarantees that as the mesh is refined the physically correct solution is still obtained.

The vorticity confinement method has been used by many authors. Fedkiw et al. (2001) applied this technique to create nice looking swirling smoke simulations while Feldman et al. (2003) used the vorticity confinement method to enhance the gas flow in their animation of suspended particle explosions.

Chapter 3

The Liquid Surface

3.1 Introduction

Initially, massless marker particles were used to delineate the fluid from the empty air (section 1.4.2). This Lagrangian technique is easy to implement. has a low computational overhead, and provides a convenient way to carry information into neighbouring fluid cells or empty cells from the previous time step. Moreover, the particles contain more detailed information about the liquid shape than the simulation grid.

However, there is no straightforward way to extract a polygonal or parametric representation of the liquid surface from these particles as they do not contain connectivity information. In an attempt to combine a polygonal representation with the advantages of particles, one could represent the liquid surface with a triangular mesh and update its position and shape by treating the vertices as particles. However, this approach quickly leads to several problems. First, the triangles will distort when the liquid deforms so that one needs to smooth and regularize the polygonal mesh periodically. Second, even the most trivial velocity fields result in topology changes, which can only be handled with rather complicated techniques involving detaching and reattaching boundary elements. Third, as the particles do not generally form a smooth surface, the resulting polygonal mesh suffers from temporal aliasing as the polygons "pop" in or out.

The use of such Lagrangian methods to track an interface according to equation 1.3, along with numerical techniques for smoothing, regularization, and surgery, are collectively known as front tracking methods (Osher & Fedkiw 2002). We refer the interested reader to Tryggvason et al. (2001) for a current state-of-the-art review.

In recent years, another representation for the free surface, a level set, has become very popular in computer graphics. This chapter presents this Eulerian technique to model

and track the liquid surface. Section 3.2 introduces the reader to the representation and visualization of level sets as well as to the terminology and concepts of this field. Section 3.3 add dynamics to implicit surfaces: numerical techniques for computing the motion of implicit surfaces, known as level set methods, are discussed. Special attention is given to the particle level set method which was invented by Enright et al. (2002a). Finally, section 3.4 gives a brief overview of the wide range of applications of level sets.

3.2 Implicit Surfaces and Level Sets

3.2.1 Introduction

Any mathematical function of the form

$$\phi : \mathbb{R}^3 \rightarrow \mathbb{R} : \mathbf{x} \rightarrow \phi(\mathbf{x})$$

implicitly defines a surface as the set of points which map to zero,

$$\{\mathbf{x} \in \mathbb{R}^3 | \phi(\mathbf{x}) = 0\}.$$

The related level set (also called isocontour or level surface) is given by

$$\{\mathbf{x} \in \mathbb{R}^3 | \phi(\mathbf{x}) = c\},$$

where c is the isocontour value of the surface. Therefore, the implicit surface defined by ϕ is also called the zero set of ϕ, and ϕ is called the level set function or implicit function.

As an example, consider the function

$$\phi : \mathbb{R}^3 \rightarrow \mathbb{R} : (x, y, z) \rightarrow \phi(x, y, z) = x^2 + y^2 + z^2 - 1.$$

The corresponding implicit surface or zero set

$$\{(x, y, z) \in \mathbb{R}^3 | \phi(x, y, z) = 0\}$$

is the unit sphere in three dimensions.

Implicit surface representations have some very nice properties. For example, since the surface was matched with zero level set of ϕ, it is easy to determine whether a given point $\mathbf{x}$ is inside, outside or on the surface based on the local sign of ϕ. In the above example, $\mathbf{x}$ is inside the sphere when $\phi(\mathbf{x}) = x^2 + y^2 + z^2 - 1 < 0$, outside when $\phi(\mathbf{x}) > 0$ and on the surface when $\phi(\mathbf{x}) = 0$.

Implicit functions make boolean operations and constructive solid geometry operation easy to apply. If ϕ_1 and ϕ_2 are two different level set functions, then

- $\phi(\mathbf{x}) = min(\phi_1(\mathbf{x}), \phi_2(\mathbf{x}))$ defines the union of the interior regions of ϕ_1 and ϕ_2,
- $\phi(\mathbf{x}) = max(\phi_1(\mathbf{x}), \phi_2(\mathbf{x}))$ defines the intersection of the interior regions of ϕ_1 and ϕ_2,
- $\phi(\mathbf{x}) = -\phi_1(\mathbf{x})$ defines the complement of the interior region of ϕ_1, and
- $max(\phi_1(\mathbf{x}), -\phi_2(\mathbf{x}))$ represents the region obtained by subtracting the interior of ϕ_2 from the interior of ϕ_2.

The gradient of the implicit function is perpendicular to the isocontours of ϕ and points in the direction of increasing ϕ. Hence, the (outward) normal to the surface can be easily computed as follows:

$$\mathbf{N} = \frac{\nabla\phi}{|\nabla\phi|}.$$

Furthermore, techniques have been developed to

- convert between parametric and implicit forms,
- manipulate blends of geometric primitives, such as meta-balls and soft objects, which facilitates the design of smooth, complex, organic-looking shapes,
- easily deform implicit surfaces,
- represent detail hierarchically so that, during rendering, an appropriate level of detail is chosen, and
- visualize implicit surfaces.

Hence, implicit surfaces are increasingly competing with the well-established parametric surfaces used for geometry modelling (Bloomenthal & Wyvill 1997).

3.2.2 Rendering Implicit Surfaces

Different methods have been developed to visualize implicit surfaces depending on whether the corresponding level set function ϕ is given as a mathematical expression or as a discrete 3D data set. We only consider the latter case here. There are two classes of techniques to render implicit surfaces:

1. Intermediate geometric representation techniques extract an intermediate geometric representation of the surface and then use conventional techniques to render the surface (Watt & Watt 1991). The most common intermediate surface representation consists of polygons. The conversion of an implicit surface to a polygonal mesh is called polygonization, tiling, or tessellation. The Marching-Cubes algorithm (Lorensen & Cline 1987) is probably the most well-known polygonization method and is discussed in section 3.2.4.

2. Direct rendering methods visualize an implicit surface directly without attempting to impose any geometric structure upon it. Volume visualization and ray tracing are examples of such direct rendering techniques. The reader is referred to Bloomenthal & Wyvill (1997) for a detailed discussion of direct rendering techniques.

Creating an intermediate surface representation has the advantage that conventional techniques can be used to render the implicit surface. Moreover, conventional graphics environments are optimized for polygon display and manipulation so that, in practice, polygonization followed by polygon rendering is often more efficient than direct rendering methods. However, rendering directly from the discrete 3D data set reduces the volume of data and makes it possible to zoom in on fine detail in a model without losing quality (Bloomenthal & Wyvill 1997). Both types of techniques have been adopted to visualize liquid surfaces in computer animation. For example, Foster & Fedkiw (2001) used a ray-tracing technique while a method based on the Marching-Cube algorithm was adopted by Carlson (2004). The next section is devoted to polygonization methods.

3.2.3 Implicit Surface Polygonization

At a high level, a polygonizer performs the following two tasks:

1. adopting a spatial partitioning to divide space into semi-disjoint cells[1] that completely enclose the implicit surface,

2. fitting one or more polygons to the implicit surface in each cell intersected by this surface.

Bloomenthal & Wyvill (1997) distinguish three classes of spatial partioning methods:

1. continuation: a class of incremental surface tiling techniques which track the surface from a seed point.

[1] "Semi-disjoint cells" means adjacent but nonoverlapping cells.

2. exhaustive enumeration: all grid cells are examined to determine whether they intersect the surface.

3. subdivision: the recursive division of space into subvolumes. These subvolumes are often organised hierarchically (e.g. octree, k-d tree) where only the leaf nodes produce polygons.

Generally, continuation and subdivision are used to partition space in the case of continuous data while exhaustive enumeration is adopted for discrete data. Continuation methods are more efficient than exhaustive enumeration (time cost $O(n^2)$ compared to $O(n^3)$ with n some measure of the size of the object). However, unlike exhaustive enumeration techniques, continuation methods do not guarantee the detection of all pieces of a set of disjoint surfaces.

3.2.4 Marching Cubes

As an example, the popular marching cubes technique (Lorensen & Cline 1987) is discussed in this section. This algorithm subdivides the environment in cubic cells by means of an axis-aligned partition. A value zero or one is associated with each cell vertex depending on whether it lies inside or outside the liquid. This bit value is called the polarity of the corresponding vertex. If a cell edge connects two vertices with a different polarity, it is intersected by the implicit surface. The algorithm assumes that the surface intersect a cell edge at most once. This condition can be fulfilled by adopting a partition with a sufficiently high resolution.

As each of the eight cell vertices can have two possible polarity values (0 or 1), there are $2^8 = 256$ cell configurations which have to be considered. In practice, a table is constructed storing the cell edges intersected by the implicit surface for each of the possible configurations of the cell vertex polarities. The intersection of the implicit surface and each of those cell edges is usually computed by linear interpolation. These intersection points are called the surface vertices. Finally, the surface vertices of each cell are connected to form one or more polygons which approximate the implicit surface.

Unfortunately, some polarity combinations are ambiguous. In those cases, the eight polarity values do not lead to a unique way to connect the resulting surface vertices. The original marching cubes method produces errant holes in the surface because it treats these ambiguities inconsistently. Several slightly more complex techniques have been proposed which extend the original marching cubes method to resolve these ambiguities (see e.g. Bloomenthal (1988)). Alternatively, adopting tetrahedra instead of cubes as polygonizing cells naturally removes all ambiguous cases. The latter technique is described in more detail in the following section.

3.2.5 Marching Tetrahedra

The Marching Tetrahedra method as suggested by Payne & Toga (1990) decomposes each cell into six tetrahedra which are then polygonized. This algorithm was developed to circumvent the patent on the marching cubes method[2] and to resolve the ambiguity problem mentioned in section 3.2.4.

The overall algorithm structure is the same as for the Marching Cubes algorithm described in the previous section. In the case of the Marching Tetrahedra method, however, there are only $2^4 = 16$ possible configurations of the cell vertex polarities, which makes the implementation more straightforward. The tetrahedral decomposition yields a greater number of surface vertices per surface area than cubical polygonization because seven extra edges are created if a cube is subdivided into tetrahedra. Therefore, tetrahedral polygonization requires more computation time but leads to a better approximation of the implicit surface for a given number of cubical cells.

3.3 Dynamic Implicit Surfaces and Level Set Methods

3.3.1 Introduction

In the previous section, a level set function ϕ was used to represent a surface. In this section, we consider how ϕ can also be used to evolve this surface.

The zero level set of ϕ can be considered as simply being passively advected by the velocity field $\mathbf{u}$ obtained from solving the Navier-Stokes equations. The derivation of this advection equation for an arbitrary scalar field is given in appendix A. Applied to the level set function, this results in the following equation

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = 0. \tag{3.1}$$

This equation, known as the level set equation, was introduced by Osher & Sethian (1988). It defines the evolution of the implicit function ϕ caused by the velocity field $\mathbf{u}$. Level set methods are numerical techniques for efficiently solving the level set equation.

Equation 3.1 is an Eulerian formulation of the interface evolution. Unlike standard Lagrangian techniques which typically require ad hoc techniques to address mesh connectivity during merging and pinching, this approach can easily handle gross changes to interface topology. Furthermore, this approach creates a very smooth surface. One difficulty with level set methods is that they suffer from severe volume loss especially on the

[2]The patent on marching cubes has expired in 2005.

coarse grids commonly used in computer graphics. This is clearly visible when regions of liquid break away during splashing and then disappear because they are too small to be resolved by the level set (Foster & Fedkiw 2001). Lagrangian techniques, on the other hand, have difficulties with creating a smooth surface (see section 3.1) but they are much better at conserving volume. Hence, research has been carried out to find efficient hybrid methods which combine the best of both worlds.

Foster & Fedkiw (2001) focused on modelling the liquid volume by adding massless marker particles at the inside of the liquid surface (zero level set). These particles and the level set ϕ are evolved forward in time separately at each time step. Next, the particles are used for detecting the regions in which the level set has lost volume, and for correcting the level set locally in these underresolved regions. The particle level set method (Enright et al. 2002a,b), on the other hand, focuses on modelling the liquid surface. Particles are randomly placed at both sides of the liquid surface and are again used to correct the level set in case they detect errors due to numerical dissipation.

Until recently, it was believed that computationally expensive, high order accurate numerical discretizations in time and space were required to update the particles and the level set. However, Enright et al. (2004) showed that fast, low order accurate numerical schemes suffice. This fast and accurate particle level set method is described in detail in the next section.

3.3.2 The Particle Level Set method

To recapitulate, the basic idea is to:

1. embed the liquid interface as the zero level set of a higher-dimensional function ϕ, and

2. update the shape and position of the liquid surface indirectly by evolving the implicit function ϕ.

The particle level set (PLS) method evolves ϕ accurately on relatively coarse grids by (1) initializing the level set function ϕ and particles at both sides of the zero level set of ϕ and (2) carrying out the following six steps after solving the Navier-Stokes equations at each time step:

1. updating the position of the massless marker particles,

2. updating the position of the level set function,

3. correcting the level set function by means of the particles,

4. reinitializing the level set function,

5. correcting the level set function again by means of the particles,

6. adjusting the size of the particles.

These different components of the level set method are described next.

Initialization of the level set function

The PLS method updates the position of ϕ by means of equation (3.1). This requires an initial level set function from which the computation should start. One way of constructing this initial implicit function, based on the approach taken by Foster & Fedkiw (2001), involves the following steps:

1. Use massless marker particles to delineate the liquid volume as was done in chapter 1.

2. Associate a spherical implicit function ϕ_p with each particle,

$$\phi_p(\mathbf{x}) = |\mathbf{x} - \mathbf{x}_p| - r,$$

   with r a user-definable radius.

3. Define the initial level set function $\phi(\mathbf{x})$ as the $\phi_p(\mathbf{x})$ value of the particle closest to the position $\mathbf{x}$.

4. In order to ensure that this level set function is a smoothly varying function well suited for accurate numerical computations, ϕ should be turned into a signed distance function which satisfies the property

$$|\nabla\phi| = 1. \tag{3.2}$$

   Sethian (1999) mentions several techniques to make an implicit function satisfy the signed distance property. One efficient method is the Fast Marching method which is described in appendix D. (Note that equation (3.2) is equivalent to equation (D.1) with $F = 1$.)

Initialization of the massless marker particles

Inertialess marker particles are placed in a band of about three grid cells wide on each side of the zero level set. A user-definable number of positive and negative particles are located in the $\phi > 0$ and $\phi < 0$ region, respectively. Each particle has a radius, r_p which is initialized as follows

$$r_p = \begin{cases} r_{\max} & \text{if } s_p\phi(\mathbf{x}_p) > r_{\max} \\ s_p\phi(\mathbf{x}_p) & \text{if } r_{\min} \leq \text{if } s_p\phi(\mathbf{x}_p) \leq r_{\max} \\ r_{\min} & \text{if } s_p\phi(\mathbf{x}_p) < r_{\min} \end{cases} \tag{3.3}$$

where s_p is the sign of the particle (+1 for positive particles and -1 for negative particles) and $r_{\min}$ and $r_{\max}$ a minimum and maximum value, resp., by which the particles' radius is constrained. Enright et al. (2002a) suggest $r_{\min} = 0.1\Delta\tau$ and $r_{\max} = 0.5\Delta\tau$ with $\Delta\tau$ the grid spacing. Equation 3.3 ensures that the boundary of the particles keeps tangent to the surface whenever possible.

Updating the massless marker particles

Enright et al. (2004) point out that a first order accurate method for the time integration of the particles does not yield sufficiently accurate results. They propose a second order accurate Runge-Kutta midpoint rule for updating the interialess marker particles. Assume a particle at position $\mathbf{x}_p^t$ at time t, then its new position $\mathbf{x}_p^{t+\Delta t}$ at time $t + \Delta t$ is computed in two steps:

$$\begin{aligned} \mathbf{x}_{p,\,\text{tmp}} &= \mathbf{x}_p^t + \Delta t\mathbf{u}(\mathbf{x}_p^t), \\ \mathbf{x}_p^{t+\Delta t} &= \mathbf{x}_p^t + \Delta t\frac{\mathbf{u}(\mathbf{x}_p^t) + \mathbf{u}(\mathbf{x}_{p,\,\text{tmp}})}{2}. \end{aligned}$$

Note that solving the Navier-Stokes equations only provides velocities at the inside of the liquid surface. Hence, a priori, there are no velocities defined at the outside of the liquid surface to update the position of the (positive) particles.

Enright et al. (2002b) propose a velocity extrapolation method based on the following equation

$$\frac{\partial u}{\partial \tau} = -\mathbf{N}.\nabla u. \tag{3.4}$$

where τ is fictitious time. A similar equation holds for the v and w components of the velocity field. Equation (3.4) can be solved efficiently by means of a method based on the Fast Marching method. The latter is described in appendix D. For more details about the velocity extension method, the interested reader is referred to Adalsteinsson & Sethian (1999).

Updating the level set function

The level set function ϕ is updated by solving the level set equation (equation 3.1). Note that this equation has the same form as the advection term in the Navier-Stokes equation (equation 1.2) which was solved by means a semi-Lagrangian method (see section 2.3). Enright et al. (2004) showed that a fast first order accurate semi-Lagrangian method is also sufficient for the evolution of the level set function in case one uses massless marker particles as a diffusion correction technique. We refer the reader to appendix B for a detailed discussion of the semi-Lagrangian method.

Error correction

Particles are considered to have escaped their side of the surface if they are on the other side by more than their radius. The latter can be easily checked:

$$\begin{aligned} \phi(\mathbf{x}_p) < -r_p \text{ and } s_p > 0 \quad &\Rightarrow \quad \text{escaped positive particle} \\ \phi(\mathbf{x}_p) > r_p \text{ and } s_p < 0 \quad &\Rightarrow \quad \text{escaped negative particle} \end{aligned}$$

An escaped particle means that the level set function ϕ has suffered inaccuracies. Therefore, the escaped particles are used to correct ϕ locally as follows

1. Two new level set function, ϕ^+ and ϕ^-, are created which are initialized with the ϕ values.
2. An implicit function ϕ_p is associated with each escaped particle
$$\phi_p(\mathbf{x}) = s_p(r_p - |\mathbf{x} - \mathbf{x}_p|),$$
the zero level set of which corresponds to a sphere with radius r_p.
3. The ϕ_p values of escaped positive particles are calculated for the eight grid points on the boundary of the cell containing the particle. Each of these ϕ_p values is compared to the local value of ϕ^+ and the maximum is taken for the new local value of ϕ^+.
4. The previous step is repeated for all escaped negative particles but now the minimum of each of the ϕ_p values and the local value of ϕ^- is taken for the new local value of ϕ^-.
5. Finally, ϕ^+ and ϕ^- are merged back into a single level set by giving priority to values that are closer to the surface.
$$\phi = \begin{cases} \phi^+ & \text{if } |\phi^+| \leq |\phi^-| \\ \phi^- & \text{if } |\phi^+| > |\phi^-| \end{cases}$$
This results in a level set function corrected by means of the massless marker particles.

Reinitialization of the level set function

In general, updating the level set function does not yield a smooth signed distance function. Reinitialization refers to the idea of stopping a level set calculation at some point in time and rebuilding the level set function corresponding to the signed distance function (Sethian 1999). As mentioned before, there are several ways to enforce the signed distance property (equation 3.2). One method is the Fast Marching method described in appendix D. As the reinitialization of the level set function may cause the zero level set to move, which is not desirable, the particles are used to correct these errors as well.

Radii adjustment

Finally, the radii of the particles are adjusted to their new postion according to equation (3.3). Any particles which remain escaped have their radius set to $r_{\min}$, the minimum particle radius value.

Reseeding of the massless marker particles

The distribution of the particles around the surface can become very uneven during the course of the simulation. Therefore, a reseeding operation is required at fixed time intervals or according to some measure of surface stretching/compression. Enright et al. (2002a) propose the following reseeding strategy

```
procedure reseedingParticles()
  for each grid cell
    if the cell is far away from the surface
       delete all non-escaped particles
    else if the number of non-escaped particles is too low
       add particles
    else if the number of non-escaped particles is too high
       remove the particles furthest away from the surface
  end
end
```

Note that escaped particles are never deleted as they indicate errors in the current level set function.

3.4 Level set applications

As material form is ubiquitous, its design is of interest to many disciplines, including architecture, engineering, manufacturing, and medicine (Bloomenthal & Wyvill 1997). Level sets have gained popularity because they provide for a robust representation of geometry that can be dynamically evolved by solving partial differential equations in a manner that has been fine tuned by a community of researchers over a number of years. Implicit surfaces have several advantages compared to more traditional geometry modelling techniques (see section 3.2.1). For example, level sets easily allow to apply boolean operations and constructive solid geometry operations, which makes them a popular choice in computer aided design (CAD).

Level set methods have been applied to a wide variety of problems including fluid mechanics, computer vision, material science, and computer graphics. Examples include image enhancement and noise removal, shape detection and recognition, optimal path planning, reconstruction of surfaces from unorganized data points, etc.

Level set methods have become increasingly popular in computer graphics in the last few years for the simulation of natural phenomena. Foster & Fedkiw (2001) introduced level sets as a way to model and track a liquid surface while Nguyen et al. (2002) used a dynamic level set for the simulation of fire. The (Particle) Level Set Method has already been used in several feature films.

- the melting of a terminatrix in *Terminator 3: Rise of the Machines (2003)*,
- the wine falling through the skeleton of a pirate in *Pirates of the Caribbean: The Curse of the Black Pearl (2003)*,
- the animation of water pouring off a magical ship in *Harry Potter and the Goblet of Fire (2005)*,
- the ocean simulation in *Poseidon (2006)*.

Part II

Development of a liquid simulation library

Chapter 4

The FluidLib library

4.1 Introduction

The aim of this Master project is twofold:

1. researching the mathematical concepts and algorithms which are used in the visual effects industry to simulate fluid behaviour, and
2. developing a high-quality liquid simulation library.

The research part involves

- the study of the Navier-Stokes equations which govern the motion of liquids as well as numerical techniques to solve these equations efficiently, and
- the study of the theory behind implicit surface representations and level set methods to evolve dynamic implicit surfaces.

This was the subject of the first three chapters.

This chapter describes the second part of this Master project: the design and implementation of a liquid simulation library. The goal was to develop a versatile library which has the flexibility to

- simulate liquids in 2D or 3D,
- use either implicit or explicit integration schemes to solve the Navier-Stokes equations, and

- use either massless marker particles or a level set as representation for the liquid.

C++ was chosen as the programming environment. OpenGL was used to render the 2D simulations while Renderman was adopted for the simulations in 3D.

The structure of this chapter is as follows: Section 4.2 gives a brief overview of the design of the liquid simulation library and how to use it. Next, an overview is given of the most important implementation details, the problems encountered during the development of the C++ library, and the solution strategies which were tried out (section 4.3). Finally, section 4.4 is devoted to the techniques which were applied to reduce the time and memory cost of the simulations.

4.2 Structure of the FluidLib library

4.2.1 From the inside looking out

The main class in the FluidLib library is the class `Sim` which is responsible for

- reading the inputfile which contains the simulation parameters specified by the user,
- creating all other class objects and linking them to each other,
- time management,
- updating the simulation by one time step in each loop, and
- writing the results out to files.

An abstract `SolverBase` class was implemented which serves as a base class for the `slowSolverLiquid` and `fastSolverLiquid` classes. The `slowSolverLiquid` class implements the explicit method based on the work by Foster & Metaxas (1996) (see section 1.4.5). The `fastSolverLiquid` class implements a significantly more efficient Navier-Stokes solver by computing the advection term with the semi-Lagrangian method (see section 2.3), the diffusion term with an implicit integration scheme (see section 2.4), and the pressure term with an efficient conjugate gradient solver (see section 2.5).

The `LevelSet` class keeps the grid with the level set values and is responsible for all the level set operations such as (re)initialization, updating its position, turning the level set into a signed distance function, error correction, etc. (see section 3.3.2). This class

contains an object of the `Polygonizer` class which implements the marching tetrahedra algorithm and writes the resulting triangular mesh to an outputfile.

We list the other classes, which are mainly used to help in the implementation of the classes mentioned above, with a brief description below.

- the `Particle` class implements a massless marker particle as used in the PLS method (see section 3.3.2),
- the `Vector` class encapsulates a 3D point or vector,
- the `Field` class is a template class storing the information of a 3D grid,
- the `VectorField` class is a subclass of the Field class, which stores the data of a vector field
- the `ParticleManager` class manages the massless marker particles when they are used to track the liquid volume and visualize the liquid,
- the `KdTree` class encapsulates a kd-tree abstract datatype,
- the `heap` class implements a heap data structure with the heap property that each node is not greater than its children,
- the `maxHeap` class implements a heap data structure with the heap property that each node is not smaller than its children,
- the `SparseMatrix` class which encapsulates an efficient data structure for a sparse matrix,
- the `SparseRealArray` class implements an efficient data structure for a vector which can be multiplied with an object of the `SparseMatrix` class,

Doxygen documentation files were enclosed on the accompanying CD to provide the reader with detailed information about these different classes, their data members and member functions.

4.2.2 From the outside looking in

It is straightforward to carry out simulations with the FluidLib library. One only needs to specify the desired simulation parameters in a file and write a simple C++ program as given below.

```
#include "FluidLib.h"

using namespace FluidLib;

int main()
{
  Sim sim("name of inputfile");
  sim.start();
  return 0;
}
```

The important elements of the above code are

- the inclusion of the file “FluidLib.h”, the library’s master header file which automatically includes all the header files of the FluidLib library,
- informing the compiler that the namespace FluidLib is used (the whole library implementation is part of the namespace FluidLib),
- the declaration of an object of the `Sim` class and providing the constructor with the name of the inputfile which contains the user-definable simulation parameters,
- the call of the member function “start” of the `Sim` object to start the simulation.

The inputfile allows the reader to specify several simulation parameters such as the size of the computational domain, the grid resolution, the type of boundary conditions, the time stepsize, initial values for the velocity and the pressure, the length of the simulation, the initial number of particles per cell, the prefixname of the outputfiles, the viscosity and density of the liquid, and many others. A documented inputfile is included on the accompanying CD, which should allow the reader to easily specify a customized inputfile.

4.3 Implementation details

4.3.1 Extending the liquid solver engine from 2D to 3D

Setting the surface boundary conditions in 3D

Testing the 3D liquid simulation engine before the implementation of the level set representation of the liquid surface, requires the specification of the velocity boundary

conditions at the surface. However, there is a lack of information in the literature on setting the surface boundary conditions in 3D (see section 2.2).

Therefore, I extended the method proposed by Griebel et al. (1998) for liquid simulations in 2D. The result is a four-step strategy to set the surface boundary conditions and the relevant velocity values outside the liquid as given below.

For each surface cell:

1. *for all* faces bordering to an empty cell:

   set the velocity value as described by Carlson (2004).

2. *for all* empty cells sharing a face with the surface cell:

   if this face is shared with another empty cell
   and if this face is not at the outside of the 3×3 cube surrounding the surface cell
   set the value at the left, back and lower face of the current empty cell

3. *for all* empty cells sharing at least two faces with empty cells considered in the previous step (step 2):

   if this face is shared with another empty cell
   and if this face is not at the outside of the 3×3 cube surrounding the surface cell
   set the value at the left, back and lower face of the current empty cell

4. *for all* empty cells sharing at least two faces with empty cells considered in the previous step (step 3):

   if this face is shared with another empty cell
   and if this face is not at the outside of the 3×3 cube surrounding the surface cell
   set the value at the left, back and lower face of the current empty cell

In step 2, 3 and 4 the velocity component values at the left, back and lower face of the empty cells surrounding the surface cell are set to the nearest corresponding velocity component value of the current surface cell. This value is set in step 1 or is known from updating the Navier-Stokes equations. Several experiments have shown that this strategy is successful.

A faster and stable Navier-Stokes solver

Section 2.4 described an implicit integration scheme to solve the diffusion term. In practice, this method requires the efficient computation of a system of linear equations. Note that the size of the corresponding matrix A can potentially be very large as both

the number of rows and columns equals the number of cell faces which are shared by two fluid cells. Moreover, A is a very sparse matrix, which means that most of its elements are equal to zero. Therefore, special data structures are required which easily allow to store sparse matrices and perform classic matrix operations (such as e.g. multiplication with a vector) in an efficient way. The linear system was solved with an efficient conjugate gradient technique as described by Golub & Van Loan (1983). Note that in fact three linear systems have to be solved, one for each component of the velocity.

The computation of the pressure also requires solving a linear system (section 2.5). The corresponding matrix A is again sparse and very large as each fluid cell in the computational domain has an entry in every row (and column) of A. The sparse matrix classes mentioned above were designed in a generic way so that they can be reused for the computation of the Poisson equation.

4.3.2 Implicit surfaces and the PLS method

The implementation of the PLS method

The level set function is stored on a high-resolution subgrid of the Navier-Stokes grid. The code was written in such a way that the resolution of the level set grid can be increased by the user without the need to increase the resolution of the Navier-Stokes grid.

As explained in section 3.3.2, the initial level set function can be constructed at each grid point by means of the closest massless marker particle. Given the large amount of massless marker particles, a data structure is needed which stores the massless marker particles and allows for an efficient nearest neighbour search. A kd-tree was chosen for this purpose. We refer the reader to appendix D for more information about nearest neighbour search and our implementation of a kd-tree data type.

Both the calculation of the extension velocities and the reinitialization of the level set function are carried out by means of a Fast Marching method (section 3.3.2). In the inner loop of this method, one requires to find the point with the smallest T value (see appendix D). A heap data structure was used, as suggested by Sethian (1996), to carry out this operation efficiently. A heap abstract data type was implemented from scratch as the heap provided by the C++ Standard Template Library (STL) was inadequate for our purposes here.[1]

One of the main advantages of using a zero level set to represent the liquid surface is that it easily allows the merging and breaking up of liquid. However, it should be noted that the positive particles which are placed at the outside of the liquid surface in the PLS

[1] The combined data type of a heap with indirect pointers was required.

method can prevent two liquid drops from merging together properly. Although this issue is not mentioned in the original paper by Enright et al. (2002a), the author found two other references to such a problem. Rasmussen et al. (2004) suggest a technique which makes the extension velocities divergence-free (i.e. mass conserving) so that the positive particles are "pushed away" when two liquid interfaces are about to merge together. Bargteil et al. (2006), on the other hand, suggest to subdivide the grid points into two groups depending on whether they trace back to points inside or outside the surface during the semi-Lagrangian update of the level set function. By enforcing that their polygonizer never creates a surface between two grid points of the same group, they were able to handle the topology change.

The author implemented a technique which is inspired by the latter method. More specifically, positive particles are deleted immediately after the semi-Lagrangian level set update if all the surrounding grid nodes are inside the liquid. This approach has proven to be successful as can be seen from the included demos.

The tessellation of the implicit surface

The author only knows of two published polygonization implementations: a Marching Cubes exhaustive search method given by Watt & Watt (1991) and a Marching Polyhedra continuation method given by Heckbert (1994). For reasons stated in sections 3.2.3 and 3.2.4, we desire an exhaustive search technique and a tetrahedron as polygonizing cell. Therefore, the author studied both implementations, adopted the useful components of each of them, and combined them into an efficient Marching Polyhedra exhaustive search method which was integrated in the liquid simulation library. The code was made efficient by adopting the points-polygons format to store the polygonal mesh and by using the signed distance property of the level set function.

Initially, each cubical cell was subdivided into 6 tetrahedra as suggested by Payne & Toga (1990). However, as this subdivision is not symmetrical, this led to small artifacts at sharp corners. Therefore, the same approach as Carlson (2004) was taken to subdivide each cube into 24 tetrahedra.

The visualization of the implicit surface

Although photorealism is not the aim of this project, a Renderman shader was written by the author to give the liquid a more interesting look. Light reflection and refraction is modelled by rendering the liquid with ray tracing. The Fresnel equations are used to determine the reflection and transmission coefficients.

4.4 Time and data storage issues

A lot of effort has been done to speed up the liquid simulation library. All the optimizing strategies discussed so far were focussed on the efficiency of the numerical techniques and algorithms. In this section, we briefly discuss the techniques which were implemented to speed up the I/O operations and reduce the memory cost associated with the storage of the simulation results.

The memory and CPU requirements for managing the liquid representation in three dimensions is rather high. In case one adopts massless marker particles to visualize the liquid, each grid cell should contain 27 or even 64 particles in the initial configuration rather than 9 or 16, as in the two-dimensional case. In case of a level set representaton, high-resolution 3D grids have to be stored. Therefore, several additional strategies have to be implemented to create an efficient liquid simulation library.

The author considered the following optimizing methods:

1. Reducing the accuracy with which the simulation data are stored to reduce the memory cost.
2. Changing the outputfiles to a binary format to speed up the I/O operations.
3. Experimenting with compression techniques to find a good trade-off between saving memory and the additional overhead of compressing and uncompressing files.
4. Only storing and visualizing the particle at the surface for testing purposes.

Conclusion

This Master project involves several components:

- a thorough study of the Navier-Stokes equations and numerical techniques to solve them efficiently,
- an extensive literature study of dynamic level set representations and level set methods,
- the design and development of a liquid simulation library, which includes the implementation of the following techniques:
  - an implicit integration scheme to solve the diffusion term more efficiently for high viscosity liquids (Carlson et al. 2002),
  - a conjugate gradient method to carry out the pressure projection (Carlson 2004),
  - the Vorticity Confinement method to inject rotational energy lost by the coarse simulation grids (Fedkiw et al. 2001),
  - the Fast Marching method to make level set functions smoother (Sethian 1996),
  - a technique to extend the velocities outside the liquid surface (Adalsteinsson & Sethian 1999),
  - the level set method to represent and track the liquid surface (Foster & Fedkiw 2001),
  - the PLS method (Enright et al. 2002a,b, 2004), an improved variant of the level set method,
  - the marching tetrahedra method (Payne & Toga 1990) to extract a polygonal representation of the liquid surface,
  - a Renderman shader including ray tracing, the Fresnel equations, and anti-aliasing techniques,

– several abstract data types such as a kd-tree, heap, sparse matrix class, etc.

This work resulted in a sophisticated C++ library to efficiently simulate liquids in 2D and 3D. Several demos have been created which illustrate the main techniques implemented in the liquid simulation library.

There are many ways in which this work can be extended or improved.

1. Functionalities of the simulation library
   - interaction between liquids and moving objects such as rigid bodies (see e.g. Carlson et al. 2004), animated characters (e.g. Foster & Fedkiw 2001), etc.
   - interaction between multiple liquids (Losasso et al. 2006),
   - simulation of other natural phenomena such as e.g. smoke and explosions,
   - adding surface tension effects,
   - extending the liquid solver to medium-scaled problems (Irving et al. 2006),
2. Visualization of the liquid surface
   - implementation of regularized marching tetrahedra (Treece et al. 1998), a method which combines marching tetrahedra and vertex clustering to generate isosurfaces which are topologically consistent with the data and contain typically 70% less triangles than marching tetrahedra with significantly improved aspect ratios. This improvement in aspect ratio greatly enhances the display of the surface.
   - alternatively, a direct rendering method such as ray tracing may be implemented. One can adopt more sophisticated ray tracing methods which takes all the different light paths in the environment into account (e.g. Monte Carlo ray tracing (see e.g. Dutre et al. (2002)) or the Photon Mapping method (Jensen 1996)). These physically-based rendering methods ensure that the photorealism of the simulation carries over to the final images.
   - advecting texture coordinates
3. Further work on improving the efficiency of the FluidLib library
   - implementing an octree data structure which allows to increase the resolution only at the surface,
   - investigate the possibilities to parallellize the computations
4. Further work on the control aspects

- allowing the specification of curves to control the liquid motion similar to the way camera paths are specified by a 3D curve (Foster & Fedkiw 2001),
- adding velocity fields which can be blended with the velocity field obtained from solving the Navier-Stokes equations,
- integrating the library into an application framework such as Maya.

Appendix A

Passive advection through a vector field

Assume a scalar field $\psi(\mathbf{x}, t)$ which may for example represent the density or temperature of the liquid. We derive the equation which describes the passive advection of ψ through the velocity field **u**.

Imagine that ψ is carried by a small particle moving passively in the velocity field **u**. Let the path of the particle be described by

$$\mathbf{x} = s(t) = (x(t), y(t), z(t)).$$

This path defines a curve which is also known as a characteristic curve, or simply a characteristic. Since this curve is determined by the velocity field, the velocity is tangent to the characteristic.

$$\mathbf{u} \equiv (u(t), v(t), w(t)) = (x'(t), y'(t), z'(t)). \tag{A.1}$$

Because the curve $s(t)$ was defined as the path of a passively advected particle carrying a certain amount of ψ, ψ is a constant along the curve:

$$\psi(s(t), t) = \psi(x(t), y(t), z(t), t) = \psi_0. \tag{A.2}$$

In order to derive a relationship between the scalar field ψ and the velocity field **u**, we differentiate equation (A.2) with respect to time

$$\frac{d}{dt}(\ \psi(x(t), y(t), z(t), t)\ ) = 0$$

Using the chain rule, equation (A.1) and the definition of the gradient operator, we obtain

$$\begin{aligned}
\frac{\partial \psi}{\partial t} + \frac{\partial \psi}{\partial x} \cdot x'(t) + \frac{\partial \psi}{\partial y} \cdot y'(t) + \frac{\partial \psi}{\partial z} \cdot z'(t) &= 0 \\
\frac{\partial \psi}{\partial t} + \frac{\partial \psi}{\partial x} \cdot u(t) + \frac{\partial \psi}{\partial y} \cdot v(t) + \frac{\partial \psi}{\partial z} \cdot w(t) &= 0 \\
\frac{\partial \psi}{\partial t} + \mathbf{u} \cdot \nabla \psi &= 0 \qquad \text{(A.3)}
\end{aligned}$$

Equation (A.3) is a linear first-order PDE which describes the passive advection of ψ through the velocity field **u**.

Appendix B

A first-order Semi-Lagrangian method

This appendix describes a first-order semi-Lagrangian method which can be used to solve the following equation

$$\frac{\partial \psi}{\partial t} = -\mathbf{u} \cdot \nabla \psi.$$

This equation describes the passive advection of a scalar quantity ψ through a vector field $\mathbf{u}$ and was derived in appendix A. The reader is encouraged to read this appendix first as it provides the necessary intuition to understand the way Semi-Lagrangian methods solve this equation.

Given that the scalar field $\psi(\mathbf{x}, t)$ is known at time t, the value of $\psi(\mathbf{x}, t + \Delta t)$ at any point $\mathbf{x}$ at time $t + \Delta t$ can be computed by means of the following three steps:

1. find the characteristic curve passing through $(\mathbf{x}, t + \Delta t)$[1],
2. follow it backward to some previous point $(\mathbf{x}_0, t)$ where the value of ψ is known, and
3. set $\psi(\mathbf{x}, t + \Delta t) = \psi(\mathbf{x}_0, t)$.

Courant, Isaacson, and Rees (1952) developed the first semi-Lagrangian scheme: the backward characteristic or CIR scheme. CIR approximates the backward characteristic through $\mathbf{x}$ at time $t + \Delta t$ by a straight line to find the point $\mathbf{x}_0$

$$\mathbf{x}_0 \approx \mathbf{x} - \Delta t\, \mathbf{u}(\mathbf{x}, t). \tag{B.1}$$

Subsequently, the value of ψ is interpolated at time t to this point $\mathbf{x}_0$ and this value is assigned to $\psi(\mathbf{x}, t + \Delta t)$.

[1] This characteristic curve is unique in the absence of shockwaves.

The pseudo-code of this first-order semi-Lagrangian method is given below.

```
procedure advectScalarField()
  for each voxel in grid at time t + Δt
     1. back-trace the voxel position to time t
        using the velocity at time t
     2. lookup the six scalar field values in the grid
        at time $t$ at neighbouring voxels of the
        back-traced voxel position
     3. trilinear interpolate these six neigh-
        bouring scalar field values
     4. assign the result of the interpolation
        to the current voxel at time t + Δt
  end
end
```

The important property of semi-Lagrangian methods is that they are unconditionally stable: no matter how big the time step the simulation will not blow up. It is easy to understand why this technique is unconditionally stable; since the scalar ψ values computed at a particular time step are a linear interpolation of the ψ values calculated in the previous time step, the maximum of the new ψ values is always bounded by the maximum of the old values. One can easily verify this in one dimension:

$$\begin{aligned} \psi_{\text{new,interp}} &= (1-s)\,\psi_{\text{old},0} + s\,\psi_{\text{old},1} \\ &\leq (1-s)\,\psi_{\text{old,max}} + s\,\psi_{\text{old,max}} \\ &= \psi_{\text{old,max}} \end{aligned}$$

This idea extends naturally to more dimensions. So the ψ values are always bounded no matter how big the time step and therefore will never blow up and become unstable.

Because of this unconditional stability, semi-Lagrangian techniques have been frequently used in computer animation (see e.g. Stam (1999), Foster & Fedkiw (2001), Enright et al. (2002b), Losasso et al. (2004), Carlson et al. (2004), Bargteil et al. (2006) and many others) as well as in other disciplines, such as atmospheric sciences in which large time steps are desired to model large scale flows (see e.g. Staniforth & Cote (1991) for a review).

Appendix C

Nearest Neighbour Search

Assume that a set P of n points in $\mathbb{R}^3$ is given and that our goal is a data structure for P which allows to efficiently compute the point in P which is closest to a given location $\mathbf{x} \in \mathbb{R}^3$. Simple data structures such as arrays and lists are not appropriate as the nearest neighbour search operation is far too costly.

An octree is a tree data structure which is often used to partition a 3D space by recursively subdividing it into eight subspaces (octants). A search for the nearest neighbour is simply a matter of

1. finding the leaf node which represents the subspace to which **x** belongs, and
2. examining the points of P which belong to that subspace and perhaps the neighbouring subspaces.

Although versatile and easy to implement, an octree has the disadvantage that a large number of empty subspaces occur in the case that the points in P are non-uniformly distributed in 3D space. This makes the nearest neighbour search operation less efficient.

A data structure that is much better at handling a non-uniform distribution of points is the (three-dimensional) kd-tree (Bentley 1975). This is a binary search tree in which each node is used to partition one of the dimensions. A kd-tree can be considered as a special case of a BSP-tree (Sung & Shirley 1992) where the partitioning planes cut one of the three dimensions (x, y, or z) into two pieces. Each node in a kd-tree contains

- one point from P,
- an axis-orthogonal plane that contains this point [1],

[1] The axis-orthogonal plane is undefined for leaf nodes.

- pointers to the left and right subtrees.

All points in the left (right) subtree are below (above) this partitioning plane.

The nearest neighbour algorithm searches depth first in a kd-tree, and uses the heuristic of searching first the child node which contains the target. The maximum radius (from **x**) in which any possible closer point could lie is initially set to infinity and is adapted during the search. If a point is found which lies closer to **x**, this radius is set to the distance between this point and **x**. This mechanism allows to prune the search tree effectively: if a node represents a subspace which lies completely outside this radius, it is not necessary to examine this node and all its descendants.

Care should be taken that a well-balanced kd-tree is constructed as a skewed tree does not allow effective tree pruning (and hence, considerable time savings). This can be accomplished by means of the following algorithm (Jensen 2001):

```
kdTree* balance( set P){
   Find the bounding box of P
   Select the dimension in which the bounding box is largest
   Find the median of the points in that dimension
   S1 = set containing all points below the median
   S2 = set containing all points above the median
   node = median
   node.left = balance (S1)
   node.right = balance (S2)
   return node
}
```

An efficient median search algorithm is described by Sedgewick (1992).

This balancing algorithm results in a left-balanced kd-tree which can be represented very compactly by means of a heap data structure. In this structure it is not necessary to store pointer to the subtrees explicitly: the array element at index 1 is the root, and element i has element $2i$ as left child and element $2i + 1$ as the right child. Not storing pointers can lead to substantial savings when the set P is large.

Appendix D

The Fast Marching method

Level Set methods are numerical techniques for evolving implicit surfaces. They solve the level set equation

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = 0.$$

(See section 3.3 for more details.) This is an initial value formulation as the initial position of the implicit surface gives initial data for a time-dependent problem.

In the special case that the underlying velocity field never changes sign, so that the surface is always moving forward or backward, one can turn this time-dependent problem into a stationary problem and use a faster method, the Fast Marching method, introduced by Sethian (1996). This appendix aims to give an intuition for the stationary approach and the idea behind the Fast Marching method.

Consider the example of a circular implicit function which is expanding with a constant speed in 2D and assume that a grid is laid down on top of the problem. Define the arrival time function $T(x, y)$ as the time when the surface crosses the position (x, y). This function is single-valued as the velocity field was assumed to never change sign. The graph of the arrival time function in our example has a cone-like shape. This surface has the nice property that its t level set

$$\{(x, y) \in \mathbb{R}^2 | T(x, y) = t\}$$

gives the position of the circle at time t. Hence, we have exchanged our initial problem of evolving the circle for the stationary problem of constructing the arrival time function T. This is called the boundary value formulation as the initial position of the moving surface is the boundary (zero level set) for the arrival time surface T(x,y) which we are looking for.

The equation for this arrival time function is easily derived. Assume an implicit surface which moves in a direction normal to itself with a known speed function F. In one dimension, we have that

$$F = \frac{dx}{dT},$$

or equivalently,

$$F\frac{dT}{dx} = 1.$$

In multiple dimensions, ∇T is orthogonal to the level sets of T and, similar to the one-dimensional case,

$$F|\nabla T| = 1. \tag{D.1}$$

The Fast Marching method allows the efficient computation of the arrival time function at all grid points. It is based on the observation that if the underlying velocity field never changes sign, information propagates in only one direction away from the surface. This implies that if we calculate the T values in the correct order, we only need to visit each grid point once.

In more detail, all the grid points which do not have to be updated are tagged as "known" while all the others are tagged as "far". Subsequently, all the far grid points neighbouring to a known grid point are tagged as "close". For example, in the case of an expanding circle, the grid points inside or on the curve are tagged as known, and all the grid points outside the circle are tagged as "far" except for the ones who are less than one grid spacing away from the circle. The latter grid points are tagged as "close".

After this initialization step, the following steps are carried out until there are no close grid points left anymore.

1. Let **p** be the close point with the smallest T value. Tag this point as "known",
2. tag all neighbouring grid points of **p** which are not known as close,
3. recompute the T values at all close neighbours of **p** by means of a properly discretized version of equation (D.1).

Sethian (1996) proofs that this algorithm yields the arrival time function of ψ. More background information about Fast Marching methods can be found in Sethian (1999).

References

Adalsteinsson, D. & Sethian, J. A. 1999, J. Comput. Phys., 148, 2

Anderson, J. D. 1995, Computational Fluid Dynamics. The basics with applications (New York: McGraw-Hill, Inc.)

Bargteil, A. W., Goktekin, T. G., O'brien, J. F., & Strain, J. A. 2006, ACM Trans. Graph., 25, 19

Bentley, J. L. 1975, Commun. ACM, 18, 509

Bloomenthal, J. 1988, Comput. Aided Geom. Des., 5, 341

Bloomenthal, J. & Wyvill, B., eds. 1997, Introduction to Implicit Surfaces (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.)

Carlson, M., Mucha, P. J., R. Brooks Van Horn, I., & Turk, G. 2002, in SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation (New York, NY, USA: ACM Press), 167–174

Carlson, M., Mucha, P. J., & Turk, G. 2004, ACM Trans. Graph., 23, 377

Carlson, M. T. 2004, PhD thesis, Georgia Institute of Technology

Chorin, A. J. 1968, Math. Comp., 22, 745

Clavet, S., Beaudoin, P., & Poulin, P. 2005, in SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation (New York, NY, USA: ACM Press), 219–228

Deusen, O., Ebert, D. S., Fedkiw, R., et al. 2004, in GRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes (New York, NY, USA: ACM Press), 32

Dutre, P., Bala, K., & Bekaert, P. 2002, Advanced Global Illumination (Natick, MA, USA: A. K. Peters, Ltd.)

Enright, D., Fedkiw, R., Ferziger, J., & Mitchell, I. 2002a, J. Comput. Phys., 183, 83

Enright, D., Losasso, F., & Fedkiw, R. 2004, A fast and accurate semi-Lagrangian particle level set method

Enright, D., Marschner, S., & Fedkiw, R. 2002b, in SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques (New York, NY, USA: ACM Press), 736–744

Fedkiw, R., Stam, J., & Jensen, H. W. 2001, in SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques (New York, NY, USA: ACM Press), 15–22

Feldman, B. E., O'Brien, J. F., & Arikan, O. 2003, ACM Trans. Graph., 22, 708

Foster, N. & Fedkiw, R. 2001, in SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques (New York, NY, USA: ACM Press), 23–30

Foster, N. & Metaxas, D. 1996, Graph. Models Image Process., 58, 471

Foster, N. & Metaxas, D. 1997, in SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques (New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.), 181–188

Golub, G. H. & Van Loan, C. F. 1983, Matrix Computations (Baltimore, Maryland, USA: The Johns Hopkins University Press)

Griebel, M., Dornseifer, T., & Neunhoeffer, T. 1998, Numerical simulation in fluid dynamics: a practical introduction (Philadelphia, PA, USA: Society for Industrial and Applied Mathematics)

Harlow, F. H. & Welch, J. E. 1965, Physics of Fluids, 8, 2182

Heckbert, P. S., ed. 1994, Graphics gems IV (San Diego, CA, USA: Academic Press Professional, Inc.)

Jensen, H. W. 1996, in Proceedings of the eurographics workshop on Rendering techniques '96 (London, UK: Springer-Verlag), 21–30

Jensen, H. W. 2001, Realistic image synthesis using photon mapping (Natick, MA, USA: A. K. Peters, Ltd.)

Lorensen, W. E. & Cline, H. E. 1987, in SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques (New York, NY, USA: ACM Press), 163–169

Losasso, F., Gibou, F., & Fedkiw, R. 2004, ACM Trans. Graph., 23, 457

Müller, M., Charypar, D., & Gross, M. 2003, in SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation (Aire-la-Ville, Switzerland, Switzerland: Eurographics Association), 154–159

Nguyen, D. Q., Fedkiw, R., & Jensen, H. W. 2002, in SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques (New York, NY, USA: ACM Press), 721–728

Osher, S. & Fedkiw, R. 2002, Level Set Methods and Dynamic Implicit Surfaces (New York, USA: Springer-Verlag)

Osher, S. & Sethian, J. A. 1988, J. Comput. Phys., 79, 12

Payne, B. A. & Toga, A. W. 1990, IEEE Comput. Graph. Appl., 10, 33

Premoze, S., Tasdizen, T., Bigler, J., Lefohn, A., & Whitaker, R. 2003, Particlebased simulation of fluids

Sedgewick, R. 1992, Algorithms in C++ (Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.)

Sethian, J. 1996, in Proc. Nat. Acad. Sci., Vol. 93, 1591–1595

Sethian, J. A. 1999, Level Set Methods and Fast Marching Methods (Cambridge University Press)

Stam, J. 1999, in SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques (New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.), 121–128

Staniforth, A. & Cote, J. 1991, Monthly Weather Review, 119, 2206

Steinhoff, J. & Underhill, D. 1994, Physics of Fluids, 6, 2738

Sung, K. & Shirley, P. 1992, Graphics Gems III, 271

Temam, R. 1969, Arch. Rational Mech. Anal, 32, 135

Treece, G. M., Prager, R. W., & Gee, A. H. 1998, Regularised marching tetrahedra: improving iso-surface extraction, Tech. Rep. CUED/F-INFENG/TR 333, Cambridge University, Cambridge, England

Tryggvason, G., Bunner, B., Esmaeeli, A., et al. 2001, J. Comput. Phys., 169, 708

Watt, A. & Watt, M. 1991, Advanced animation and rendering techniques (New York, NY, USA: ACM Press)