Fluid Flow for the Rest of Us: Tutorial of the Marker and Cell Method in Computer Graphics

David Cline David Cardon Parris K. Egbert *
Brigham Young University

Abstract

Understanding how fluid is modeled for computer graphics can be a challenge. This is especially true for students who have not taken courses in vector calculus and differential equations, or are rusty in these subjects. Beginning students tend to get bogged down in the notation of the Navier-Stokes equations and the inevitable difficulties that arise when trying to discretize them. Despite this fact, we will attempt to show that, given a little instruction, building a complete fluid simulator is actually fairly straightforward.

1 Introduction

In the fall of 2001 I attempted to implement a fluid simulator, with only limited success. In retrospect, I can see that there were really two reasons for my lack of success. First, my math background had not prepared me for the notation of the Navier-Stokes equations, and second, the implementation details of the fluid flow systems described in the literature were scattered over a lot of "previous work".

A couple of years later, in the Winter of 2004, a friend of mine, Dave Cardon, implemented a liquid simulator as a class project. While he was better prepared for the task than I was, he still found it difficult to ferret out all of the details needed to implement a fluid simulator.

After both of our experiences, we realized that implementing a fluid simulator is really not that difficult once all the details have been tracked down. It was understanding the notation and tracking down the details that was the hard part. At that point, we decided to write what we wish we'd had to begin with, a detailed tutorial describing both the Navier-Stokes equations, and some of the most recent advances in fluid simulation for computer graphics. To that end, we have written this paper, which is organized as follows: First, the mathematical notation used by the Navier-Stokes equations is described. Then the Navier-Stokes equations are presented, giving what we hope is an intuitive explanation of each of the terms. Next, we describe a fluid simulator similar to the one described in [Foster and Fedkiw 2001]. After the basic simulator has been described, we show how to extend it to achieve different effects, including making the fluid respond to moving objects as in [Foster and Fedkiw 2001], the simulation of fluids with variable viscosity [Carlson et al. 2002], viscoelastic fluid simulation [Goktekin et al. 2004], and the simulation of smoke [Fedkiw et al. 2001]. Finally, we discuss how to track the surface of a fluid using *level sets*.

2 Understanding the Navier-Stokes Equations

2.1 Fluid Flow as a Velocity Field

In this paper we model fluid flow as a *velocity field*, a vector field that defines the motion of a fluid at a set of points in space. Throughout our discussions, we will use the term $\mathbf{u}$ to refer to this velocity field. To simulate a flowing fluid, we evolve $\mathbf{u}$ over time, and move marker particles (or some other representation of the fluid, such as a level set) through space as dictated by $\mathbf{u}$. Since we cannot store velocity information at every point in space, velocities are stored at discrete grid points, and velocities between these samples are found by interpolation. The rules used to evolve $\mathbf{u}$ are based on the Navier-Stokes equations, which will be presented in section 2.3.

2.2 Mathematical Notation

In this section, we introduce the mathematical notation used to define the Navier-Stokes equations. In our descriptions we will use **bold** to indicate vector quantities, and *itallics* to indicate scalar quantities. For each term, we will try to give an intuitive explanation of what means, and then explain how we evaluate it numerically in our application. Although the terms described in this section may appear challenging, keep in mind that in every case they boil down to adding or subtracting a few values from a small neighborhood in the grid.

The partial derivative (∂). The derivative of a function of one variable $f(x)$ can be written $\frac{df}{dx}$. If a function is defined over multiple variables, such as $g(x,y,z)$, it has a *partial derivative*, or simply *partial*, for each variable over which the function is defined. To distinguish the partial from the standard derivative, a stylized "d" is used in the notation. For example, the partial of g with respect to y is written $\frac{\partial g}{\partial y}$, and is defined as the limit of a *central difference*:

$$\frac{\partial g(x,y,z)}{\partial y} = \lim_{h \to 0} \frac{g(x,\ y+h,\ z) - g(x,\ y-h,\ z)}{2h}$$

This is almost identical to the definition of the derivative found in any Calculus text. One way to evaluate the partial on a discrete grid is to drop the limit from the above expression and use an h of one grid cell width. The denominator of the expression can also be ignored for our application, leaving the central difference

$$\frac{\partial g(x,y,z)}{\partial y} = g(x,\ y+1,\ z) - g(x,\ y-1,\ z) \qquad (1)$$

Besides the central difference, it is sometimes convenient to use a *forward* or a *backward* difference to evaluate the partial:

$$\frac{\partial g(x,y,z)}{\partial y} = g(x,\ y+1,\ z) - g(x,\ y,\ z) \qquad (2)$$

$$\frac{\partial g(x,y,z)}{\partial y} = g(x,\ y,\ z) - g(x,\ y-1,\ z) \qquad (3)$$

Sometimes we want to take the partial of one of the components of a vector-valued function. We will use subscripts to denote the different components of the function. For example, $\frac{\partial u_x}{\partial y} = u_x(x,y+1,z) - u_x(x,y-1,z)$ is the partial derivative of the

*cline@rivit.cs.byu.edu, the_dave@byu.edu, egbert@cs.byu.edu

x component of $\mathbf{u}$ in the y direction evaluated using central differences.

The gradient operator (∇)**.** The gradient operator is a vector of partial derivatives. In three dimensions, it is defined

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

The gradient of a scalar field, ∇g, is a vector field that tells how the scalar field changes spatially. In our application, we will calculate the gradient of the pressure field using *backward* differences:

$$\begin{aligned} \nabla g(x,y,z) = (\ & g(x,y,z) - g(x-1,y,z), \\ & g(x,y,z) - g(x,y-1,z), \\ & g(x,y,z) - g(x,y,z-1)\ ) \end{aligned} \quad (4)$$

The divergence of a vector field $(\nabla\cdot)$**.** Since the gradient operator is just a vector of partial derivatives, it can be used in a dot product with a vector field. For instance, consider the vector field $\mathbf{u}$. The dot product of the gradient operator with the vector field $\mathbf{u}$ produces the scalar field

$$\nabla \cdot \mathbf{u} = \frac{\partial u_x}{\partial x} + \frac{\partial u_y}{\partial y} + \frac{\partial u_z}{\partial z}$$

where u_x, u_y and u_z are the x, y and z components of $\mathbf{u}$. This use of the gradient operator is called the *divergence* of a vector field because it describes the net flow out of or into points in the vector field. Rather than finding the divergence of points within the vector field, however, we will use *forward* differences to find the net flow out of or into a small cube within the vector field:

$$\begin{aligned} \nabla \cdot \mathbf{u}(x,y,z) = \ & (u_x(x+1,y,z) - u_x(x,y,z)) + \\ & (u_y(x,y+1,z) - u_y(x,y,z)) + \\ & (u_z(x,y,z+1) - u_z(x,y,z)) \end{aligned} \quad (5)$$

The Laplacian operator, (∇^2)**.** The Laplacian operator is the dot product of two gradient operators. This dot product produces a scalar quantity which is the sum of the x, y and z second partials (the derivatives of the derivatives):

$$\nabla^2 = \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}.$$

Applying ∇^2 to a scalar field produces another scalar field that roughly describes how much values in the original field differ from their neighborhood average. This property becomes readily apparent when we look at the terms of equation 6. To evaluate the laplacian, we subtract a *backward difference* from a *forward difference* at the point of interest. This formulation keeps the calculation within a neighborhood of three grid cells:

$$\begin{aligned} \nabla^2 g(x,y,z) = \ & g(x+1,y,z)\ +\ g(x-1,y,z)\ + \\ & g(x,y+1,z)\ +\ g(x,y-1,z)\ + \\ & g(x,y,z+1)\ +\ g(x,y,z-1)\ -\ 6g(x,y,z). \end{aligned} \quad (6)$$

To apply ∇^2 to a vector field, we apply the operator to each vector component separately, resulting in a vector field that gives the difference between vectors in the original field and their neighborhood average:

$$\nabla^2 \mathbf{u}(x,y,z) = (\ \nabla^2 u_x(x,y,z),\ \nabla^2 u_y(x,y,z),\ \nabla^2 u_z(x,y,z)\ ). \quad (7)$$

2.3 The Navier-Stokes Equations for Incompressible Flow

the Navier-Stokes equations are a set of two differential equations that describe the velocity field of a fluid, $\mathbf{u}$, over time. They are called *differential* equations because they specify the derivatives of the velocity field rather than the velocity field itself. The first of the equations arises because we are simulating *incompressible* fluids, and is given by

$$\nabla \cdot \mathbf{u} = 0. \quad (8)$$

Simply put, this equation says that the amount of fluid flowing into any volume in space must be equal to the amount flowing out. Note that this is just the divergence of the velocity field. During simulation, we will solve a pressure term to satisfy equation 8.

The second of the Navier-Stokes equations is a little more complicated. It specifies how $\mathbf{u}$ changes over time, and is given by

$$\frac{\partial \mathbf{u}}{\partial t} = -\ (\nabla \cdot \mathbf{u})\mathbf{u}\ -\ \frac{1}{\rho}\nabla p\ +\ \nu\nabla^2\mathbf{u}\ +\ \mathbf{F}. \quad (9)$$

The second equation accounts for the motion of the fluid through space, along with any internal or external forces that act on the fluid. The terms of this equation can be described as follows:

$\frac{\partial \mathbf{u}}{\partial t}$ **The derivative of velocity with respect to time.** This will be calculated at all grid points containing fluid during each time step of the simulation.

$-(\nabla \cdot \mathbf{u})\mathbf{u}$ **The convection term.** This term arises because of the conservation of momentum. Since the velocity field is sampled at fixed spatial locations, the momentum of the fluid must be moved or "convected" through space along with the fluid itself. In practice, we will use a "backward particle trace" method to solve this term.

$-\frac{1}{\rho}\nabla p$ **The pressure term.** This term captures forces generated by pressure differences within the fluid. ρ is the density of the fluid, which is always set to one in our simulations, and p is the pressure. Since we are simulating incompressible fluids, the pressure term will be coupled with equation 8 to make sure the flow remains incompressible.

$\nu\nabla^2\mathbf{u}$ **The viscosity term.** In thick fluids, friction forces cause the velocity of the fluid to move toward the neighborhood average. The viscosity term captures this relationship using the ∇^2 operator. The variable ν is called the *kinematic viscosity* of the fluid, and it determines the thickness of the fluid. The higher the value of ν, the thicker the fluid.

$\mathbf{F}$ **External force.** Any external forces, such as gravity or contact forces with objects are included in this term. In practice, the external force term allows us to modify or set velocities in the grid to any desired value as one of the simulation steps. The fluid then moves naturally, as though external forces have acted on it.

3 The MAC Grid

This section describes how to discretize the velocity and pressure fields used in the Navier-Stokes equations. We use the staggered Marker And Cell (MAC) grid first published in [Harlow and Welch 1965], which has become one of the most popular ways to simulate fluid flow in computer graphics.

The MAC grid method discretizes space into cubical cells with width h. Each cell has a pressure, p, defined at its center. It also has a velocity, $\mathbf{u} = (u_x, u_y, u_z)$, but the components of the velocity are placed at the centers of three of the cell faces, u_x on the x-min face, u_y on the y-min face, and u_z on the z-min face, as shown in figure 1. Staggering the position of the velocity in this manner tends to produce more stable simulations than storing the velocity at the cell centers.

In addition to spatial grid cells, a set of marker particles (points in space) is used to represent the fluid volume. As the simulation progresses, the marker particles are moved through the velocity field and then used to determine which cells contain fluid.

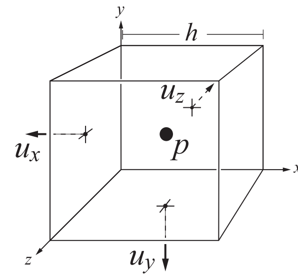


Figure 1: A MAC grid cell. Velocity components, u_x, u_y and u_z, are stored on the minimal faces of the cell. Pressure, p, is stored at the cell center.

3.1 Putting the MAC Grid in a Hashtable

One of the main disadvantages to using a fixed MAC grid structure is that $\mathbf{u}$ is not defined outside the grid bounds. To get around this problem, we use a dynamic grid. Grid cells are created as needed, and destroyed when no longer in use. The cells are stored in a hash table which is keyed by the cell coordinates. We use the hash function presented in [Worley 1996] and used by [Steele et al. 2004]:

$$hash = 541x + 79y + 31z. \tag{10}$$

[Teschner et al. 2003] define a similar hash function that could also be used. Using a hash table version of the MAC grid frees the fluid to go anywhere in the environment, and reduces memory requirements so that the number of grid cells is proportional to the amount of fluid being simulated rather than the entire volume of the simulation space.

Despite the trouble to which we have gone to make the simulation space unbounded, we allow simulations to specify explicit bounds on where the fluid can go. This makes it easy to define walls for the fluid to collide against, and prevents fluid particles from flying off into infinity.

4 Simulation

This section describes our basic simulation method, which is essentially the same as the one presented in [Foster and Fedkiw 2001], except that we use a dynamic MAC grid instead of a static one. Figure 2 shows several screen shots captured from the basic simulator described in this section.

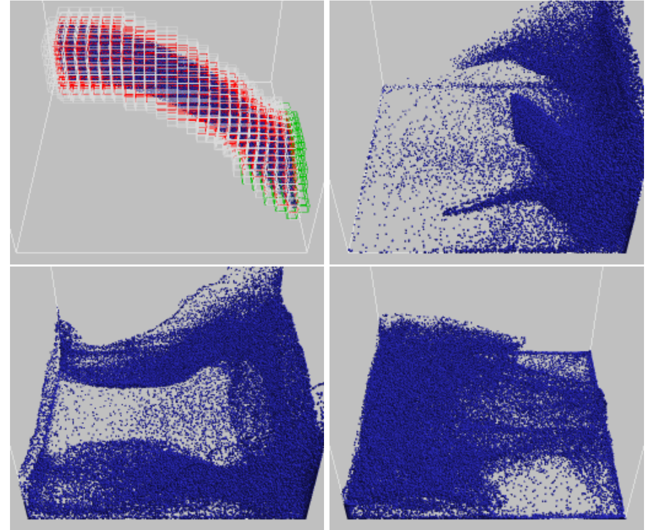


Figure 2: Screen shots from the basic fluid simulator. The upper left image shows the grid cells that define the velocity field. In the other images, the fluid particles that track the liquid volume are shown. The particles are rendered in OpenGL as billboards (pictures of spheres) rather than actual 3D spheres for efficiency.

4.1 The Basic Simulator

The MAC grid method simulates fluid flow by periodically updating the velocity field. Marker particles are moved through the velocity field to keep track of where the fluid is in space. The actual sequence that our simulator uses to do this is shown in figure 3. The remainder of this section will detail each of the steps in the algorithm.

Step 1. Calculate the time step. To avoid unnecessary computation, we would like to use a time step that is as large as possible, but if it is too long, the flow may look unnatural or become unstable. We use a heuristic called the CFL condition to determine what Δt to use. For our application, the CFL condition states that Δt should be small enough so that the maximum motion in the velocity field is less than the width of a grid cell. In other words, $\Delta t \leq h/|\mathbf{u}_{max}|$ where h is the width of a grid cell and $\mathbf{u}_{max}$ is the maximum velocity in $\mathbf{u}$.

In practice, the strict CFL condition can be relaxed somewhat as long as there is a big enough buffer zone around the cells containing fluid. [Foster and Fedkiw 2001] and [Enright et al. 2002] got away with time steps as much as five times larger than dictated by the CFL condition. Our implementation finds Δt by scaling the CFL time step by a user-specified constant, k_{cfl}:

$$\Delta t = k_{cfl} \frac{h}{|\mathbf{u}_{max}|}. \tag{11}$$

Δt is then clamped to lie between user-specified min and max time steps.

Step 2. Update the grid. Before the velocity field can be advanced, the list of cells containing fluid must be updated, and a *buffer zone* of air must be created around the fluid so that the fluid

Basic Fluid Dynamics Algorithm

1. Calculate the simulation time step, Δt (equation 11).
2. Update the grid based on the marker particles (figure 4).
3. Advance the velocity field, $\mathbf{u}$.
   3a. Apply convection using a backwards particle trace.
   3b. Apply external forces.
   3c. Apply viscosity.
   3d. Calculate the pressure to satisfy $\nabla \cdot \mathbf{u} = 0$.
   3e. Apply the pressure.
   3f. Extrapolate fluid velocities into buffer zone (figure 7).
   3g. Set solid cell velocities.
4. Move the particles through $\mathbf{u}$ for Δt time.
   4a. If Δt extends beyond the next displayed frame time
   Advance particles to the next displayed frame.
   Display frame and repeat step 4a.
   4b. Move particles through $\mathbf{u}$ for the remainder of Δt.

Figure 3: Fluid dynamics algorithm

Dynamic Grid Update

```
set "layer" field of all cells to −1

// update cells that currently have fluid in them
for each marker particle, P
  if the cell, C, containing the center of P does not exist
    if C is within the simulation bounds
      create C and put it in the hash table
      set the cell type of C to "fluid"
      C.layer = 0
  else if C is not part of a solid object
    Set the cell type for C to "fluid"
    C.layer = 0

// create a buffer zone around the fluid
for i = 1 to max(2, ⌈k_cfl⌉)
  for each liquid or air cell, C, such that C.layer == i−1
    for each of the six neighbors of C, N
      if N already exists in the hash table
        if N.layer == −1 and N is not solid
          set the cell type of N to "air"
          N.layer = i
      else
        create N and put it in the hash table
        N.layer = i
        if N is in the simulation bounds
          set the cell type of N to "air"
        else
          set the cell type of N to "solid"

delete any cells with layer == −1.
```

Figure 4: Algorithm to update the dynamic grid.

will have a place to flow to during the next time step. Note that the velocities from the previous time step are needed to update the velocity field, so we can't simply throw away the old grid and create a new one. Instead the current grid must be updated. To do this, we identify which grid cells currently contain fluid, and then iteratively create a buffer zone around the fluid cells big enough to accomodate any flow that might occur during Δt time.[1] Finally, any cells that are no longer needed are removed from the grid. In our implementation, each grid cell has a "cell type" field to determine whether the cell contains fluid, air, or is part of a solid object. We also use an integer field called "layer" to help with some of the simulation steps that build outwards from the fluid into the air buffer in layers. The complete algorithm that we use to update the grid is presented in figure 4.

Step 3. Advance the velocity field. After the grid has been updated, we advance the velocity field by Δt. Some of the steps required to advance the velocity field cannot be done in place, and thus require an extra copy of the velocity field data. Our solution is to store a temporary vector in each grid cell. We found this be easier and more memory efficient than keeping two synchronized copies of the hash table grid.

Step 3a. Apply convection using a backwards particle trace. The velocity field is convected using the backwards particle trace method described in [Stam 1999]. A backwards particle trace starts at the location for which we want to update the velocity field, $\mathbf{x} = (x,y,z)$. A virtual particle is traced from $\mathbf{x}$ backwards through $\mathbf{u}$ for $-\Delta t$ time, arriving at point $\mathbf{y}$. The velocity at $\mathbf{x}$ is then replaced by the current velocity at point $\mathbf{y}$. Note that this step cannot be done in place, so we use the temporary vectors in the cells to store intermediate values, and then copy the temporary values to $\mathbf{u}$ after all of the particles have been traced.

The easiest way to trace point $\mathbf{x}$ through the velocity field is to use the Euler step method, which simply evaluates the velocity at $\mathbf{x}$ and then updates the position of $\mathbf{x}$ based on this velocity (i.e. $\mathbf{y} = \mathbf{x} + \Delta t\ \mathbf{u}(\mathbf{x})$). Unfortunately, the Euler step method can be rather inaccurate because the velocity field is not constant. A more accurate method, which we use, is called Runge-Kutta order two interpolation (RK2). The idea behind RK2 is to take half an Euler step, and then use the velocity at this intermediate location as an approximate average over the whole time step:

$$\mathbf{y} = \mathbf{x} + \Delta t\ \mathbf{u}\left(\mathbf{x} + \frac{\Delta t}{2}\mathbf{u}(\mathbf{x})\right). \quad (12)$$

Since the MAC grid stores the components of the velocity at different spatial locations, a separate particle must be traced for each component of $\mathbf{u}$. In addition, the staggered grid makes interpolating the velocity field more difficult (a separate trilinear interpolation is required for u_x, u_y and u_z). Our simulator performs the interpolation by scaling the location to be interpolated by the reciprocal of the cell width, $1/h$. It then shifts the scaled location so that the desired data element can be treated as if it were located at the minimal cell corners. For example, $u_y(x,y,z)$ is found by interpolating from the grid coordinates $(x/h - 0.5,\ y/h,\ z/h - 0.5)$.

While the details of the particle trace and component interpolation may seem trivial, we have found these two concepts to be some of the hardest for implementers to conceptualize and put into code. For this reason, we include pseudocode for these operations in figure 5.

Step 3b. Apply external forces. After the velocity field has been convected, the next step is to add external forces. In the case of gravity, the simulation adds the vector $\Delta t\mathbf{g}$ to the velocities in the grid, where $\mathbf{g} = (g_x, g_y, g_z)$ is the gravitational force defined for the

[1] In practice, the buffer zone should be at least 2 cells in width to avoid some odd interpolation artifacts related to the staggered grid representation.

```
// Trace a particle from point (x, y, z) for t time using RK2.
Point traceParticle(float x, float y, float z, float t)
    Vector V = getVelocity(x, y, z);
    V = getVelocity(x+0.5*t*V.x,  y+0.5*t*V.y,  z+0.5*t*V.z);
    return Point(x, y, z) + t*V;

// Get the interpolated velocity at a point in space.
Vector getVelocity(float x, float y, float z)
    Vector V;
    V.x = getInterpolatedValue(x/h, y/h-0.5, z/h-0.5, 0);
    V.y = getInterpolatedValue(x/h-0.5, y/h, z/h-0.5, 1);
    V.z = getInterpolatedValue(x/h-0.5, y/h-0.5, z/h, 2);
    return V;

// Get an interpolated data value from the grid.
float getInterpolatedValue(float x, float y, float z, int index)
    int i = floor(x);
    int j = floor(y);
    int k = floor(z);
    return (i+1-x) * (j+1-y) * (k+1-z) * cell(i, j, k).u[index] +
            (x-i) * (j+1-y) * (k+1-z) * cell(i+1, j, k).u[index] +
            (i+1-x) * (y-j) * (k+1-z) * cell(i, j+1, k).u[index] +
            (x-i) * (y-j) * (k+1-z) * cell(i+1, j+1, k).u[index] +
            (i+1-x) * (j+1-y) * (z-k) * cell(i, j, k+1).u[index] +
            (x-i) * (j+1-y) * (z-k) * cell(i+1, j, k+1).u[index] +
            (i+1-x) * (y-j) * (z-k) * cell(i, j+1, k+1).u[index] +
            (x-i) * (y-j) * (z-k) * cell(i+1, j+1, k+1).u[index];
```

Figure 5: Pseudocode for functions to trace a particle through the velocity field, get the velocity at a point in space, and determine an interpolated data value at a real-valued grid location. The large expression in "getInterpolatedValue" is just the trilinear interpolation formula. Note that in practice some of the grid cells requested in this routine may not exist. An actual implementation of "getInterpolatedValue" should drop terms associated with non-existant cells, and then divide the result by the total weight associated with the remaining terms. (For example, the weight for cell(i+1, j, k) is (x-i)*(j+1-y)*(k+1-z).) In our simulator, MAC cells store pointers to all of the seven neighbors used by "getInterpolatedValue" and a flag telling whether all of these interpolating neighbors exist. This allows the routine to bypass hashtable lookups and individual checks to see if the neighbors exist in most instances.

simulation. Once again, external force is only applied to velocity components that border fluid cells.

Besides adding simple body forces such as gravity, the external force step can be used as a versatile tool to create different flow effects. Later we will describe how to use this step to make spouts and fountains, enhance swirling motions in the flow and create viscoelastic effects.

Step 3c. Apply viscosity. We solve the viscosity term by directly evaluating $\nabla^2\mathbf{u}$ as in equation 7, and then adding $\Delta t \nu \nabla^2 \mathbf{u}$ to $\mathbf{u}$. As in the case of convection and external force, we only apply viscosity to the components of $\mathbf{u}$ that border fluid cells. In addition, only components of $\mathbf{u}$ that border fluid cells are allowed to take part in the computation of $\nabla^2\mathbf{u}$, so that some of the terms in equation 6 are omitted for cells on the fluid border. Note that as with convection, the algorithm cannot apply viscosity in place, so once again the temporary vectors are used to store intermediate values.

Step 3d. Calculate pressure. At this point, we have a velocity field that does not satisfy equation 8, but we still have to apply the pressure term. What we would like to do is set the pressures in fluid cells so that the divergence throughout the fluid will be zero. For any given fluid cell, we could alter its pressure to satisfy $\nabla \cdot \mathbf{u} = 0$. Doing this would change the divergence of the neighboring cells, however. To make all of the fluid cells divergence free after pressure has been applied, we need to solve for all of the pressures simultaneously. This gives rise to a large, sparse linear system with one variable for the pressure of each cell containing fluid. The system can be written as follows:

$$\mathbf{A}\ \mathbf{P} = \mathbf{B} \tag{13}$$

where $\mathbf{A}$ is a matrix of coefficients, $\mathbf{P}$ is the vector of unknown pressures for which we want to solve, and $\mathbf{B}$ is a vector based on the divergence the velocity field after step 3c.

The values in $\mathbf{A}$ are defined as follows: each fluid cell, C_i, is assigned a row in $\mathbf{A}$. The coefficient corresponding to the C_i in that row, a_{ii}, is set to minus the number of non-solid neighbors of C_i. For all liquid neighbors of C_i, the coefficient in row i is set to 1. Other coefficients in row i are set to zero.

As an example of how to set values in matrix $\mathbf{A}$, consider the hypothetical cell, C_{10}, shown in figure 6. C_{10} has six neighbors, two air cells, two solid cells, and two fluid cells, C_9 and C_{21}. Note that only fluid cells are numbered, since other cells do not go into the matrix. With this information, we can fill in row 10 of the matrix. $A_{10,10}$ is set to -4 (negative the number of non-solid neighbors of C_{10}). $A_{10,9}$ and $a_{10,21}$ are set set to 1. All other values in row 10 are set to zero.

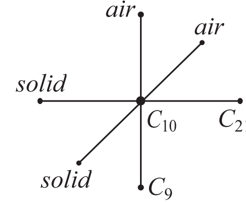


Figure 6: A hypothetical cell, C_{10}, and its neighbors. For clarity, cells are shown as dots rather than cubes.

The values of vector $\mathbf{B}$ in equation 13 are calculated using a modified version of $\nabla \cdot \mathbf{u}$ (equation 5) in which velocity components between fluid cells and solid cells are considered to be zero. In addition to the divergence term, we explicitly subtract the pressure contributions of air cells since they are not included in the matrix coefficients. This leaves

$$b_i = \frac{\rho h}{\Delta t}(\nabla \cdot \mathbf{u}_i) \ - \ k_i p_{atm} \tag{14}$$

where b_i is the i-th entry in $\mathbf{B}$, ρ is the density in $cell_i$, h is the cell width, $\nabla \cdot \mathbf{u}_i$ is the modified divergence just described, k_i is the number of air cells neighboring $cell_i$, and p_{atm} is the atmospheric pressure. In matrix form, the example system just described looks something like

$$\begin{bmatrix} & & \vdots & & \\ & & \vdots & & \\ \dots & 1 & -4 & \dots & 1 \\ & & \vdots & & \\ & & \vdots & & \end{bmatrix} \begin{bmatrix} \vdots \\ p_9 \\ p_{10} \\ \vdots \\ p_{21} \end{bmatrix} = \begin{bmatrix} \vdots \\ \vdots \\ \frac{\rho h}{\Delta t}(\nabla \cdot \mathbf{u}_{10}) - 2p_{atm} \\ \vdots \\ \vdots \end{bmatrix}$$

Our simulator uses a fairly standard implementation of the conjugate gradient method to solve equation 13 for the vector of unknown pressures. We believe that any method designed for sparse matrices will probably work well, and recommend downloading one of the many free sparse matrix solvers available online instead of coding one from scratch. (Creating a good sparse matrix solver may be a larger project than the rest of the simulator combined!)

Step 3e. Apply pressure. Pressure is only applied to the velocity components in $\mathbf{u}$ that border fluid cells but not solid cells. Air cells are considered to be at atmospheric pressure and have a density of one. The velocity $\mathbf{u}(x,y,z)$ is updated by

$$\mathbf{u}_{new}(x,y,z) = \mathbf{u}(x,y,z) - \frac{\Delta t}{\rho h}\nabla p(x,y,z) \tag{15}$$

where h is the cell width, ρ is the density of the cell, and $\nabla p(x,y,z)$ is the pressure gradient evaluated using equation 4.

Step 3f. Extrapolate fluid velocities into surrounding cells. Once the pressure has been applied, the velocity field conforms to the Navier-Stokes equations within the fluid, but velocities outside the fluid volume need to be set. We find these velocities using a simple extrapolation method that propagates the known fluid velocities into the buffer zone surrounding the fluid. Figure 7 gives the algorithm that we use to extrapolate the velocities. A better, but more complicated extrapolation method is described in [Enright et al. 2002].

Extrapolation of Fluid Velocities

set "layer" field to 0 for fluid cells and -1 for non fluid cells

for i = 1 to $max(2, \lceil k_{cfl} \rceil)$
  for each cell, C, such that C.layer == -1
    if C has a neighbor, N, such that N.layer == $i-1$
      for velocity components of C not bordering fluid cells, u_j
        set u_j to the average of the neighbors of C
          in which N.layer == $i-1$.
      C.layer = i

Figure 7: Algorithm to extrapolate fluid velocities into the buffer zone surrounding the fluid.

Step 3g. Set the velocities of solid cells. To prevent the fluid entering solid objects, we set velocity components that point into solid cells from liquid or air cells to zero. Velocity components that point out of solid cells, or that lie on the border between two solid cells are left unchanged.

Step 4. Move the marker particles through the velocity field. After the velocity field has been advanced, the marker particles can be moved. Since Δt does not necessarily coincide with frame boundaries, the particles should not always be advanced by Δt, however. There are two cases to consider. If the current time step does not include the next displayed frame, we advance the particles by Δt. If, on the other hand, the current time step includes the next displayed frame, we advance the particles to the next displayed frame, and then repeat until Δt time is exhausted. In both cases, we use RK2 interpolation to advance the particles (equation 12).

4.2 Extensions to the Basic Simulator

At this point we have finished describing the basic fluid simulator. If you are implementing a fluid flow system based on this tutorial, it would be a good idea to get the basic simulator that we have just described working as a first milestone, then proceed to the extensions presented in later sections. All of the extensions build directly on the framework of the basic simulator, and can be programmed independent of one another. They will be presented in the following order:

- Adding fluid sources and sinks to the simulator is described in section 5.
- A method to allow the fluid to respond to moving objects is described in section 6.
- Section 7 explains how to to simulate fluids with variable or high viscosity.
- A techiniqe to animate viscoelastic fluids such as jello or liquid soap is presented in section 8.
- Section 9 explains how to modify the simulator to animate smoke instead of liquids.
- Finally, section 10 describes how *level sets* can be used to create a believable liquid surface.

5 Fluid Sources and Sinks

A simple addition that can be made to the simulator is the ability to have fluid sources and sinks. With this capability, simulations are not constrained to have a fixed amount of fluid.

Sources. Fluid sources introduce fluid into the simulation. We support two types of sources, *blob sources* and *spout sources*.

A blob source is nothing more than a group of fluid particles that is added to the simulation at a specified time with a specified initial velocity. Note that the initial fluid state can be thought of as a set of blob sources which are introduced at the beginning of the simulation.

Spout sources allow fluid to be sprayed into the simulation as if from a nozzle. We implement spout sources using a planar object (e.g. a circle or rectangle) from which fluid particles emerge with a specified speed, s. A planar spout source is positioned in space, and the velocity of cells that the spout intersects are set to point in the direction of the spout normal, $\mathbf{n}_s$, with magnitude s at the beginning of each time step. At regular time intervals, fluid particles are created over the entire area of the spout. The time interval between adding particles is d/s, where d is the desired particle spacing and s is the spout speed. Note that particle creation times may not coincide exactly with time steps. To remedy this problem, we add particles only at the beginnings of time steps and move them along the velocity field as needed to correspond to the exact time when they should have been introduced into the simulation.

Sinks. Just as fluid sources introduce fluid into a simulation, fluid sinks take it away. Sinks are even easier to implement than sources. Our simulator handles fluid sinks by specifying "sink" cells that always contain air. Since air cells are always set to atmospheric pressure, the fluid will naturally flow into these cells. After each time step, fluid particles that end up in a sink cell are removed from the simulation.

Pumps and fountains. Besides adding or removing fluid from a simulation, other interesting effects can be achieved by direct manipulation of the velocity field. As mentioned in section 2.3, the velocity of any cell or group of cells can be set arbitrarily during the external force step of the algorithm, and the fluid will react naturally, as if a force has acted on it. We can use this fact to create pumps and fountains within the fluid.

6 Fluid-Solid Interaction

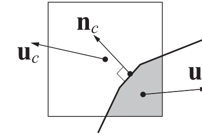
This section describes a method to make a fluid simulation respond realistically to moving polygonal objects, which was first presented in [Foster and Fedkiw 2001]. Their interaction model does not provide a way for the fluid to affect object motion; however, it can produce physically convincing behavior in many instances.

A theoretical model of object interaction. In the last section we noted that the velocity of any cell in the grid can be set arbitrarily, and the fluid will react as if a force has acted on it. A simple way to make the fluid react to a moving object would therefore be to determine which grid cells are occupied by the object and set the velocity of those cells to the object velocity. In theory, the fluid would react as though it had been pushed out of the way by the object.

Problems with the theoretical model. There are several problems with the interaction model just outlined. A first problem is that if the velocity field is just set the to object velocity any fluid particles that come close to the surface of the object will stick to it. Stickiness may be desireable in some simulations, but more often we want the fluid to slip around objects rather than stick to them. An even bigger problem is the pressure equation. When the pressure update occurs, it will modify the velocity field and push fluid back into the object, an unacceptable situation.

Foster and Fedkiw's solution. [Foster and Fedkiw 2001] present an algorithm that solves these two problems. They handle the stickiness problem by only modifying the part of the velocity that is normal to the object. Thus, fluid will be pushed out of the object while still being able to slide across it. Additionally, they solve the pressure problem by changing the pressure equation so that velocities inside or on the border of objects are held fixed. In the context of the simulator described in section 4, Foster and Fedkiw's object interaction algorithm can be described as follows:

- Steps 3a through 3c (convection, viscosity and external force) continue as usual. During these steps, cells inside or on the border of moving objects are considered to contain fluid.
- After step 3c, the velocity of any cell that is completely inside an object is set to the velocity of the object within that cell, $\mathbf{u}_o$.
- An average surface normal for each cell on the object border, $\mathbf{n}_c$, is then calculated. This can done by averaging the normals of any polygons that intersect the cell.
- Next, the velocity of the object within border cells, $\mathbf{u}_o$ is calculated. Note that this value will be constant if the object is moving uniformly.
- After this, the value of the velocity field at the center of border cells, $\mathbf{u}_c$, is found by interpolation.
- $\mathbf{U}_c$ is then pushed in the normal direction, $\mathbf{n}_c$, to prevent the flow from going into the object. Mathematically, $(\mathbf{u}_o \cdot \mathbf{n}_c)$ is compared to $(\mathbf{u}_c \cdot \mathbf{n}_c)$. If $(\mathbf{u}_o \cdot \mathbf{n}_c)$ is greater than $(\mathbf{u}_c \cdot \mathbf{n}_c)$, $\mathbf{u}$ is flowing into the object (something it shouldn't do). To solve this problem, $\mathbf{u}_c$ is changed as in equation 16.



$$\mathbf{u}_c^{new} = \mathbf{u}_c + max\,(0, \quad (\mathbf{u}_o \cdot \mathbf{n}_c) - (\mathbf{u}_c \cdot \mathbf{n}_c))\,\mathbf{n}_c. \tag{16}$$

- The newly modified $\mathbf{u}_c$ is copied back into the grid by setting the six velocity components that surround the border cell.
- The pressure equation (Steps 3d and 3e) is modified to hold fixed the velocities of grid cells inside and on the borders of objects. The modifications are as follows: cells inside or on the border of objects are not included in matrix $\mathbf{A}$ in equation 13. However, these cells are allowed to contribute to the divergence $(\nabla \cdot \mathbf{u}_i)$ in equation 14. Finally, during the pressure update (step 3e), velocity components on the faces of cells inside or on the borders of objects are not changed.

The result of this algorithm is that the fluid reacts to object motion in a physically plausible way. Moving objects push the fluid out of the way, but the fluid is free to flow around the objects without getting stuck to them. Additionally, pressure is handled in a physically consistent way. The velocity field remains divergence free, and fluid does not flow into objects. Figure 8 summarizes the object interaction algorithm.

Fluid-Solid Interaction

New Parameters
$\mathbf{u}_o$ The velocity of the object in a grid cell.
$\mathbf{n}_c$ The average normal of the object in a cell.

Changes to the Basic Algorithm
Steps 3a to 3c:
Cells inside moving objects are considered to be fluid.
After step 3c:
Set velocity of cells inside objects to the object velocity.
For each cell on the surface of a moving object, C:
Find the average object surface normal inside C, $\mathbf{n}_c$.
Determine the object velocity inside C, $\mathbf{u}_o$.
Find the value of the velocity field at the center of C, $\mathbf{u}_c$.
Push $\mathbf{u}_c$ in the direction of $\mathbf{n}_c$. (equation 16)
Distribute the components of $\mathbf{u}_c$ to the faces of cell C.
Steps 3d and 3e:
Modify the pressure term to hold velocities inside object cells constant.

Figure 8: Modifications to the fluid dynamics algorithm (figure 3) to allow the fluid to respond to moving objects.

7 Variable and High Viscosity Fluids

One of the shortcomings of the simulator defined in section 4 is that it does not handle variable or high viscosity fluids. Variable

viscosity is not accounted for in the model, and if the viscosity is set too high, the simulation becomes unstable. This section describes an extension to handle variable and high viscosity fluids that was originally presented in [Carlson et al. 2002].

To start with, let's look at why the simulation blows up if the viscosity is too high. Recall from section 4 that viscosity was modeled using the Laplacian of the velocity field, $\nabla^2\mathbf{u}$. To compute the viscous acceleration we scaled the Laplacian by the viscosity (ν) and the time step (Δt) and added the result to the velocity. This had the effect of pushing the velocity towards the neighborhood average. Unfortunately, if $\nu\Delta t$ gets above about 1/6, the viscous force vector pushes $\mathbf{u}$ beyond the neighborhood average, reversing the flow direction. To solve this problem, we could decrease the viscosity so that $\nu\Delta t$ is always under 1/6, but this would alter the properties of the fluid. We could also decrease the time step, but this would make the simulation very slow for high viscosity fluids. [Carlson et al. 2002] showed that a better solution is to use what is called *implicit* integration to calculate the viscous force.

7.1 Explicit and Implicit Integration.

The way that viscosity was modeled in section 4 is called *explicit* integration. Explicit integration schemes compute the derivatives of the function to be integrated and then march forward in time. In our case, the function to be integrated is the velocity field $\mathbf{u}$, and its derivative is $\nu\nabla^2\mathbf{u}$. (For this example, we will only consider a single component of the velocity, since all of the components are handled in the same way.) We can write the viscosity equation as follows:

$$u_{new} = u + k\nabla^2 u$$

where $k = \nu\Delta t$. The expression above can be rewritten in matrix form by collecting the values of u into a column vector, and putting the coefficients of the equation in a matrix:

$$\begin{bmatrix} & & \vdots & & \\ & & \vdots & & \\ & & \vdots & & \\ .. \; k \; .. & (1-6k) & .. \; k \; .. \\ & & \vdots & & \\ & & \vdots & & \\ & & \vdots & & \end{bmatrix} \begin{bmatrix} \vdots \\ u(x-1,y,z) \\ \vdots \\ u(x,y,z) \\ \vdots \\ u(x,y,z+1) \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \\ u_{new}(x,y,z) \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

This equation is really easy to solve. All we have to do is multiply the vector of coefficients by the known vector of current velocities to get the answer. Unfortunately, as we have seen, this is an explicit integration scheme that can be unstable. In *implicit* integration, stability is gained by posing the problem backwards. Instead of calculating the derivatives of the function that we want to integrate, we calculate the negative of the derivatives, and then march backwards along these negative derivatives one time step. In the case of viscosity, the negative of the derivative is $-\nu\nabla^2\mathbf{u}$. We again pose the problem in matrix form, but this time the known vector of current velocities is placed on the right hand side of the equals sign and the unknown vector that we want to solve for is placed on the left:

$$\begin{bmatrix} & & \vdots & & \\ & & \vdots & & \\ & & \vdots & & \\ .. -k \; .. & (1+6k) & .. \; -k \; .. \\ & & \vdots & & \\ & & \vdots & & \\ & & \vdots & & \end{bmatrix} \begin{bmatrix} \vdots \\ \vdots \\ \vdots \\ u_{new}(x,y,z) \\ \vdots \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ u(x-1,y,z) \\ \vdots \\ u(x,y,z) \\ \vdots \\ u(x,y,z+1) \\ \vdots \end{bmatrix} \quad (17)$$

The equation is no longer trivial, but it can be solved with the same sparse solver that was used for the pressure equation in section 4.

Implicit integration works because it defines an explicit integration scheme that is run backwards. Running explicit integration forwards tends to decrease the stability of the system, but running it backwards actually makes the system more stable! Using implicit integration may seem like a lot of work, but it is worth the extra effort. What we gain is the ability to take arbitrarily large time steps without compromising stability.

7.2 Variable Viscosity

[Carlson et al. 2002] handle variable viscosity by augmenting the simulator to store temperature values at the centers of grid cells. The temperature is advected and diffused, and then used to calculate the viscosity at the cell centers.

Updating the temperature. The temperature, q, is advected using a backwards particle trace that is nearly identical to the one used to advect the velocity field. (i.e. A virtual particle is traced backwards from the cell center along $\mathbf{u}$ for one time step using RK2 interpolation. the temperature at the cell center is then replaced with the interpolated temperature at the resulting location.) Finally, the temperature is diffused using the equation

$$q_{new} = q + k_{therm}\Delta t \; \nabla^2 q \quad (18)$$

where Δt is the usual time step, k_{therm} is a property of the fluid called the *thermal diffusion constant*, and $\nabla^2 q$ is the Laplacian of the temperature evaluated using equation 6.

Calculating the viscosity. Viscosity is computed at the centers of grid cells using a linear model that transitions between a maximum and minimum viscosity within a specified temperature zone, as shown in figure 9. The viscosity for a given temperature, ν_q, is given by the interpolation formula

$$\nu_q = \begin{cases} \nu_{max} & \text{if } q < q_{min} \\ \nu_{max} - \frac{(q-q_{min})(\nu_{min}-\nu_{max})}{q_{max}-q_{min}} & \text{if } q_{min} \le q \le q_{max} \\ \nu_{min} & \text{if } q > q_{max} \end{cases} \quad (19)$$

where ν_{min} and ν_{max} are the minimum and maximum viscosities of the fluid, and q_{min} and q_{max} define the viscosity transition zone for the fluid. Note that these values are properties of the fluid as a whole, so they do not need to be stored in each grid cell.

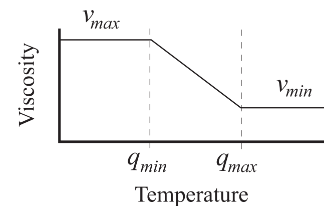


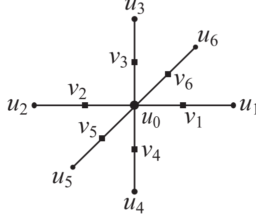
Figure 9: Variable viscosity is calculated using a linear model that transitions from ν_{max} to ν_{min} between the temperatures q_{min} and q_{max}.

For reasons that will become apparent shortly, we store the logarithm of the viscosity at the cell centers instead of the viscosity itself. We also extrapolate the (log) viscosity values calculated within fluid cells into the air buffer, so that the viscosity at the surface of the fluid will not be contaminated by the air temperature.

Using the variable viscosity. At this point, we have (log) viscosity values at the centers of the grid cells. However, viscosities will need to be calculated at other points in space as well. [Carlson et al. 2002] note that better results are obtained using geometric

rather than arithmetic averaging to interpolate the viscosity (i.e., the average of v_1 and v_2 should be $\sqrt{v_1\ v_2}$ rather than $(v_1 + v_2)/2.$), and this is why we store the logarithm of the viscosities. By linearly interpolating the logarithm of the viscosities, we are implicitly computing the geometric interoplation of the viscosities themselves.

To use variable viscosity to update the velocity field, we need to modify the Laplacian operator (equation 6). Recall that we Laplacian (∇^2) uses six neighbors around the point to be evaluated. To simplify the notation, we will label this neighborhood as follows:



where $u_0 \dots u_6$ represent a velocity component at a given cell location and its six neighbors, and $v_1 \dots v_6$ represent the viscosity between u_0 and its neighbors. The way that the Laplacian is modified is to weight the terms of the Laplacian by the viscosity evaluated halfway between u_0 and its neighbors. Using the labels just described, the modified Laplacian becomes

$$\nabla_v^2 = v_1 u_1 + v_2 u_2 + v_3 u_3 + v_4 u_4 + v_5 u_5 + v_6 u_6 - u_0 \sum_{i=1}^{6} v_i$$

where ∇_v^2 is the variable Laplacian operator, and v_i is viscosity evaluated halfway between u_0 and u_i using geometric interpolation.

To avoid instability, [Carlson et al. 2002] use implicit integration to solve the variable viscosity term. The matrix of coefficients used for the implicit integration is based on the variable Laplacian just defined, and is given in equation 20. The two vectors in the system, which we omit here for the sake of space, are exactly as defined in equation 17.

$$\begin{bmatrix} & & & & \vdots & & & \\ & & & & \vdots & & & \\ -\Delta t v_1 \;..\; -\Delta t v_2 \;..\; -\Delta t v_3 \;..\; \left(1 + \Delta t \sum_{i=1}^{6} v_i\right) \;..\; -\Delta t v_4 \;..\; -\Delta t v_5 \;..\; -\Delta t v_6 \\ & & & & \vdots & & & \\ & & & & \vdots & & & \end{bmatrix} \quad (20)$$

Figure 10 summarizes the changes to the fluid simulator needed to simulate variable and high viscosity.

8 Simulation of Viscoelastic Fluids

Some fluids exhibit more complex behaviors than can be described by equation 9. For instance, jello and egg-white bounce and wiggle because of internal stresses that build up as they move. Fluids that behave in this way are called *viscoelastic* because they display some properties of viscous fluids and some properties of elastic solids (solids that return to their original shape after deformation). This section describes a method to animate viscoelastic fluids that was presented in [Goktekin et al. 2004].

The tensor product, $\otimes$. The algorithm used by [Goktekin et al. 2004] to animate viscoelastic fluids relies on the *tensor* or *outer* product of vectors. A good way to understand the tensor product is to compare it to the dot product. A dot product can the thought of as the product of a 1×3 row matrix and a 3×1 column matrix. The result is a 1×1 matrix that is treated as a scalar quantity:

Simulating High and Variable Viscosity

New Parameters

v_{min}, v_{max}	Minimum and maximum viscosity of the fluid.
q_{min}, q_{max}	Viscosity transition zone temperature limits.
k_{therm}	The thermal diffusion constant of the fluid.
q	Temperature at the cell centers.

Changes to the Basic Algorithm

Before step 3a:
- Advect temperature using a backwards particle trace.
- Diffuse temperature (equation 18).

Replace step 3c with:
- Calculate viscosity at cell centers (equation 19).
- Store logarithm of viscosities at cell centers.
- Extrapolate (log) viscosities from fluid cells into air buffer.
- Solve the variable viscosity term for u_x, u_y and u_z separately using implicit integration (equations 17 and 20).

Figure 10: Modifications to the basic fluid algorithm (figure 3) to handle variable and high viscosity fluids.

$$\begin{bmatrix} x & x & x \end{bmatrix} \begin{bmatrix} x \\ x \\ x \end{bmatrix} = [\,x\,]$$

The tensor product, denoted by $\otimes$, does the opposite of the dot product. The vectors are multiplied again, but this time the first vector is considered a column matrix and the second is considered a row matrix. The result is a 3×3 matrix:

$$\begin{bmatrix} x \\ x \\ x \end{bmatrix} \begin{bmatrix} x & x & x \end{bmatrix} = \begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix}$$

The viscous stress tensor. Taking the tensor product of the gradient and the velocity field, $\nabla \otimes \mathbf{u}$ yields a matrix filled with all the partial derivatives of $\mathbf{u}$:

$$\nabla \otimes \mathbf{u} = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z \end{bmatrix} = \begin{bmatrix} \frac{\partial u_x}{\partial x} & \frac{\partial u_y}{\partial x} & \frac{\partial u_z}{\partial x} \\ \frac{\partial u_x}{\partial y} & \frac{\partial u_y}{\partial y} & \frac{\partial u_z}{\partial y} \\ \frac{\partial u_x}{\partial z} & \frac{\partial u_y}{\partial z} & \frac{\partial u_z}{\partial z} \end{bmatrix}.$$

If we compute one half of this matrix plus its transpose, the result is a symmetric matrix, $\mathbf{T} = (\nabla \otimes \mathbf{u} + (\nabla \otimes \mathbf{u})^T)/2$, that has a very surprising property. In particular, it describes all the ways in which $\mathbf{u}$ is warping and stretching. One way to think about it is to imagine that rubber bands are embedded throughout the fluid. The matrix $\mathbf{T}$, which is commonly referred to as the *viscous stress tensor*, describes the rate at which the rubber bands are stretching or contracting as the fluid moves. $\mathbf{T}$ can be written explicitly as follows:

$$\mathbf{T} = \frac{1}{2} \begin{bmatrix} \frac{2\partial u_x}{\partial x} & \frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y} & \frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z} \\ \frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y} & \frac{2\partial u_y}{\partial y} & \frac{\partial u_z}{\partial y} + \frac{\partial u_y}{\partial z} \\ \frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z} & \frac{\partial u_z}{\partial y} + \frac{\partial u_y}{\partial z} & \frac{2\partial u_z}{\partial z} \end{bmatrix}. \quad (21)$$

The definition of **T** may look complicated, but once again, we just evaluate the terms of the matrix using finite differences. Specifically, the diagonal elements of the **T** are evaluated using forward differences, and the off-diagonal elements are evaluated using backward differences. In other words,

$$\frac{\partial u_x}{\partial x} = u_x(x+1,y,z) - u_x(x,y,z) \text{ and}$$
$$\frac{\partial u_y}{\partial x} = u_y(x,y,z) - u_y(x-1,y,z).$$

The elastic strain tensor. **T** describes the rate at which elastic strain builds up as a fluid moves, but this is not the whole story. To compute the elastic force exerted on a viscoelastic fluid the simulator must determine the total strain that exists within the fluid. To put it in terms of our rubber band analogy, the fluid simulator must determine the total amount that the rubber bands have stretched since the beginning of the simulation. Fortunately, the total elastic strain at a point in space can be described using another 3×3 symmetric *elastic strain tensor*, which we will call **E**. To simulate viscoelastic fluids, [Goktekin et al. 2004] place a separate elastic strain tensor in each MAC cell, storing the elements of the tensor at different spatial locations within the cell. Specifically, the diagonal elements of **E** are stored at the cell centers, and the off-diagonal elements are stored on the minimal cell edges (see figure 11).

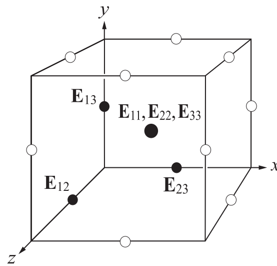


Figure 11: Elements of the elastic strain tensor, **E**, are stored at various locations within a MAC cell. The diagonal elements of the tensor, $\mathbf{E}_{11}$, $\mathbf{E}_{22}$ and $\mathbf{E}_{33}$ are stored at the center of the cell. Off-diagonal elements $\mathbf{E}_{12}$, $\mathbf{E}_{13}$ and $\mathbf{E}_{23}$ are stored on the minimal cell edges as shown. Elements below the diagonal are not stored since the tensor is symmetric. Tensor elements from neighboring cells are shown as hollow circles.

Integrating the elastic strain. At the beginning of a simulation, the elastic strain tensors stored in the grid cells are set to all zeros, which corresponds to zero strain. As the simulation progresses, strain builds up according to the differential equation

$$\frac{\partial \mathbf{E}}{\partial t} = \mathbf{T} - k_{yr}\, max(0,\ \|\mathbf{E}\| - k_{yp}) \frac{\mathbf{E}}{\|\mathbf{E}\|} - (\nabla \mathbf{E}) \cdot \mathbf{u}\ , \quad (22)$$

where $\frac{\partial \mathbf{E}}{\partial t}$ is the rate at which the elastic strain is changing over time. Note that the right hand side of the equation consists of three terms. Each of the terms are solved separately before the velocity field convection step:

- The first term of equation 22, **T**, is just the viscous stress tensor, and we evaluate it using forward and backward differences as was described earlier in this section. In fact, the elements of **E** are stored where they are because of the manner in which we evaluate **T**. Our simulator solves his term by adding $\Delta t\ \mathbf{E}$ to the elastic strain tensors.

Simulating Viscoelastic Fluids

New Parameters

k_{em}	The elastic modulus of the fluid.
k_{yp}	The elastic yield point of the fluid.
k_{yr}	The elastic yield rate of the fluid.
E	The elastic strain tensor, an array of 6 values stored at various locations in each grid cell.

Changes to the Basic Algorithm

Before step 3a:
- Calculate **T** and accumulate new strain in **E**.
- Yield the elastic strain.
- Advect the elastic strain tensors.
- Extrapolate the elastic strain into the air buffer.

Before step 3c:
- Apply elastic strain force to **u** using $\nabla \cdot \mathbf{E}$.

Figure 12: Modifications to the basic fluid algorithm to model viscoelastic fluids.

- The second term, $k_{yr}\, max(0,\ \|\mathbf{E}\| - k_{yp})\ \mathbf{E}/\|\mathbf{E}\|$, describes the *plastic yielding* that occurs in the fluid. To understand how this term works, let's return to the rubber band analogy that we used earlier. In a viscoelastic fluid, elastic strain can build up (the rubber bands stretch), but if too much strain builds up, the fluid permanently deforms to relieve the strain (some of the rubber bands break). This effect is known as plastic yielding. The plastic yielding of a viscoelastic fluid is modeled by two parameters. The first, k_{yp}, is called the *elastic yield point* of the fluid, and it describes how much strain the fluid can support before it starts yielding. The second parameter, k_{yr}, is called the *elastic yield rate* of the fluid, and it governs the rate at which plastic yielding will occur if the strain exceeds k_{yp}. The expression $\|\mathbf{E}\|$ is the magnitude of **E**, which is just the square root of the sum of the squares of the nine elements of **E**. In our simulator, this term is evaluated once for each grid cell, treating the tensor stored in the cell as a unit. The term is then multiplied by the time step and added to **E**. [2]

- The third term, $-(\nabla \mathbf{E}) \cdot \mathbf{u}$, describes the advection of the elastic strain tensor along with the fluid volume. Like velocity, temperature and smoke density, we solve the advection of **E** using the backwards particle trace method. Four particle traces per cell are needed to advect the tensors since the elements of **E** are stored at four different locations. (See figure 11.)

Once the elastic strain tensors have been advected, we extrapolate tensor values that exist in the fluid into the air buffer. The extrapolation is done in the same manner extrapolation of the velocity field values. Tensor elements that border fluid cells are left unchanged, and other tensor values are extrapolated outward from the fluid.

The elastic strain force. The elastic strain in a viscoelastic fluid exerts a force on the fluid proportional to the divergence of the elastic strain. The divergence of a tensor is just a vector that contains the divergence of the columns of the tensor matrix. In other words, if $\mathbf{c}_1$, $\mathbf{c}_2$ and $\mathbf{c}_3$ are the columns of **E**, then

[2] [Goktekin et al. 2004] yield the components of **E** separately, assembling interpolated tensors at the locations where the components of **E** are stored to compute $\|\mathbf{E}\|$.

$\nabla \cdot \mathbf{E} = (\nabla \cdot \mathbf{c}_1,\ \nabla \cdot \mathbf{c}_2,\ \nabla \cdot \mathbf{c}_3)$. The actual force exerted on the fluid is given by

$$\mathbf{F}_e = k_{em}\, \nabla \cdot \mathbf{E} \tag{23}$$

where k_{em} is a property of the fluid called the *elastic modulus*. To evaluate $\nabla \cdot \mathbf{E}$ both forward and backward differences are used. Backward differences are used to evaluate terms related to diagonal elements of the tensor, and forward differences are used to evaluate terms related to off-diagonal elements. For example:

$$\begin{aligned}\nabla \cdot \mathbf{c}_1(x,y,z) = \; & (\mathbf{E}_{11}(x,y,z) - \mathbf{E}_{11}(x-1,y,z)) + \\ & (\mathbf{E}_{21}(x,y+1,z) - \mathbf{E}_{21}(x,y,z)) + \\ & (\mathbf{E}_{31}(x,y,z+1) - \mathbf{E}_{31}(x,y,z)).\end{aligned}$$

Note that this is the opposite order from the evaluation of $\mathbf{T}$. Figure 12 summarizes the changes to the basic simulator that are needed to simulate viscoelastic fluids.

9 Smoke Simulation

It is quite straightforward to modify the basic simulator to handle smoke and other gaseous phenomena. In this section we describe how to modify the simulator to animate smoke or steam instead of liquids. The resulting animation system will be similar to the one described by [Fedkiw et al. 2001].

9.1 Changes to the Simulator

Changes to the grid. To model smoke, we add a smoke density, ρ_s, and temperature, q, to the centers of the MAC grid cells. These quantities are advected along the velocity field using a backwards particle trace in exactly the same way as the components of $\mathbf{u}$.

Tracking the smoke volume without particles. Rather than using particles to track the smoke volume, the smoke density itself is used. Smoke is considered to exist for simulation purposes wherever the smoke density exceeds some small positive constant, ρ_0. Note that smoke could be defined to exist wherever the smoke density is non zero, but we found this definition to be impractical because small amounts of smoke leak into a large surrounding volume during the course of a simulation.

The buffer zone. Just as in the liquid case, we create a buffer zone of air around the smoke for the smoke to flow into during a given time step. The algorithm to create the buffer zone proceeds as in figure 4, except that ρ_s is used to determine the initial set of "fluid" cells. Additionally, the width of the buffer zone needed for smoke simulation is not merely a function of the CFL number, as in the liquid case. Since smoke has approximately the same density as the air that surrounds it, the pressure that the surrounding air exerts on the smoke is important. To account for this effect in our simulations, we require the air buffer to be at least three grid widths in size. A final difference between the buffer zone that is created for smoke and the one created for liquids is that the velocity in the air buffer is not extrapolated from the velocity within the smoke volume (i.e. The algorithm in figure 7 is not used.) Instead, the velocity in the air buffer is allowed to evolve as though it were part of the smoke volume. When cells are added to the air buffer, their velocity components are initially set to the prevailing wind velocity.

The pressure equation. When simulating liquids, air cells were excluded from the pressure calculation. In a smoke simulation, however, the air buffer forms a "pressure envelope" around the smoke that must be included in the pressure update. Thus, in a smoke simulation air cells are assigned a row in matrix $\mathbf{A}$ of equation 13. Vector $\mathbf{B}$ in equation 13 is also changed slightly from the description in equation 14 to

$$b_{i(smoke)} = \frac{\rho h}{\Delta t}(\nabla \cdot \mathbf{u}_i) - e_i p_{atm} \tag{24}$$

where e_i is the number of neighbors of cell i that do not exist in the hashtable grid.

9.2 Smoke Dissipation

We model steam and other mists that evaporate in the air by allowing the smoke density to dissipate over time. At each time step the dissipation is modeled by changing the density according to

$$\rho_s^{new} = max\,(0,\ \ \rho_s - k_{diss}\Delta t) \tag{25}$$

where k_{diss} is the dissipation rate of the smoke.

9.3 Extra Forces in a Smoke Simulation

Besides the alterations to the basic simulator, we add three extra forces to the smoke simulator, *wind*, *buoyancy* and *vorticity confinement*. We will describe each of these forces in turn.

Wind A simple wind force can be added to the simulation by pushing the borders of the air buffer towards a prevailing wind velocity. Mathematically, this is done by computing a weighted average of the cell velocity and the wind velocity at grid cells in the air buffer that are missing one or more neighbors:

$$\mathbf{u}_{new} = (1 - k_{wind}\Delta t)\ \mathbf{u}\ +\ k_{wind}\Delta t\ \mathbf{w} \tag{26}$$

where $\mathbf{w}$ is the prevailing wind velocity, and k_{wind} is a constant that determines how fast $\mathbf{u}$ will move towards the wind velocity.

The Buoyancy Force [Fedkiw et al. 2001] model the tendency of smoke to rise and fall in the air with a simple "buoyancy" force that is calculated using the temperature and density of the smoke. The force is based on the observation that hot smoke tends to rise, and dense smoke tends to fall. Combining these two ideas, they produce the smoke buoyancy model:

$$\mathbf{F}_{buoy} = (k_{rise}(q - q_{atm})\ +\ k_{fall}(\rho_s))\frac{\mathbf{g}}{\|\mathbf{g}\|}. \tag{27}$$

where $\mathbf{F}_{buoy}$ is the buoyancy force, ρ_s is the smoke density, q is the smoke temperature, q_{atm} is the temperature of the surrounding atmosphere, $\mathbf{g}$ is the gravitational force vector, and k_{rise} and k_{fall} are user-defined constants that control the rise and fall of the smoke.

The curl, $(\nabla\times)$**.** The vorticity confinement force, which we will describe shortly, uses yet another application of the gradient operator called the *curl* or *vorticity* of a vector field. The curl of a vector field is defined as the cross product of ∇ and the vector field. The curl of $\mathbf{u}$ is a vector field given by:

$$\nabla \times \mathbf{u} = \left(\frac{\partial u_z}{\partial y} - \frac{\partial u_y}{\partial z},\ \frac{\partial u_x}{\partial z} - \frac{\partial u_z}{\partial x},\ \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}\right). \tag{28}$$

Recall that the cross product of two vectors is a third vector that is perpendicular to the first two. $\nabla \times \mathbf{u}$ is a vector field that is perpendicular to $\mathbf{u}$ at every location in space. The physical interpretation of the new vector field may not be readily apparent, but what it actually describes is how much and in what direction $\mathbf{u}$ is twisting as you follow its flow lines. Hence, the name "curl".

Smoke Simulation Algorithm

New Parameters

k_{wind}	Wind force coefficient.
k_{rise}, k_{fall}	Buoyancy force coefficients.
k_{vort}	Vorticity confinement coefficient.
ρ_s	The smoke density at the grid cell centers.
q	The temperature at the grid cell centers.

Algorithm

1. Calculate the time step, Δt.
2. Update the grid based on the smoke densities (section 9.1).
3. Advance the velocity field, $\mathbf{u}$.
   3a. Apply convection to $\mathbf{u}$.
   3b. Apply buoyancy and wind forces.
   3c. Apply vorticity confinement force.
   3d. Calculate the pressure including the air buffer.
   3e. Apply the pressure.
   3f. -
   3g. Set solid cell velocities.
4. Apply convection to q and ρ_s and dissipate ρ_s.

Figure 13: Modifications to the simulator to animate smoke.

Vorticity confinement. A well known problem with fluid dynamics solutions like the one outlined in section 4 is that they tend to dampen any swirling motions in the velocity field. In other words, the curl of $\mathbf{u}$ diminishes too quickly as the simulation progresses. To counteract the undesired dampening of swirling motions, [Fedkiw et al. 2001] adopt a technique known as *vorticity confinement* from the CFD literature. The idea is to isolate and increase swirling motions in the fluid by amplifying the curl of the velocity field. During a simulation, the vorticity confinement force is added as follows:

- The curl of the velocity field, $\nabla \times \mathbf{u}$, is calculated at the center of each grid cell and stored in the temporary vector for that cell. The curl is calculated by first finding an interpolated velocity on all six faces of the cell. Then central differences across the cell are used to evaluate the terms of 28.
- The magnitude of the curl is calculated at the center of each grid cell, creating a new scalar field, $n = \|\nabla \times \mathbf{u}\|$.
- The gradient direction of n, $\mathbf{N} = \nabla n / \|n\|$, is then calculated using central differences. Note that this is not the same as equation 4, which uses backwards differences.
- A vorticity confinement force is computed at the center of each grid cell as follows:

$$\mathbf{F}_{vort} = k_{vort} \left(\mathbf{N} \times (\nabla \times \mathbf{u})\right). \quad (29)$$

- Finally, the velocity components on all sides of the cell are updated by $\mathbf{u}_{new} = \mathbf{u} + \Delta t \, \mathbf{F}_{vort}$.

Figure 13 summarizes the changes to the basic simulator that are needed to simulate smoke or steam instead of liquids.

9.4 Rendering Smoke

To visualize the evolving smoke field, we use OpenGL to render blurry dots at the centers of grid cells that have a non-zero smoke density. The cells are rendered in back to front in terms of distance from the viewer, and the blurry dots are rendered as billboards that are approximately fifty percent larger than the cell width, so that there will be substantial overlap between them. The transparency of the dots is modulated using alpha blending, with the alpha value for a cell being set to $max(0,\ 1-\rho_s)$. Figure 14 shows images from a smoke simulation.

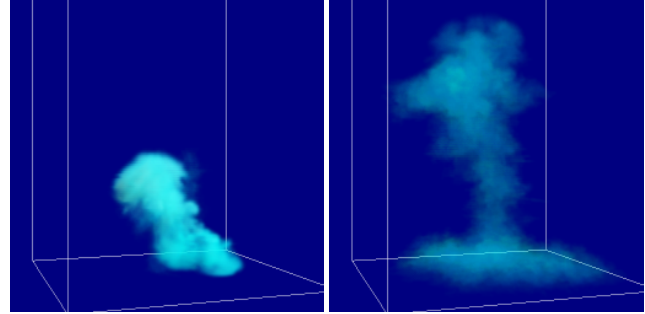


Figure 14: Images from the smoke simulator.

10 Level Sets

Fluids simulated using the Navier-Stokes equations may be broken into essentially two categories: smoke and liquid. To render smoke-like fluids, we usually track the density of the fluid as it flows through its velocity field. Then, using the smoke densities, we can use CSG techniques to convincingly render the volume of the fluid. With liquids, however, we are more interested in displaying and tracking only the surface of the liquid. In addition, we would like the liquid surface to appear similar to liquids in the real world: generally smooth when there's not a lot of movement, but choppy when high velocities are involved.

Probably the most prevalent method for tracking the liquid surface in a fluid flow simulation incorporates a data structure called a *level set*. The use of level sets in fluid flow simulations was introduced by [Foster and Fedkiw 2001]. Since then, almost every advance in computer graphics fluid flow has used some version of a level set to track a liquid surface.

Unfortunately, literature on level sets has a very specialized vocabulary and implementing a level set surface tracking system can be very difficult for a student new to the subject. This section provides the student with a beginner's guide to implementing a surface tracking system, so that liquid flow may be convincingly rendered.

10.1 Overview

Before we start looking at how to use a level set, it will help to have a brief look at what it is and the theory behind it. A level set is essentially a function of higher dimension used to track a curve or surface of lower dimension. For example, a function in 3D may be used to track a curve in 2D. We could write our 3D function like: $z = f(x,y)$. The term *level set* refers to a set of points that all have the same z value. The c-th level set is the curve, whose points satisfy $f(x,y) = c$. Usually, we are only interested in the zero level set of the function. For our system, the zero level set is the liquid surface that we're trying to track. In that case, we use a 4D function to track a zero level set in in 3D.

This discussion raises a valid question: If all we want is the 3D surface, why do we bother with a 4D function? The answer: the extra dimension simplifies complex topological changes in the curve or surface. For example, suppose we have two cones placed

side by side, expanding as z increases (see Figure **??**). At some point in the expansion, the two cones join together into a single surface. Consider what happens as we look at the various level sets of the function. Initially, the two cones appear as single points, then become two separate circles and eventually combine into a single large curve. Using the level set idea, tracking the evolution of the curve is easy.

Now, imagine a different system that explicitly tracks the evolution of the curves as 2D objects. Throughout the curve evolution the system will have to detect when the curves collide and figure out how to combine the two curves into a single large one. The processing and detection of those types of changes can quickly become really complicated. However, with level sets the evolution is much simpler to compute. With our liquid surface, we want the liquid to be able to splash away from and then back onto itself, while maintaining a single surface. So, level sets offer us that property without having to write code specifically to manage it.

10.2 Building a Useful Level Set Function

At its most basic definition, a level set function is simply a function that at some value (typically zero) embeds a desired curve or surface. While this definition is valid, is not particularly useful if we want to alter the position of the curve or surface. One very useful function that makes it easy to update the surface position of the level set is the *signed distance function*.

A signed distance function is simply a scalar distance function (from the curve or surface), but stores negative distances within the curve/surface and positive outside of it. To see the usefuleness of the signed distance function, consider one of the cones in Figure **??**. For that cone, suppose our zero level set is a circle midway through the cone. Notice that the sides of the cone recede uniformly and smoothly away from the zero level set curve. So, if we wish to perturb a small portion of the level set curve, we can essentially just translate a portion of the cone down to the zero level set. This is analogous to moving the function in the direction of the function's gradient. When we update a level set curve, we use the function's surface gradient to perturb the curve. Thus, to reduce errors in our update it is very useful to have a function that is smoothly differentiable. A signed distance function fulfills that property and is easily conceptualized.

In order to create a signed distance function for a curve or surface, we want to use an accurate, yet efficient method to build it. We could use a chamfer map technique to build the distance field, but that is probably not accurate or smooth enough. Instead, we'll use a method developed specifically for level sets called the *fast marching method*. Since the direct implementation of this method is not very explicitly outlined in most texts, Appendix **??** contains specific details on implementing this method to build a signed distance function.

10.3 Moving Under an External Velocity

One important operation on level sets is the movement of the function under some external velocity. In the fluid flow case, that velocity is supplied by the velocity field embedded in the MAC grid. While we won't derive the equation for updating the level set under the velocity field, we will describe its parts and how to use it. The equation for updating the level set is given by

$$\frac{\partial \phi}{\partial t} + \mathbf{u}(x,y,z) \cdot \nabla \phi = 0. \tag{30}$$

This equation is also called the simple convection (or advection) equation. While this expression may appear daunting, it actually works out to be pretty simple. First, the function that embeds our level set is denoted by ϕ, which is actually short for $\phi(x,y,z)$–this function is a scalar field in 3D. Then, $\frac{\partial \phi}{\partial t}$ is the amount which we multiply by Δt (more on this later) in order to update the level set. You have already been introduced to $\mathbf{u}(x,y,z)$, which is the fluid velocity field. Finally, $\nabla \phi$ is a vector field as shown in (4).

Before we continue, we should note that (30) excludes a lot of information about performing the surface update. First, notice that the expression contains two types of derivatives: spatial and temporal. The spatial derivatives are in the $\nabla \phi$ term and the temporal derivative is the $\frac{\partial \phi}{\partial t}$ term. Furthermore, those derivatives have not been explicitly discretized for us; i.e., the equation does not describe at all how the derivatives in it should be computed numerically.

While this may not seem like a very important point at first glance, it turns out to be extremely important in this case. The way that the derivatives are discretized ultimately affects whether the surface will evolve stably or not. In other words, if we calculate the derivatives in certain ways, the surface will usually fall apart after a few iterations.

10.4 Approximating Derivatives for Level Set Updates

Since derivative computations are so important in doing a level set update, a brief discussion of them is in order. Recall that the level set update combines two kinds of derivatives: spatial and temporal. In this section we will discuss the techniques that are applied to each derivative type.

For the spatial derivatives of (30), we must employ a special discretization called an *upwind difference*. Upwind differences use velocity field information to choose between backward and forward differences. If the velocity field is positive at the point we want to update, we use a backward difference; if it is negative we use a forward difference. If we do not use an upwind difference for $\nabla \phi$ in (30), the surface is certain to become unstable.

Generally, higher-order upwind differencing has been used in past liquid surface tracking systems. These higher-order methods use various forms of the Taylor series expansion for a function in order to improve the accuracy of the calculated derivatives. There are several specific schemes that are typically used, among them ENO and WENO (essentially non-oscillatory and weighted essentially non-oscillatory) are the most common. However, the cost of using these techniques is increased computation time. Fortunately, a new and more accurate method was recently developed [**?**] that avoids the use of higher-order upwind differences and this method (discussed in section 10.7) is recommended for both its speed and simplicity.

For the temporal derivative, $\frac{\partial \phi}{\partial t}$, in (30), most systems employ a higher-order Runge-Kutta interpolation technique like that discussed in section **??**. Most temporal discretization methods require third-order Runge-Kutta to advance the surface in the velocity field accurately. However, the new method that we will discuss in section 10.7 uses a simple first-order Euler timestep (i.e., $\phi^{n+1} = \phi^n + \Delta t \frac{\partial \phi}{\partial t}$) to advance the level set surface.

10.5 Reinitialization

After performing a level set update (using the appropriate spatial and temporal discretizations) an additional step is required before we may proceed to the next iteration. When the level set surface is updated under the fluid velocity field, that field defines the motion of specifically the zero level set and not any of the other level sets in the level set function. Using the entire fluid velocity on all of the level sets, does adjust the zero level set correctly, but typically causes the other level sets embedded in the function to bunch up or to flatten out. Eventually, these small adjustments cause the surface

of our level set to become less smooth, which can add error to the update of the zero level set's position.

The remedy to this bunching and flattening problem is to reinitialize the level set function to a signed distance function around the zero level set of the function. This reinitialization step is quickly accomplished by finding the zero level set and then rebuilding the signed distance function around the zero level using the fast marching method (see Appendix **??**).

10.6 Data Structures

Now that we have an idea of what level sets are, we should discuss the best way to represent them. One possibly wasteful way that we could represent the level set function's scalar field, would be to store the entire scalar field in a single, static, 3D array. This organization has several disadvantages: it is both space and time inefficient and restricts the liquid flow to a single static area.

A much better way of storing the level set would be to use the hashtable method discussed in section 3.1. Also, because we are only interested in tracking the surface of the liquid, we only need a narrow band of scalar field entries around the liquid surface (usually six to seven entries is sufficient). This hashtable method allows our liquid surface to move freely in our virtual environment and is both time and memory conserving. In addition, if we've already implemented the hash table for the MAC grid, it should be a very simple matter to adjust it for the level set function grid.

10.7 Fast Particle Level Set Method

While the advantages of using a level set are clear, there are still some difficulties with using the level set method. Because we are using a discrete scalar field to represent a continuous function, our method is subject to aliasing errors. As a result of the discretization level set surfaces tend to lose significant volume when moving through a velocity field. Especially where the surface becomes very thin.

A method introduced in [Enright et al. 2002] that alleviates this problem is the Particle Level Set Method. This method tracks the liquid surface as a level set coupled with marker particles along the liquid surface. Marker particles are placed on both sides of the surface and are marked as positive or negative, indicating which side of the surface the particle is tracking. Each marker particle also has a radius, such that the particle is tangent to the surface.

To update the surface, both the particles and the surface move through the velocity field and the particles are used to repair errors introduced into the liquid surface through the update. [**?**] combines the particle level set method with a fast, first-order surface update and shows that there is little need for higher-order upwind differences when particles are available to correct errors. Because of its simplicity, accuracy and speed, we will design our system using almost exactly the method described in [**?**].

10.8 Putting it All Together

Now that we've discussed the relevant considerations for tracking the level set surface, we can design an effective system for managing our liquid surface. If we view our surface tracking system as a black box from the outside, it has essentially two main functions: initialization and update of the surface. So, we'll proceed with the details of implementing these two functions.

10.8.1 Surface Initialization

The surface initialization step has a solitary goal: to build a narrow band of a signed distance function around the desired isosurface. In order to do this, we need to somehow be able to identify the isosurface we're interested in. This may be done in several different ways, but the net result of the operation should be a list of grid points bordering the desired isosurface. This list will be the input into the fast marching method, which we'll use to build the signed distance function.

One way that we might identify the isosurface would be to start with an implicit function, like that of a sphere. Then, by sampling the implicit function over a certain volume at the level set grid points, we could identify the grid points neighboring the zero isosurface and add those to our list. The resulting list of points should, however, form a closed surface for the fast marching method to work.

After collecting the necessary initial points, we then build the signed distance function up to a certain distance away using the fast marching method. Since this method is outlined explicitly in Appendix **??**, it will not be repeated here.

Once the narrow band of grid points has been initialized to a signed distance function, we randomly place marker particles into volume of the narrow band around the isosurface. These particles are used later on in the fast particle level set method (as discussed in section 10.7). Each randomly placed particle tracks both its position and its radius. The radius of each particle, r_p, is determined using the following piecewise function:

$$r_p = \begin{cases} r_{max} & \text{if } |\phi(\vec{x_p})| > r_{max} \\ |\phi(\vec{x_p})| & \text{if } r_{min} \leq |\phi(\vec{x_p})| \leq r_{max} \\ r_{min} & \text{if } |\phi(\vec{x_p})| < r_{min} \end{cases} \quad (31)$$

where $\phi(\vec{x_p})$ is the tri-linearly interpolated value of the level set function at the particle position $\vec{x_p}$. r_{min} and r_{max} are the minimum and maximum radii of the particles, set to $0.1h$ and $0.5h$, where h is the minimum spacing between grid points in the level set function. Recall that particles are marked as either negative or positive, depending on the side of the interface on which they are initialized. To track this attribute, we use the sign bit on the particle radius to indicate the particle side—i.e., for negative particles we store $-r_p$ and for postive particles we store $+r_p$.

Once the particles and signed distance function have been initialized, the surface initialization process is complete and the level set surface is ready to be updated according to the liquid velocity field.

10.8.2 Updating the Level Set Surface

The following paragraphs outline the details and order of the operations that need to be performed to update the level set surface. Again, most of this process is taken from [**?**].

1. Update the Level Set Function — In this step, we do a first-order accurate update of the level set function, which is represented in a narrow band around the liquid isosurface. When we do the update, we exclude the outside border of the narrow band. So, if our narrow band is six grid points wide on each side of the isosurface, we only perform the update up to five grid points from the surface.

The surface update is performed using the following equations:

$$\begin{aligned} \phi^{n+1}_{i,j,k} = (\; & \alpha\beta\gamma\phi^n_{r+1,s+1,t+1} \\ & +(1-\alpha)\beta\gamma\phi^n_{r,s+1,t+1} \\ & +\alpha(1-\beta)\gamma\phi^n_{r+1,s,t+1} \\ & +\alpha\beta(1-\gamma)\phi^n_{r+1,s+1,t} \\ & +(1-\alpha)(1-\beta)\gamma\phi^n_{r,s,t+1} \\ & +(1-\alpha)\beta(1-\gamma)\phi^n_{r,s+1,t} \\ & +\alpha(1-\beta)(1-\gamma)\phi^n_{r+1,s,t} \\ & +(1-\alpha)(1-\beta)(1-\gamma)\phi^n_{r,s,t} \;), \end{aligned} \quad (32)$$

where

$$r = i - \left\lceil u_{i,j,k} \frac{\Delta t}{\Delta x} \right\rceil, \quad \alpha = \frac{(i-r)\Delta x - u_{i,j,k}\Delta t}{\Delta x}, \tag{33}$$

$$s = j - \left\lceil v_{i,j,k} \frac{\Delta t}{\Delta y} \right\rceil, \quad \beta = \frac{(j-s)\Delta y - v_{i,j,k}\Delta t}{\Delta y}, \tag{34}$$

$$t = k - \left\lceil w_{i,j,k} \frac{\Delta t}{\Delta z} \right\rceil, \quad \gamma = \frac{(k-t)\Delta z - w_{i,j,k}\Delta t}{\Delta w}, \tag{35}$$

and $\vec{u}(\vec{x}_{i,j,k}) = (u_{i,j,k}, v_{i,j,k}, w_{i,j,k})$–i.e., $(u_{i,j,k}, v_{i,j,k}, w_{i,j,k})$ are the components of the interpolated velocity at the level set grid point (i, j, k).

While these expressions appear complicated, (32) simply boils down to a linear interpolation of the level set function combined with the velocity field values. When all has been computed, $\phi^{n+1}_{i,j,k}$ is the new grid point value at indices (i, j, k) in the narrow band. In order to correctly produce the updated level set function, a temporary copy of the narrow band must be made to draw values from. At the end of this step, the old level set function is replaced with the newly computed values.

2. Advance the Marker Particles Once the level set function has been advanced, we advance the marker particles that we placed around the zero level set. These marker particles are advanced using RK2, exactly as described in equation 12 (see Step 3a of Section 4).

3. Correct Level Set Errors using the Marker Particles

11 Conclusion

References

CARLSON, M., MUCHA, P. J., VAN HORN, III, R. B., AND TURK, G. 2002. Melting and flowing. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, ACM Press, 167–174.

ENRIGHT, D., MARSCHNER, S., AND FEDKIW, R. 2002. Animation and rendering of complex water surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 736–744.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 15–22.

FOSTER, N., AND FEDKIW, R. 2001. Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 23–30.

GOKTEKIN, T. G., BARGTEIL, A. W., AND O'BRIEN, J. F. 2004. A method for animation viscoelastic fluids. In *Siggraph 2004 (Update Reference)*.

HARLOW, F. H., AND WELCH, J. E. 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. *The Physics of Fluids 8*, 2182–2189.

STAM, J. 1999. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 121–128.

STEELE, K., CLINE, D., AND EGBERT, P. 2004. Modeling and rendering viscous liquids. In *Computer Animation and Social Agents (CASA2004) Update Reference*.

TESCHNER, M., HEIDELBERGER, B., MUELLER, M., POMERANETS, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, Visualization VMV'03*, 47–54.

WORLEY, S. 1996. A cellular texture basis function. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 291–294.