Project Report

STUDY AND DESIGN ON FLUID SIMULATION FOR COMPUTER GRAPHICS

CSE3016 – Computer graphics and multimedia

Submitted by

17BCE0756– Priyansh Agarwal

17BCE2408 – RACHIT JAIN

Under the guidance of

Prof. GLADYS GNANA KIRUBA B - SCOPE

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computing Science and Engineering

26th October 2018

Contents

1.Introduction

In this project we will be seeing and working on the simulation of the fluid. In the context of physics, the word fluid may mean something different than you might usually think

.

Fluid simulation for computer graphics is a special part of Computational Fluid Dynamics (CFD) which is used in graphics applications to generate realistic representations of different types of fluids such as water, smoke etc. Fluid can be simulated from 2 viewpoints Lagrangian *or Eulerian. In the Lagrangian viewpoint, we simulate the fluid as discrete blobs of fluid. Each particle has various properties, such as mass, velocity, etc. The benefit of this approach is that conservation of mass comes easily. The Eulerian viewpoint, on the other hand tracks fixed points inside of the fluid. The Eulerian approach corresponds to grid based techniques.*

In this section we describe an alternative approach to fluid simulation. Here we discuss Lagrangian techniques. In this approach, we have a set of discrete particles that move through space to represent our fluid

2)Algorithm:

The Python implementation has the following algorithm using pygame library:

- Initialize all particles
- Set t = 0
- Choose a Δt
- for i from 0 to n for j from 1 to numparticles

Get list Lj of neighbors for Pj

Calculate Densityj for Pj using Lj

Calculate Pressurej for Pj using Lj

CalculateaccelerationAj forPj usingDensityj and Pressurej

Move Pj using Aj and Δt using Euler step

t = t + Δt

- Cleanup all data structures
- Exit

In starting we initialize the values of these but while running the program we can change these accordingly.

3)Overview of the Work

3.1) Problem description:

We are stimulating fluids using the particle based implementation. Here we can change the density ,mass gravity effect on each particle taking each one as individual. Thus we have to implement lagrangian method of particle.

3.2)Software Requirements: python only, with requied library installed

3.2) Hardware Requirements: Computer system

4) Future Directions:

This approach has many pros and cons compared to grid based techniques. In general, particle based approaches are less accurate than their grid based counterparts. This is primarily due to the difficulties in dealing with spatial

derivatives on an unstructured particle cloud. However, particle based simulations are typically much easier to program and understand. Furthermore, particle based techniques are much faster, and can be used in real time applications such as video games.

5)Conclusion:

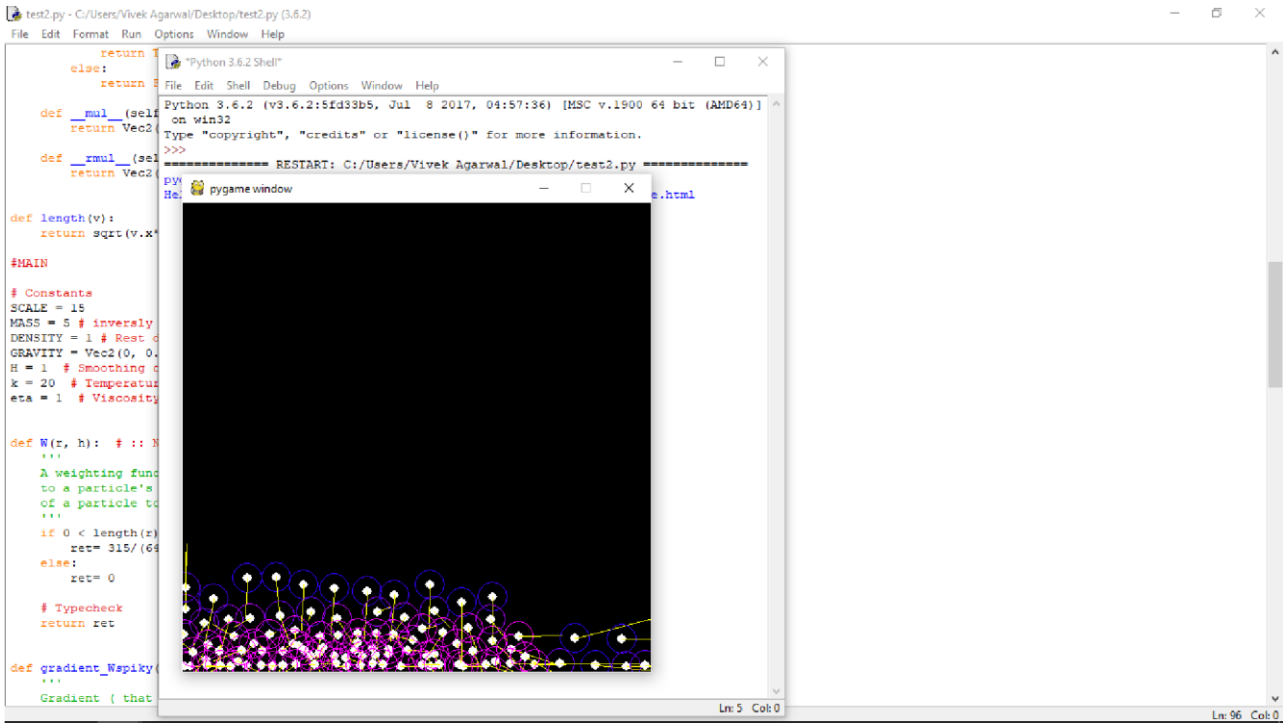
We described a particle-based method for simulation fluid motion. It is based on a particle discretization of the differential operators to solve the Navier-Stokes equations. It is suited for simulating a wide variety of fluid flows including multifluids, multiphase flows and medium scale problems such as the corridor flood. Because of the Lagrangian nature small in areas where large deformations of the interface occurs.

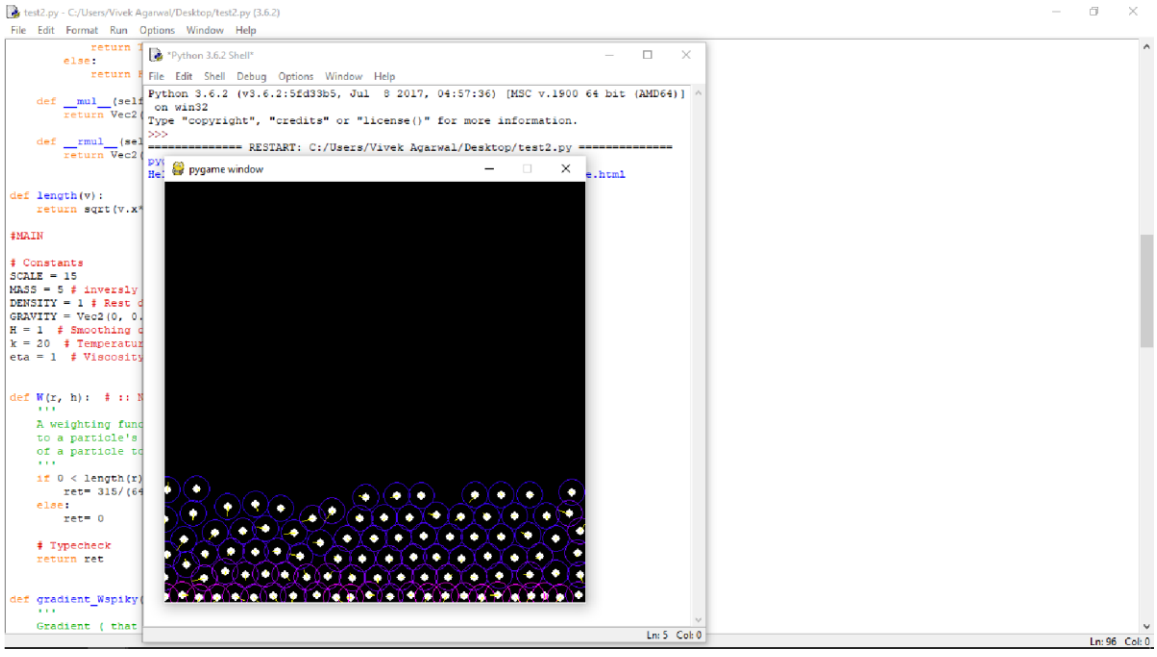
6) References

[1] https://en.wikipedia.org

[2] http://www.sci.utah.edu/~tolga/pubs/ParticleFluidsHiRes.pdf

7) Output snaps





8)Appendix – A

BASE PAPER

Fluid Simulation For Computer Graphics: A Tutorial in Grid Based and Particle Based Methods

Colin Braley[1] Virginia Tech

Adrian Sandu[2] Virginia Tech

Figure 1: *Fluid Simulation Examples*

Abstract

In this paper we present a tutorial on the implementation of both a grid based and a particle based fluid simulator for computer graphics applications. Many research papers on fluid simulation are readily available, but these papers often assume a very sophisticated mathematical background not held by many undergraduates. Furthermore, these papers tend to gloss over the implementation details, which are very important to people trying to implement a working system.

Recently, Robert Bridson release the wonderful book, "Fluid Simulation for Computer Graphics.[Bridson 2009]" We base a large portion of our own grid-based simulator off of this text. However, this text is very dense and theory intensive, and this document serves as easy version for those who want to implement a simulator quickly. Furthermore, Bridson's text does not cover particle based methods, like SPH, which are quickly becoming commonplace within the graphics community. This work provides an introduction to SPH as well.

Keywords: Fluids, Physically Based Animation, SPH, Grid Based

1 Introduction

2 Introduction

In the context of physics, the word fluid may mean something different than you might usually think. In physics, fluids fall into two categories *incompressible* and *compressible* flow. Incompressible flow is a liquid, such as water or juice. Compressible flow, on the other hand, corresponds to gas such as air or steam. Compressible flow is called compressible, because you can easily change the volume of this fluid. Note that there is no such thing as 100 percent incompressible fluid. All fluids, even water can change volume to some degree. If they could not, there would be no way to audibly yell under water. However, we simply choose to ignore compressibility in fluids like water that are nearly incompressible, and instead we refer to them simply as incompressible.

There are many, many ways to simulate fluids. In graphics, the most common two techniques are grid based simulations, and particle based simulations. (Very recently, new techniques such as the Lattice-Boltzmann method have been introduced to graphics, but they are beyond the scope of this paper.) Grid based simulations are typically highly accurate, although relatively slow compared to particle based solutions. Particle based simulations are usually much faster, but they typically do not look as good as grid based simulations.

Some readers may not know the difference between grid based and particle based simulations. The best description of this can be found in page 6 of [Bridson 2009]. I will attempt to paraphrase this description here.

Fluid can be simulated from 2 viewpoints, *Lagrangian* or *Eulerian*. In the *Lagrangian* viewpoint, we simulate the fluid as discrete blobs of fluid. Each particle has various properties, such as mass, velocity, etc. The benefit of this approach is that conservation of mass comes easily. The *Eulerian* viewpoint, on the other hand tracks fixed points inside of the fluid. At each fixed point, we store quantities such as the velocity of the fluid as it flows by, or the density of the fluid as it passes by. The *Eulerian* approach corresponds to grid based techniques. Grid based techniques have the advantage of having higher numerical accuracy, since it is easier to work with spatial derivatives on a fixed grid, as opposed to an unstructured cloud of particles. However, grid based techniques often suffer from mass loss, and are often slower than particle based simulations. Finally, grid based simulations often do

[1] e-mail: cbraley@vt.edu

[2] e-mail:sandu@cs.vt.edu

much better tracking smooth water surfaces, whereas particle based approaches often have issues with these smooth surfaces.

3 Governing Equations

Here we will describe the governing equations for fluid motion, and also describe some of the special notation used in fluid simulation literature. For someone only interested in a basic implementation, this section can be skimmed with the exception of the final paragraph on notation. However, for a full understanding of the *why* in fluid simulation, this section should be read in full.

First, we will describe the general incompressible Navier Stokes equations. We will attempt to intuitively describe the vector calculus operators involved, and those without a knowledge of vector calculus may wish to consult the appendix of [Bridson 2009] for review. Here are the incompressible Navier Stokes equations:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} + \frac{1}{\rho}\nabla p = \vec{F} + \nu \nabla \cdot \nabla \vec{u} \quad (1)$$

$$\nabla \cdot \sim u = 0 \quad (2)$$

In these equations, $\sim u$ is fluid velocity. The time variable is t The density of the fluid is represented by ρ. For water, $\rho \approx 1000\ \frac{kg}{m^3}$. The pressure inside the fluid (in force units per unit area) is represented by p. Note that p and ρ are different things(the first is the Greek letter *rho* while the second is *p*). Body forces (usually just gravity) are represented by $F\sim$. Finally, ν is the fluid's coefficient of *kinematic viscosity*.

In our simulator, we currently don't take fluid viscosity into account. For inviscid fluids like water, viscosity does usually not play a large role in the look of an animation. However, if you wish to include viscosity, see chapter 8 of [Bridson 2009]. The fundamental work on viscous fluids in graphics is [Goktekin et al. 2004], and can be taken as a starting point for implementation.

When the viscosity term is dropped from the incompressible Navier Stokes equations, we get the following set of equations:

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p = \vec{F} \quad (3)$$

$$\nabla \cdot \sim u = 0 \quad (4)$$

These equations are simpler, and they are the equations we will consider for the rest of this paper. These are called the *Euler Equations*.

Lastly, we will quickly discuss some unique notation used in fluid simulation. In simulation, we take small discrete time steps in order simulate some phenomenon. Consider the velocity field $\sim u$ being simulated. In a grid based simulation, we store $\sim u$ as a discreetly sampled vector field. However, we need some type of notation to describe which grid element we are referring to. To do this, we use subscripts like the following $\sim u_{a,b,c}$ to refer to the vector at a,b,c (Note that our indexing scheme is actually more complicated, but this is discussed in the beginning of the grid based simulation section.) However, we also need a way to indicate which timestep we are referring to. To do this, we use superscripts, such as $\sim u^{\mathbf{k}}_{a,b,c}$. The previous equation would indicate the velocity at index a,b,c at timestep k. While some would consider this an egregious abuse of notation, this syntax is extremely convenient in fluid simulation. In order to maintain clarity, we will explicitly state when we are raising a quantity to a power (as opposed to indicating a timestep). Furthermore, timesteps are written in bold ($a^{\mathbf{b}}$), whereas exponents are written in a regular script (a^b).

4 Grid Based Simulation

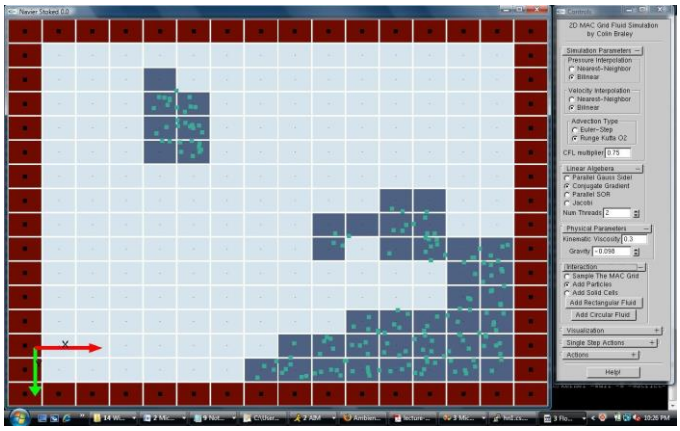
4.1 Overview

Here we will present a high level version of the algorithm for grid based fluid simulation, assuming one wants to simulate n frames of animation.

1. Initialize Grid with some Fluid
2. for(i from 1 to n) Let t = 0.0

   While $t < t_{frame}$

   Calculate Δt



   Advect Fluid

   Pressure Projection (Pressure Solve)

   Advect Free Surface

   $t = t + \Delta t$

   Write frame i to disk

Figure 2: *Our 2D Eulerian Solver*

4.2 Data Structures

While we have discussed Lagrangian vs. Eulerian viewpoints, we have yet to define what exactly we mean by "grid" in grid based simulation. Throughout the simulation, we must store many different quantities (velocity, pressure, fluid concentration, etc.) at various points in space. Clearly, we will lay them out in some form of regular grid. However, not just any grid will do. It turns out that some grids work much better than others.

Most peoples intuition is to go with the simplest approach: store every quantity on the same grid. However, for reasons which we

will soon discuss, this is not a good approach. Back in the 1950's the seminal paper [Harlow and Welch 1965a] by Harlow and Welch developed the innovate *MAC Grid* technique. (Note that MAC stands for Marker-and-Cell.) Among many things, this paper developed a new way to track liquid movement through the grid, called marker particles, and a new type of grid, a staggered grid. Marker particles are still used in some simulators for their simplicity, but they are no longer state of the art. However, the staggered grid developed by Harlow and Welch is still used in many, many simulators.

This grid is called staggered because it stores different quantities at different locations. In two dimensions, a single cell in a MAC Grid might look as follows:

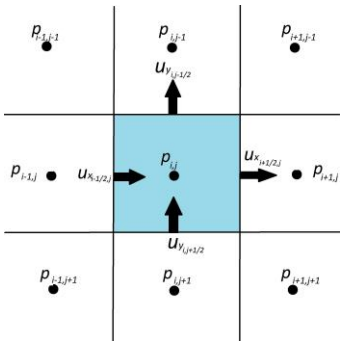


Figure 3: *Two Dimensional MAC Cell*

In three dimensions, a MAC cell would look like this:

Note that in these images p represents pressure, and ~u is velocity. We see in these images that pressure is stored in the center of every grid cell, while velocity is stored on the faces of the cells. Note that these velocity samples are the normal component of the velocity at

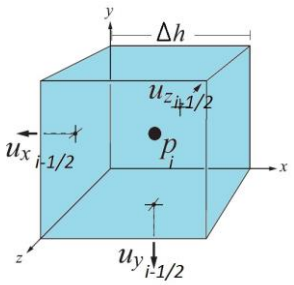


Figure 4: *Three Dimensional MAC Cell*

each cell face. We will now describe why the grid is arranged in this manner.

Consider a quantity w sampled at discrete locations $w_0, w_1, \ldots w_{i-1}, w_i, w_{i+1}, \ldots w_{n-1}, w_n$ along the real line. Imagine we want to estimate the derivative at some sample point i. We must use central differences of some form. Immediately, one can see that:

$$\frac{\partial w}{\partial x}_i \approx \frac{w_{i+1} - w_{i-1}}{2\Delta x} \tag{5}$$

Recall from numerical analysis that this is $O(\Delta x^2)$ accurate. However, there is clearly a huge problem with this equation. It ignores the actual value of w at w_i! We clearly need a way to estimate this derivative without ignoring the actual value at the point which we are trying to estimate. We could forward or backward differences, but these are biased and only $O(\Delta x)$ accurate. Instead, we choose to stagger our grid to make these accurate central differences work. Our staggered central difference looks like this:

$$\frac{\partial w}{\partial x}_i \approx \frac{w_{i+\frac{1}{2}} - w_{i-\frac{1}{2}}}{2\Delta x} \tag{6}$$

This formula is still $O(x^2)$ accurate.

It turns out that, later on in our *Pressure Projection* stage, this staggered grid is very useful, as it allows us to accurately estimate certain derivatives that we require.

However, this staggered grid is not without drawbacks. In order to evaluate a pressure value at an arbitrary point which is not an exact grid point, trilinear interpolation (or bilinear in the 2D case) is required. If we want to evaluate velocity *anywhere* in the grid, a separate trilinear interpolation is required for each component of the velocity! Therefore, in 3D we need to to 3 trilinear interpolations, and in the 2D case we need to do 2 bilinear interpolations! Clearly, this is slightly unwieldy. However, the added accuracy makes up for the complications in interpolation.

The above notation with half-indices is clearly useful since it greatly simplifies our formulae. However, it is obvious that halfindices can't be used in an actual implementation. [Bridson 2009]
recommends the following formulae for converting these halfindices to array indices for a real system:

$$p[i][j][k] = p_{i,j,k} \tag{7}$$

$$u_x[i][j][k] = u_{i-\frac{1}{2},j,k} \tag{8}$$

$$u_y[i][j][k] = v_{i,j-\frac{1}{2},k} \tag{9}$$

$$u_z[i][j][k] = w_{i,j,k-\frac{1}{2}} \tag{10}$$

Therefore, for a grid of n_x, n_y, n_z cells, we store the pressure in a n_x, n_y, n_z array, the x component of the velocity in a n_x+1, n_y, n_z array, the y component of the velocity in a n_x, n_y+1, n_z array, and the z component of the velocity in a n_x, n_y, n_z+1 array.

4.3 Algorithm

4.3.1 Choosing a Timestep

When simulating fluids, we want to simulate as fast as possible without losing numerical accuracy. Therefore, we want to chose a timestep that is as large as possible, but not large enough to destabilize our simulation. The CFL condition helps us do this. The CFL says to chose a value of Δt small enough so that when any quantity is moved from the center of some cell through the velocity field, it will only move Δh distance. This makes sense intuitively, seeing as if a particle was allowed to move in any larger amounts than this you would effectively be ignoring some parts of the velocity field. Therefore, our equation for Δt is as follows:

$$\Delta t = \frac{\Delta h}{\vec{u}_{max}} \tag{11}$$

As you can see, this requires us to know the maximum velocity in the velocity field at any given time. There are 2 ways to get this value, either by doing a linear search through all of the velocities, or by keeping track of the maximum velocity throughout the simulation. These details are discussed in the Implementation section. In computer graphics, we are often willing to sacrifice strict numerical accuracy for the sake of increased computational speed. In many situations, a practitioner is not worried about if the fluid being simulated is one-hundred percent accurate. Instead, we want plausible looking results. Therefore, in many applications you can get away with using a timestep larger than that prescribed by the CFL condition. For instance, in [Foster and Fedkiw 2001], the authors were able to use a timestep 5 times bigger than that dictated by the CFL condition. Either way, it is good practice to let the user be able to scale the CFL based timestep by a factor of their choice, k_{CFL}. In this case, our equation becomes:

$$\Delta t = k_{CFL} \frac{\Delta h}{\vec{u}_{max}} \tag{12}$$

$$f(v, v_{max}, v_{min}) = \kappa_{min}(v - v_{min}) \frac{\kappa_{max} - \kappa_{min}}{v_{max} - v_{min}} \tag{13}$$

In [Bridson 2009], a slightly more robust treatment of the CFL condition is presented. In this text, Bridson suggests a modification where $\sim u_{max}$ is calculated with:

$$\sim u_{max} = \max(|\sim u|) + \sqrt{\Delta h |F\tilde{}|} \tag{14}$$

where $F\tilde{}$ is whatever body forces are to be applied (usually just gravity), and $\max \sim u$ is simply the largest velocity value currently on the grid. This solution is slightly more robust in that it takes into account the effect that the body forces will have on the simulation's current timestep.

4.3.2 Advection

Central to any grid based method is our ability to advect both scalar and vector quantities through our simulation grid. Advection can be informally described as follows: "Given some quantity Q on our simulation grid, how will Q change Δt later?" More formally, we can describe advection as:

$$Q^{n+1} = advect(Q^n, \Delta t, \frac{\partial Q}{\partial t}^n) \tag{15}$$

In this section we will develop a computational function for $advect(Q^n, \Delta t, \frac{\partial Q}{\partial t}^n)$.

Consider a P on our simulation grid. Using our central differencing schemes described previously, we can trivially calculate $\frac{\partial Q}{\partial t}$. Using this derivative, along with our grid information, we can develop a technique to advect quantities through the grid. This technique sometimes called a *backwards particle trace*. Since we are using a particle, this is also commonly referred to as *Semi-Lagrangian Advection*. It is important to note that no particle is ever created, and the particle is purely conceptual. This is what leads to the *Semi* in *Semi-Lagrangian Advection.*

Note that our algorithm can not be done in place, and requires an extra copy of the pertinent data in our simulation grid. Our algorithm works as follows:

1. For each grid cell with index *i,j,k*

   Calculate $-\frac{\partial Q}{\partial t}$

   Calcluate the spatial position of $Q_{i,j,k}$, store it in $X\tilde{}$

   Calculate $X\sim_{prev} = X\sim - \frac{\partial Q}{\partial t} * \Delta t$

   Set the gridpoint for Q^{n+1} that is nearest to $\vec{X}_{prev}$ equal to $Q_{i,j,k}$
2. Set $Q = Q^{n+1}$

This algorithm is very simple, and fairly accurate. However, if you are familiar with numerical analysis, you will recognize that this algorithm uses the simple time integrator *Forward Euler*. This integrator is not very accurate. We recommend at *least* using an integrator such as RK2 or better (Runge Kutta Order-2). In our simulator, we tested out 5 different integrators, and found the following $O(h^3)$ accurate scheme to work the best:

$$\kappa_1 = f(Q^n) \tag{16}$$

$$\kappa_2 = f(Q^n + \frac{1}{2}\Delta t \kappa_1) \tag{17}$$

$$\kappa_3 = f(Q^n + \frac{3}{4}\Delta t \kappa_2) \tag{18}$$

$$Q^{n+1} = Q^n + \frac{2}{9}\Delta t \kappa_1 + \frac{3}{9}\Delta t \kappa_2 + \frac{4}{9}\Delta t \kappa_3 \tag{19}$$

We will use this advection scheme throughout our simulator. One common use is advecting fluid velocity itself. Another use is advecting temperatures or material properties in advanced simulators.

A careful reader might have noticed one issue with the advection psuedocode. How would we perform an advection for a boundary cell? This requires extrapolation for our MAC grid. In our experience, simply clamping grid indices is fine in the advection code, but more advanced techniques do exist. However, we have found that in practice these advanced extrapolation techniques do little to visually augment the simulation.

4.3.3 Pressure Solve

So far, we have done nothing to deal with the incompressbility of our fluids. In this section, we will develop a numerical routine such that our fluid satisfies both the incompressibility condition:

$$\nabla \cdot \sim u^{n+1} = 0 \tag{20}$$

as well as our boundary conditions:

$$\sim u_{n+1} \cdot \hat{n} = \sim u_{solid} \cdot \hat{n} \quad (21)$$

Additionally, this section finally allows us to show the reason for our staggered MAC grid discussed in our previous section. First, we will work out the individual equations to make a single grid cell satisfy our two conditions above. Then, we will show how this information can be used to make the entire grid incompressible.

Consider a 2D MAC cell at location *i,j*. Per the Euler equations, on every step we must update our cells velocity by the following equations. First, we present them in 2D where our velocity is represented by $\sim u =< u,v >$.

$$\vec{u}^{n+1}_{i+\frac{1}{2},j} = \vec{u}^{n}_{i+\frac{1}{2},j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta h} \quad (22)$$

$$\vec{v}^{n+1}_{i,j+\frac{1}{2}} = \vec{v}^{n}_{i,j+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1} - p_{i,j}}{\Delta h} \quad (23)$$

Here are the equivalent equations in 3D for $\sim u =< u,v,w >$.

$$\vec{u}^{n+1}_{i+\frac{1}{2},j,k} = \vec{u}^{n}_{i+\frac{1}{2},j,k} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta h} \quad (24)$$

$$\vec{v}^{n+1}_{i,j+\frac{1}{2},k} = \vec{v}^{n}_{i,j+\frac{1}{2},k} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta h} \quad (25)$$

$$\vec{w}^{n+1}_{i,j,k+\frac{1}{2}} = \vec{w}^{n}_{i,j,k+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta h} \quad (26)$$

Just in case these don't seem complicated enough already, there is another issue that must be attended to. These equations are only applied to *components* of the velocity that border a grid cell that contain fluid. Getting these conditions correct was one of the hardest things in the actual programming of our simulation, and we recommend that an implementor try to first program this in 2D for easier debugging.

A careful reader will also notice that these equations may require the pressures of grid cells that lie either outside of the grid, or

$$(\nabla \cdot \vec{u})_{i,j,k} \approx \frac{+_{\frac{1}{2}} \quad -_{\frac{1}{2}}}{\Delta h} + \frac{+_{\frac{1}{2}} \quad -_{\frac{1}{2}}}{\Delta h} + \frac{+_{\frac{1}{2}}}{\Delta h} \quad (29)$$

outside of the fluid. Therefore, we must specify our boundary conditions.

There are two primary types of boundary conditions in grid based simulation, *Dirichlet* and *Neumann*. We will use Dirichlet conditions for free surface boundaries, indicating that we will specify the value of the quantity at and boundary case. Therefore, we simply assume that pressure is 0 in any region of air outside of the fluid.

The more complicated boundary is with solid walls. Here we will use a *Neumann* boundary condition. Using the above pressure update equations, we substitute in the solids velocity (0 for simulations without moving solids), and then we arrive at a single linear equation for our pressure. Rearranging our terms allows us to solve for the pressure.

Now we will work out how to make our fluid incompressible. This means that, for every velocity component on the grid, we want to satisfy:

$$\nabla \cdot \sim u = 0 \quad (27)$$

Note that this divergence operator can be expanded to:

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \quad (28)$$

Therefore, it is clear that we simply want to find a way so that each component of our spatial derivatives equals zero. Recalling our central differences used earlier, we can approximate these divergences using the following numerical routines, which use our finite differences developer earlier:

Finally, we have all the quantities necessary for our pressure update. We have developed equations for how the pressure affects the velocity, and we also have numerical equations to estimate the pressure gradient. Using this information, we can create a linear equation for the new pressure in every grid cell. We can then combine these equations together into a system of simultaneous linear equations which we can solve for the whole grid, and finally complete our pressure update.

we should store is as a *sparse matrix*. Each row has at most 4 nonzero entries in the 2D case, and 6 non-zero entries in the 3D case.

Furthermore, is is clear that A is symmetric. Every entry at index *i,j* that is not on the main diagonal is defined by $\beta_{i,j}$. Through our definition of β, it is clear that $\beta_{i,j} = \beta_{j,i}$. Therefore, we only need to store half of the entries in A. We use the following scheme for storing our matrix. We store a main linked list, sorted first by column, then by row. We store in each linked list node the following $< i,j,p_{ij} >$. The column index is stored as *i*, the row index is stored by *j*, and the corresponding pressure value is stored as *p*. This storage scheme has many pros and cons. We are storing the minimum amount of data, as we are storing only half of the matrix's non-zero entries. However, this memory saving comes at the cost of speed. Accessing an arbitrary matrix entry is $O(\frac{1}{2}n) = O(n)$, which can be slow. For small simulations where memory usage is not a concern, we recommend including the option to store entries in a dense matrix.

$\sim u_{i-1,j,k} - \sim u_{i+1,j,k} \sim v_{i,j-1,k} - \sim v_{i,j+1,k} w \sim_{i,j,k-1} - w \sim$matrix. If the reader is interested in implementing their own con-$_{i,j,k+1}$jugate gradient solver, we recommend the paper [Shewchuk 2007][2] as a good starting point. However, in our implementation we are not planning on dealing with enormous bodies of water so we im-

Finally, a reader highly experienced in numerical analysis will realize that A has a form that is common to many other matrices. In 2D, A is often called the *5 point Laplacian Matrix*, whereas in 3D it is called the *7 Point Laplacian Matrix*. Bridson recommends the *Modified Incomplete Cholesky Conjugate Gradient Level 0* algorithm [Bridson 2009]. Essentially, this is simply the conjugate gradient algorithm with a special preconditioner designed for this particular

Eventually, we hope to end up with a system of equations of the form:

$$A\tilde{x} = \tilde{b} \tag{30}$$

Every row of A corresponds to one equation for one fluid cell. In this formulation, we will setup our matrix such that $\tilde{b}$ is simply our negative divergences for every fluid cell. When written out, our linear system takes the following form:

$$\begin{bmatrix} -\Omega_1 & \beta_{1,2} & \dots & \beta_{1,n} \\ \beta_{2,1} & -\Omega_2 & & \\ \dots & \dots & \dots & \\ & & \beta_{n,n-1} & \\ \beta_{n,1} & & & -\Omega_n \end{bmatrix} \begin{bmatrix} p_1 \\ \\ \\ \\ p_n \end{bmatrix} = \begin{bmatrix} -D_1 \\ \\ \dots \\ \\ -D_n \end{bmatrix} \tag{31}$$

In this equation, D_i is the divergence through cell i, Ω_i is the number of non-solid neighbors of cell i, and β_{ij} takes values based on the below equation:

$$\beta_{ij} = \begin{cases} 1 & \text{if cell } i \text{ is a neighbor of cell } j \\ 0 & \text{otherwise.} \end{cases} \tag{32}$$

Our matrix A has many unique properties we can exploit both in our choice of linear solver and in our storage of A itself. It is immediately clear that A is sparse(most of its entires are zero), indicating plemented both the regular *Conjugate Gradient* algorithm, as well as *Parallel Successive Over-Relaxation* (Parallel SOR), and the *Jacobi Method*. We found the parallel SOR to be faster than the conjugate gradient implementation, but this is probably because we are not using a preconditioner. However, conjugate gradient was slightly faster than the Jacobi method in our tests. Note that we used OpenMP for parallelizing our SOR implementation. We hope to add a preconditioner to our conjugate gradient implementation in the near future.

While we have described the characteristics of the linear system to solve, and what kind of solvers to use, we realize that most readers will not want to spend their time writing a highly optimized implementation of a specialized conjugate gradient solver. Therefore, we will quickly direct the reader towards a few good linear algebra packages that have routines that suit our purposes:

- Boost μBLAS
- SparseKit
- http://people.cs.ubc.ca/ rbridson/mpcg/ Open Source Matlab Implementation of Specialized Form of Conjugate Gradient

4.3.4 Grid Update

4.4 Tracking the Water Surface In Grid Bases Simulation

While we have discussed the basic mechanisms for a grid based simulation, we have not discussed how to unify all these ideas into a full working simulator that can output data that encapsulates the position of a moving water surface.

There are many ways to approach the problem of tracking the movement of water through a simulation grid. The simplest way, introduced all the way back in Harlow and Welch's seminal paper [Harlow and Welch 1965b]. This approach is relatively simple, and still quite useful. Here we store a collection of many discrete *marker particles* in our simulation, each representing a water particle. Every timestep, we advect them through the velocity field by Δt. Also, we store an enumeration value inside of each cell indicating whether the cell contains liquid, air, or solid. Once a fluid marker particle moves into a cell, we mark it as liquid. This is necessary for our pressure solve. After each timestep, we can output these particles to disk. However, the question remains as to how to render these particles. One approach is to use a implicit surface function to generate a water surface from these particles. We discuss this approach later, in section the section on surfacing SPH simulations. Unfortunately, this approach can lead to blobby, ugly surfaces. Therefore, we turn to a *level-set* based approach, first introduced in [?].

Level set methods are currently the best way to achieve smooth high quality free surfaces in liquid simulation. However, they are far more computationally expensive than the above implicit surface approach. Here we will present a brief introduction to these techniques. We recommend that an interested reader refer to [Fedkiw and Sethian 2002] for more detailed information.

For the level set method, we define a new value, $\varphi_{i,j,k}$, at the center of all of our simulation cells. We define our liquid free surface to exist at locations where the following equation is satisfied:

$$\varphi(\tilde{X}) = 0 \tag{33}$$

Where $\tilde{X}$ is a position vector. Note that we can define $\Phi(\tilde{X})$ at nongrid cell locations through any type of interpolation, either trilinear or Catmull-Romm is a fine choice.

Furthermore, we say that locations that satisfy $\varphi(\tilde{X}) < 0$ to be inside of the water, and $\varphi(\tilde{X}) < 0$ to be outside of the water. To represent φ, we use a function called the *signed distance function*.

Given an arbitrary set S of kSk points, we define our signed distance function $D(\tilde{X})$ as:

$$D_S(\tilde{X}) = min_{p\sim \in S} \text{k}\tilde{X} - p\sim\text{k} \tag{34}$$

Clearly, for some arbitrary point $\tilde{X}$, the magnitude of the signed distance is the distance to the nearest point in the set S. Signed distance is also useful because of another property. If we want to check whether a grid cell is inside or outside of the fluid, all we must do is examine the sign of the signed distance.

Thus far we have ignored an important problem with signed distance: how to compute it. At the beginning of a simulation, we

can assume our signed distance function is already computed on the grid.

There are many ways to calculate signed distance, and new problem-specific techniques are developed frequently. Typically, people classify these methods into two groups: *PDE based approaches* and *Geometric Approaches*. PDE Based techniques approximate something called the *Eikonal Equation*, $\|\nabla\varphi\| = 1$. These techniques are mathematically and computationally involved, and are often overkill for graphics work. Instead, we will discuss briefly the geometric approaches. Our discussion will not go into much depth; for a more in depth treatment of geometric algorithms for computing signed distance see [?].

Our algorithms for computing signed distance take the following general form:

1. Set the signed distance of each grid-point to "unknown"
2. for each grid point P directly at the free-surface, set the signed distance to 0
3. Loop over each grid-point $G_{i,j,k}$ at which signed distance is unknown

   Loop over each grid-point P that neighbors $G_{i,j,k}$ as long as the signed distance at P is known

   Find the distance from $G_{i,j,k}$ to the surface points. If this distance is closer than P's signed distance, mark P as unknown once again

   Take the minimum value of the distances of the neighbors, and determine if $G_{i,j,k}$ is inside or outside, and set the sign of the distance based on this

There are two main techniques for implementing such an algorithm. These techniques are the *fast marching method*, and the *fast sweeping method*.

The fast marching method loops over the closest grid points first, and then those that are farther away. This technique works rapidly by storing the unknown grid points in a *priority queue* data structure. This algorithm runs in $O(nlog(n))$ when the priority queue is implemented with a heap. A detailed description can be found in [Sethian 1999].

The other technique is the fast sweeping method. The fast sweeping method takes the opposite approach from the fast marching method. Here, we allow the signed distance function to first be calculated at our farthest away points. We then have this information propagate back towards the surface. Fast marching is great because it is $O(n)$, and is very simple. Furthermore, it works well with *narrow band methods*, discussed in [Bridson 2009] and [Fedkiw et al. 2001a].

While we have discussed how to compute a signed distance function, we have not discussed how to update the signed distance as the fluid's free surface moves. While this may seem complicated, this step is quite trivial. Since our φ values are stored in the center of our grid cells, we can simply advect these values according to the fluid's velocity. However, it turns out that advection does not perfectly preserve signed distance. Therefore, we periodically recalculate our signed distance every few timesteps. Bridson reccomends that we recalculate our signed distance once per frame (note that typically many timesteps of Δt are required per frame) [Bridson 2009].

TODO: Finish this section. I am still working on it since my level set implementation is not complete.

5 Particle Based Simulation

5.1 Overview

In this section we describe an alternative approach to fluid simulation. Here we discuss *Lagrangian* techniques. In this approach, we have a set of discrete particles that move through space to represent our fluid. We no longer simulate our fluid on a grid structure.

This approach has many pros and cons compared to grid based techniques. In general, particle based approaches are less accurate than their grid based counterparts. This is primarily due to the difficulties in dealing with spatial derivatives on an unstructured particle cloud. However, particle based simulations are typically much easier to program and understand. Furthermore, particle based techniques are much faster, and can be used in real time applications such as video games.

We will describe *Smoothed Particle Hydrodynamics*. This technique was originally introduced for astrophysical simulations[?], but has also found a lot of uses in computer graphics [?]. This description is especially valuable, since SPH is not discussed in Bridson's text[Bridson 2009], and crucial implementation details are scattered through both astophyics, computational fluid dynamics, and computer graphics literature.

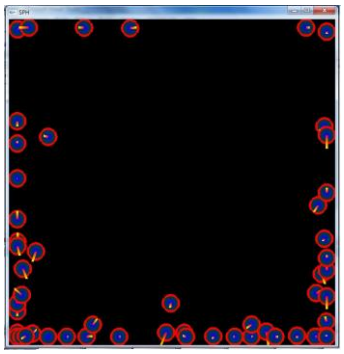


Figure 5: *Our Simple 2D SPH Implementation*

5.2 Data Structures

First, we must outline what information we need to store for our simulation. Clearly, we need a data structure to list all of our particles. Since particles must frequently be added to the simulation, we will choose a linked list. However, the question remains as to what information is stored in each particle.

Clearly, we need to store position, velocity, mass, density, and pressure. It turns out to be useful to store color and a force vector as well, so we will store these. We will refer to these quantities with the following variables throughout our discussion:

- $\tilde{X}$ Position
- $\tilde{V}$ Velocity
- M Mass

- d Density
- ρ Pressure
- $C\sim =< C_{red}, C_{green}, C_{blue} >$ Color
- $F^{\sim}$ Force

Note that for our color, each component of the color is $\in [0,1]$. Each particle can be trivially implemented as a C structure. Notationally, we will refer to particles with the variable P, and individual particles using subscript notation. For instance, the i-th particle would be P_i. Finally, we will refer to particle quantities in a similar manner. For example, the mass of the 12^{th} particle would be M_{12}.

5.3 Algorithm

Our final goal is to satisfy the following condition:

For all particles P_i:

$$\frac{\partial \vec{V}}{\partial t}_i = \vec{A}_i^{pressure} + \vec{A}_i^{viscosity} + \vec{A}_i^{gravity} + \vec{A}_i^{external} \quad (35)$$

Note that in this equation, $A^{\sim something_i}$ refers to the acceleration on particle i due to "something." Also, recall from basic physics ($F = \underline{M}_A$) that $A_i = \frac{F_i}{M_i}$.

In order for our simulation to progress, we need a way to calculate fluid density at some arbitrary point.

Certain particle properties, such as mass, are given initial values at the beginning of the simulation and are not expected to change. However, other properties must be recalculated every step. Consider the pressure property. Here is how to determine the new pressure every time step:

However, we have a discrete cloud off particles, so we must use a discrete summation to approximate this integral. This leads us to the equation:

$$\sum_{j6=i}^{n} M_j W_{Rij} \quad (36)$$

Here, R_{ij} is equal to the Euclidean distance between particle i and particle j.

This function $W(d)$ is known as a *kernel function*. This function takes a single scalar parameter, which is a distance between two particles, and returns a scalar $\in [0,1]$. Typically, a kernel function maps particles that are farther away to values closer to 0. This makes sense, since particles far away will not have a large influence on a particle.

Once particles are far enough away from the source, the kernel function drops to 0, and therefore these particles no longer have to be considered. We will exploit this fact in a later section when developing acceleration structures for SPH simulations.

The question of what kernel function is best is still a very open research question. However, since our simulations are targeted at begin visually pleasing, rather than scientifically accurate, we do not care much about this. The following kernel function has been used extensively in research and practical applications. This is the *Gaussian Kernel*.

$$W(d) = \frac{1}{\pi^{\frac{3}{2}} h^3} \exp(\frac{r^2}{h^2}) \quad (37)$$

Here r is a the distance between two particles, h is our smoothing width. Once particles are greater than distance $2h$ away, they will no longer affect the particles in question. Clearly, larger values of h will make for a more realistic simulation, albeit at the expense of computational speed.

Finally, we present full psueo-code for an SPH simulation:

- Initialize all particles
- Set $t = 0$
- Choose a Δt
- for i from 0 to n for j from 1 to *numparticles*

  Get list L_j of neighbors for P_j

  Calculate $Density_j$ for P_j using L_j

  Calculate $Pressure_j$ for P_j using L_j

  Calculate acceleration A_j for P_j using $Density_j$ and $Pressure_j$

  Move P_j using A_j and Δt using Euler step

  $t = t + \Delta t$
- Cleanup all data structures
- Exit

5.4 Acceleration Structures

As described above, there is a clear computational bottleneck in our application. We must calculate interaction forces between each and every particle. This is an $O(n^2)$ process, which is not computationally viable. By using spatial data structures, we can reduce our computation time to $O(n)$ in the typical case. In the worst case, where all of the particles are in one cell, our algorithm still runs $O(n^2)$ Advanced techniques using quadtrees in 2D, or octrees in 3D, can eliminate this possibility.

In addition to storing all of our particles in a linked list, we also store them in a spatial grid data structure. Our grid cells extend by a distance of R in each dimension. Therefore, in order to calculate the forces on a particular particle, one must only examine 9 grid cells in the 2D case, or 27 grid cells in the 3D case. This is because, for any grid cells far enough away, our kernel function will evaluate to 0 and their contributions will not be included on the current particle. In our experience, including this spatial grid can decrease simulation time by over an order of magnitude for large enough simulations.

However, the addition of this grid data structure requires additional book-keeping during the simulation process. Whenever a particle is moved, one must remove it from its current grid cell, and add it to the grid cell it belongs in. Unfortunately, there is no

way to do this simulation in place, and one must maintain two copies of the simulation grid.

Additionally, this fixed grid requires us to change our kernel function. Oftentimes, implementors in graphics define their kernel functions using a piecewise function. If the particles are less than some distance h apart, they evaluate some type of spline. If the particles are farther away, the kernel instead evalutes to 0. However, more advanced kernels exist for these types of simulations, and they have recently been used with success in graphics. One of the most common advanced kernels is the cubic spline kernel.

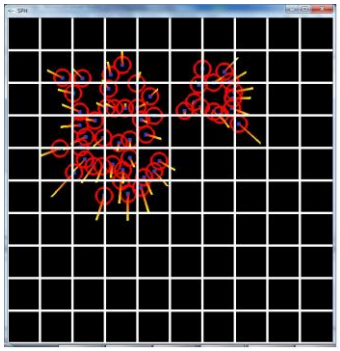


Figure 6: *Our 2D SPH Implementation with Lookup Grid*

Finally, SPH can be further optimized in another way. SPH is clearly very data parallel. Therefore, each particle can be simulated in a separate thread with relative ease. Because of this, many high performance SPH implementations are done on the GPU.

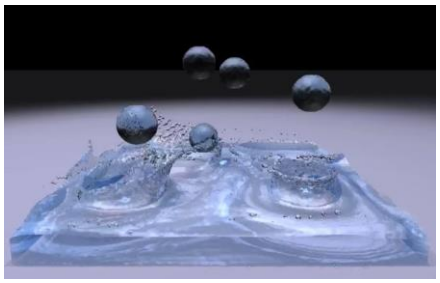


Figure 7: *Raytraced GPU Based 3D SPH from University of Tokyo*

5.5 Surface Tracking

At our current stage, all SPH results in is an unorganized point cloud of fluid particles. This is unacceptable for most applications. Usually the compuiter graphics practitioner desires a way to render these fluids using a off-the-shelf 3D renderer. In this section, we will outline a few techniques for transforming the result of our SPH simulations into a renderable form.

T and probably the easiest technique, is to sample our SPH results onto a uniform grid. Here we can step through the uniform grid points, and sample the fluid density on these points. Then, we can use this uniform grid as input to an application that performs *isosurface volume rendering*. Here we have a choice between *direct volume rendering*, such as that presented in [Colin Braley 2009], and *marching cubes*[Lorenson and Cline 1982]. Direct volume rendering, typically done through volume raycasting, has the advantage of speed. However, there is no easy way to integrate this resulting image with other generated images, and therefore this technique is only suitable for creating previews. Marching cubes, on the other hand, produces triangle meshes. These meshes are suitable for use in a 3D animation program, and this is therefore a viable option for final production.

While there are benefits to sampling our SPH onto a grid, there are other techniques that often produce better results. Usually, these techniques use a special function for each particle that, whe combined with the other particles, produces a fluid surface.

The function usually used for this purpose was introduced by [Blinn 1982]. This technique is often referred to as *meta-balls* or *blobbies* in the graphics community.

$$F(\vec{X}) = \Sigma_i^n k(\frac{\|\vec{X} - \vec{x_i}\|}{h}) \tag{38}$$

Where h is a user specified parameter representing the smoothness of the surface, and k is a kernel function like the one represented above. Incremenetal improvements have been made to the the above function throughout the years, the most imporortant of which is presetned in [Williams 2008].

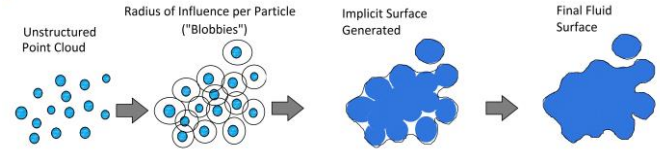


Figure 8: *Metaball Based Surface Reconstruction*

5.6 Extensions

This document has only scratched the surface in discussing the current state of the art fluid simulation techniques. For a thorough explanation of grid based methods, see [Bridson 2009]. No comparable resource for SPH and particle based technique exists, but the author recommends reading the papers of Nils Theurey, and the SIGGRAPH 2006 Fluid Simulation Course Notes as a starting point.

Other active areas of fluid simulation research in the graphics community include, smoke simulation, fire simulation, simulation of highly viscous fluids, and coupled simulations. Coupled simulations combine two or more simulations and get them to interact plausibly. Recently, Ron Fedkiw's group achieved 2-way coupled SPH and grid based simulations in [?]. Furthermore, examples of couplings with thin shells, rigid body simulations, soft body simulations, and cloth simulations exist as well.

Figure 9: *Two Way Coupled SPH and Grid Based Simulation by Ron Fedkiw's Group*

Acknowledgments

Thanks to Robert Hagan for editing this work.

References

BATTY, C., AND BRIDSON, R. 2008. Accurate viscous free surfaces for buckling, coiling, and rotating liquids. In *Proceedings of the 2008 ACM/Eurographics Symposium on Computer Animation*, 219–228.

BATTY, C., BERTAILS, F., AND BRIDSON, R. 2007. A fast variational framework for accurate solid-fluid coupling. *ACM Trans. Graph. 26*, 3, 100.

BLINN, J. F. 1982. A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3, 235–256.

BRIDSON, R. 2009. *Fluid Simulation For Computer Graphics*. A.K Peters.

COLIN BRALEY, ROBERT HAGAN, Y. C. D. G. 2009. Gpu based isosurface volume rendering using depth based coherence. *Siggraph Asia Technical Sketches*.

FEDKIW, R., AND SETHIAN, J. 2002. *Level Set Methods for Dynamic and Implicit Surfaces*.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 15–22.

FEDKIW, R., STAM, J., AND JENSEN, H. W., 2001. Visual simulation of smoke.

FOSTER, N., AND FEDKIW, R. 2001. Practical animation of liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 23–30.

GOKTEKIN, T. G., BARGTEIL, A. W., AND O'BRIEN, J. F. 2004. A method for animating viscoelastic fluids. *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH 2004) 23*, 3, 463–468.

GUENDELMAN, E., SELLE, A., LOSASSO, F., AND FEDKIW, R. 2005. Coupling water and smoke to thin deformable and rigid shells. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 973–981.

HARLOW, F., AND WELCH, J., 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. the physics of fluids 8.

HARLOW, F., AND WELCH, J., 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. the physics of fluids 8.

HARLOW, F., AND WELCH, J., 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. the physics of fluids 8.

CODE

```
import sys
import pygame
from math import sqrt
from random import random as rand
from math import pi

class Particle(object):
    def __init__(self, pos):
        # Scalars
        self.density = 0

        # Forces
        self.position = pos
        self.velocity = Vec2(0,0)
        self.pressure_force = Vec2(0,0)
        self.viscosity_force = Vec2(0,0)



class ParticleGraphics(object):
    def __init__(self, window_size):
        pygame.init()
        self.window = pygame.display.set_mode(window_size)
        self.radius = 5

    def draw(self, particles, H, scale):
        scale = 500 // scale
        self.window.fill((0,0,0))

        for particle in particles:

            # Color based on pressure just for fun
            color = particle.density*50 if particle.density*50 < 255 else
    255

            # Area of influence (H)
            pygame.draw.circle(self.window, (color, 0, 255), (
                int(particle.position.x*scale),
                int(particle.position.y*scale)),
                H*scale//2, 1)

            # Particles
            pygame.draw.circle(self.window, (255, 255, 255), (
                int(particle.position.x*scale),
                int(particle.position.y*scale)),
                self.radius)

            # Velocity vectors
            pygame.draw.line(self.window, (255, 255, 0),
                # start
                (int(particle.position.x*scale),
                 int(particle.position.y*scale)),
                # end
                (int((particle.position.x+particle.velocity.x)*scale),
                 (int((particle.position.y+particle.velocity.y)*scale))))

        # Enable to step through the sim by entering newlines in the
    console
        ### getch = input()

        # Draw to screen
        pygame.display.flip()

graphics = ParticleGraphics((500, 500))

Num = float, int # Numeric classes

class Vec2(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vec2(self.x+other.x, self.y+other.y)
```

```
    def __sub__(self, other):
        return Vec2(self.x-other.x, self.y-other.y)

    def __iadd__(self,other):
        self.x += other.x
        self.y += other.y
        return self

    def __eq__(self, other):
        if self.x == other.x and self.y == other.y:
            return True
        else:
            return False

    def __mul__(self, scalar): # vec * scalar
        return Vec2(self.x*scalar, self.y*scalar)

    def __rmul__(self, scalar): # scalar * vec
        return Vec2(self.x*scalar, self.y*scalar)


def length(v):
    return sqrt(v.x**2 + v.y**2)

#MAIN

# Constants
SCALE = 15
MASS = 5 # inversly proportional Particle mass
DENSITY = 1 # Rest density
GRAVITY = Vec2(0, 0.5)
H = 1  # Smoothing cutoff- essentially, particle size
k = 20  # Temperature constant- higher means particle repel more strongly
eta = 1  # Viscosity constant- higher for more viscous


def W(r, h):  # :: Num
    '''
    A weighting function (kernel) for the contribution of each neighbor
    to a particle's density. Forms a nice smooth gradient from the center
    of a particle to H, where it's 0
    '''
    if 0 < length(r) <= h:
        ret= 315/(64 * pi * h**9) * (h**2 - length(r)**2)**3
    else:
        ret= 0

    # Typecheck
    return ret


def gradient_Wspiky(r, h):  # :: Vec2
    '''
    Gradient ( that is, Vec2(dx, dy) ) of a weighting function for
    a particle's pressure. This weight function is spiky (not flat or
    smooth at x=0) so particles close together repel strongly
    '''
    len_r = length(r)

    if 0 < len_r <= h:
        ret =  -1 * r * (45/(pi * h**6 * len_r)) * (h - len_r)**2
    else:
        ret = Vec2(0, 0)

    return ret


def laplacian_W_viscosity(r, h):  # :: Num
    '''
    The laplacian of a weighting function that tends towards infinity when
    approching 0 (slows down particles moving faster than their neighbors)
    '''
    len_r = length(r)
```

```
    if 0 < len_r <= h:
        ret = 45/(2 * pi * h**5) * (1 - len_r/h)
    else:
        ret = 0

    return ret



# Instantiate particles!
width = 20
height = 10

particles = []
for x in range(10):
    for y in range(10):
        particles.append(Particle(Vec2(x+1+rand()*0.1, y+5)))



# random distribution
# particles = [Particle(Vec2(rand()*SCALE, rand()*SCALE))
#                 for p in range(NUM_PARTICLES)]

time = 0
delta_time = 0.1
while True:

    # Clear everything
    for particle in particles:
        particle.density = DENSITY
        particle.pressure_force = Vec2(0,0)
        particle.viscosity_force = Vec2(0,0)

    # Calculate fluid density around each particle
    for particle in particles:
        for neighbor in particles:

            # If particles are close together, density increases
            distance = particle.position - neighbor.position # A vector

            if length(distance) <= H:  # Particles are close enough to matter
                particle.density += MASS * W(distance, H)

    # Calculate forces on each particle based on density
    for particle in particles:
        for neighbor in particles:

            distance = particle.position - neighbor.position
            if length(distance) <= H:
                # Temporary terms used to caclulate forces
                density_p = particle.density
                density_n = neighbor.density
                assert(density_n != 0)  # Dividing by density later

                # Pressure derived
                pressure_p = k * (density_p - DENSITY)
                pressure_n = k * (density_n - DENSITY)

                # Navier-Stokes equations for pressure and viscosity
                # (ignoring surface tension)
                particle.pressure_force += (-1 *
                    MASS * (pressure_p + pressure_n) / (2 * density_n)
                    * gradient_Wspiky(distance, H))

                particle.viscosity_force += (
                    eta * MASS * (neighbor.velocity - particle.velocity)
                    * (1/density_n) * laplacian_W_viscosity(distance, H))

    # Apply forces to particles- make them move!
    for particle in particles:
        total_force = particle.pressure_force + particle.viscosity_force

        # 'Eulerian' style momentum:

        # Calculate acceleration from forces
```

```
    acceleration = total_force * (1/particle.density) \
            * delta_time + GRAVITY

    # Update position and velocity
    particle.velocity += acceleration * delta_time
    particle.position += particle.velocity * delta_time

    # Make sure particles stay in bounds
    # TODO: Better boundary conditions (THESE ARE BAD)
    if particle.position.x >= SCALE - 0.01:
        particle.position.x = SCALE - (0.01 + 0.1*rand())
        particle.velocity.x = 0
    elif particle.position.x < 0.01:
        particle.position.x = 0.01 + 0.1*rand()
        particle.velocity.x = 0

    if particle.position.y >= SCALE - 0.01:
        particle.position.y = SCALE - (0.01+rand()*0.1)
        particle.velocity.y = 0
    elif particle.position.y < 0.01:
        particle.position.y = 0.01 + rand()*0.1
        particle.velocity.y = 0

graphics.draw(particles, H, SCALE)

time += delta_time
```

LORENSON, AND CLINE. 1982. Marching cubes. *ACM Trans. Graph. 1*, 3, 235–256.

PETER, M. C., MUCHA, P. J., BROOKS, R., III, V. H., AND TURK, G., 2002. Melting and flowing.

PETER, M. C., MUCHA, P. J., AND TURK, G. 2004. Rigid fluid: Animating the interplay between rigid bodies and fluid. In *ACM Trans. Graph*, 377–384.

SETHIAN, J. A. 1999. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science (Cambridge ... on Applied and Computational Mathematics)*, 2 ed. Cambridge University Press, June.

SHEWCHUK, J. R. 2007. Conjugate gradient without the agonizing pain. Tech. rep., Carnegie Mellon.

STAM, J. 1999. Stable fluids. 121–128.

WILLIAMS, B. W. 2008. *Fluid Surface Reconstruction from Particles*. Master's thesis, University of British Columbia.