Contents

Part I

Survey

1 Introduction

Fluid simulation is a very active topic in the field of computer science. The complexity of many fluid phenomena makes them difficult to replicate, but when done properly yields visually pleasing results. The high challenge and high reward makes it a very appealing topic to most computer scientists, and is a driving motivator behind this survey. However, visually pleasing simulations tend to come at a high cost computationally, especially due to the complex behavior of fluid, and in many cases make it implausible to run at interactive rates. For this reason, computer scientists have simplified components of the equations that describe the behavior of fluid in an attempt to overcome this obstacle. This, coupled with the increasing power of computers, is making realistic, real-time water simulation possible.

The following is a survey on the topic of water simulation in the field of computer science. This survey will highlight the most prominent schools of thought in this field. Though there are many different ways to go about simulating fluid, most borrow from Eulerian-grid-based or Lagrangian-particle-based approaches. Before getting into the finer details of these approaches, it is important to understand the mathematics behind them. Therefore, the first sections of this survey will go over necessary vector calculus and fluid-dynamics equations.

2 Vector Calculus

2.1 Scalar and Vector Fields

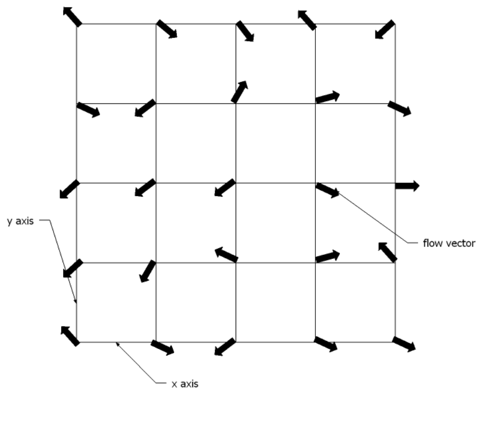
Because fluid is a continuous fluid, and its characteristics at some position within it can vary drastically from any other point in the fluid, the notion of a scalar or vector field becomes a very powerful tool. A scalar field is a function of position that results in a scalar value denoted as

$$f(p) = f(x, y, z) = s,$$

where p is a three dimensional vector and s is a scalar value. For example, the temperature, t, of some fluid at position p, would be $f(p) = t$. A vector field, on the other hand, is a function of position that results in a vector denoted as

$$f(p) = f(x, y, z) = v,$$

where v is a three dimensional vector. An example of this might be the direction, v, of flow in a fluid at position p.



A two dimensional vector field.

2.2 Gradient

Now that a data structure, that can be used to represent the state of a volume of fluid, has been defined, it is important to define operators that can be applied to it. The gradient, for example, when applied to a scalar field, results in a vector field, where each vector points in the direction of the greatest increase of the scalar field. The gradient is defined by

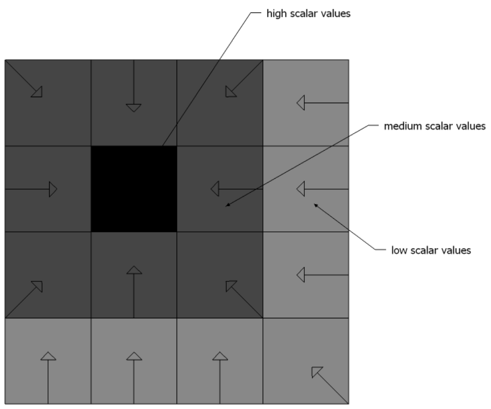
$$\nabla(s_f) = \left\{ \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right\} s_f = \left\{ \frac{\partial s_f}{\partial x}, \frac{\partial s_f}{\partial y}, \frac{\partial s_f}{\partial z} \right\},$$

where ∇, nabla, is the gradient operator and s_f is a scalar field. For example, if

$$s_f = x^2yz + 2xy^3z + 3xyz^4,$$

then

$$\nabla(s_f) = v_f \left\{ \begin{array}{ccc} 2xyz, & 6xy^2z, & 12xyz^3 \end{array} \right\}$$



A scalar field overlaid with its resultant vector field after application of gradient operator.

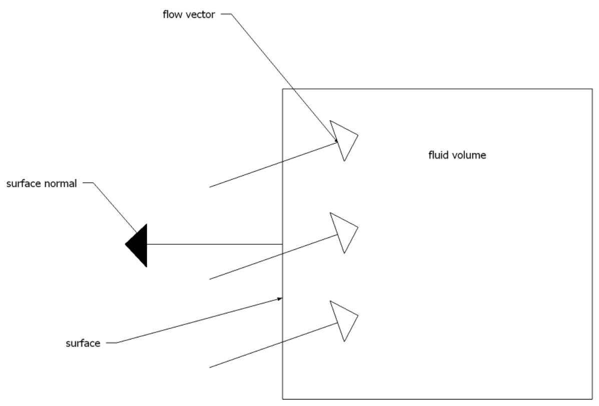
Note The gradient can also be applied to a vector field. In this case the application results in a matrix that represents the rate of increase, or acceleration.

2.3 Divergence and Flux

First consider flux. Flux, denoted as Φ, is the total volume of fluid passing through a surface per unit time, i.e.:

$$\Phi = \int_S v \cdot n dS,$$

where v is a vector representing the flow, n is a vector perpendicular to the surface S, and Φ is the resultant flux.



A two dimensional depiction of the flux through a fluid surface.

For example, let $p = \{1,2,3\}$, $v = \{1,2,1\}$, $n = \{1,0,0\}$, and $\delta y, \delta z = 2$. Where p is the center of the sub surface of S, v is the vector at position p in the vector field and n is a vector normal to the surface S. Then the flux, Φ, at position p is

$$\int_1^3 \int_2^4 v \cdot n \delta y \delta z = \int_1^3 \int_2^4 1 \delta y \delta z = \int_1^3 y|_2^4 \delta z = \int_1^3 2 \delta z = 2|_1^3 = 4 = \Phi.$$

Now consider the divergence of a vector field as it relates to flux. The divergence of a vector field yields a scalar field, where each scalar value represents the net flux through the surface of the volume surrounding its position in the field as that volume approaches zero, i.e.:

$$\nabla \cdot v_f = lim_{V \to 0} \int_S \frac{v \cdot n}{V} \delta S, \tag{2.1}$$

where V is an arbitrary volume of fluid around the position of the vector v. For example, let v_f be a vector field such that:

$$v_f = \{x + y + z, x^2 + y^2 + z^2, x^3 + y^3 + z^3$$

then,

$$\nabla \cdot v_f = \left\{ \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right\} \cdot \{x+y+z, x^2+y^2+z^2, x^3+y^3+z^3 \quad = \{1+2y+3z^2$$

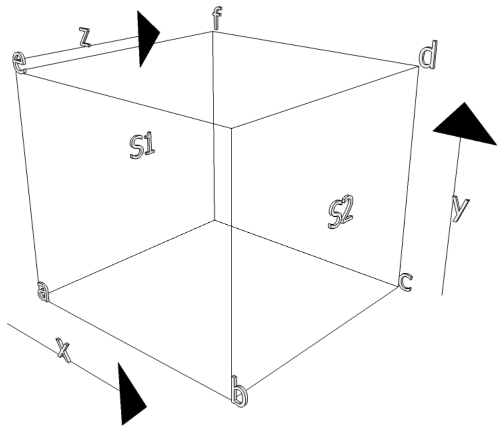
2.4 Divergence Theorem

Integrating both sides of equation 2.1 over the volume, V, yields:

$$\int_V (\nabla \cdot v_f)\, dV = \int_S v_f \cdot n dS,$$

which is the divergence theorem. The divergence theorem states that the flux through a fluid's surface is equal to the volume integral of its divergence. The divergence theorem was first discovered by Joseph-Luis Lagrange [20] in 1762 and later by Gauss [6] and Green [9].

To conceptualize this, think of a fluid in a rectangular, axis aligned cube, with boundaries $\{a \leq x \leq b, c \leq y \leq d, e \leq z \leq f\}$.



An axis aligned cube representing the volume of a fluid.

The flux along the x axis is

$$\int_{S1} v_f \cdot n dS_1 + \int_{S2} v_f \cdot n dS_2.$$

Expanding out the surface integrals gives:

$$-\int_e^f \int_c^d v_f(a, y, z) \cdot n dy dz + \int_e^f \int_c^d v_f(b, y, z) \cdot n dy dz,$$

or,

$$\int_e^f \int_c^d (v_f(b,y,z) - v_f(a,y,z)) \cdot ndydz.$$

The fundamental theorem of calculus states that

$$\int_a^b f(x)dx = F(b) - F(a),$$

therefore replacement of $(v_f(b,y,z) - v_f(a,y,z))$ with $\int_a^b \frac{\partial v_x}{\partial x} dx$ can result in

$$\int_e^f \int_c^d \int_a^b \frac{\partial v_x}{\partial x} dxdydz.$$

The same can be said for the y and z axis. Adding them together yields

$$\int_s v_f \cdot ndS = \int_e^f \int_c^d \int_a^b \left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \right) dxdydz = \int_V (\nabla \cdot v_f)\, dV.$$

This concludes the vector calculus review of the survey. What has been reviewed should prove to be enough of a basis to start the derivation of the fluid equations shown in the following sections.

2.5 Computational Fluid Dynamics (CFD) Equations

With the basics of vector calculus understood, the fundamental mathematics behind computational fluid dynamics can now be derived. Sections 2.6, 2.7 and 2.8 contain derivations of key components of the Navier-Stokes equations as shown in [1]. These equations are the basis for all realistic water simulations.

2.6 Conservation of Mass Derivation

The change in mass of a volume of fluid can be described as the rate of change of the volume's density multiplied by its area, denoted:

$$\frac{\partial}{\partial t}Mass = \frac{\partial \rho}{\partial t}\delta x \delta y \delta z.$$

The rate of mass entering the volume in the positive x direction is the density times the velocity in the x direction times the area of the surface it is flowing through, denoted:

$$\rho v_x \delta y \delta z.$$

Because the mass flux may have changed within the volume, the flux through the opposite surface of the volume in the x direction is the sum of the density and the change in density multiplied by the sum of the velocity in the x direction and the change in velocity in the x direction all multiplied by the area of the surface – the flux is negative because the flow is now leaving the volume.

$$-(\rho + \delta\rho)(v_x + \delta v_x)\delta y \delta z$$

Follow this same logic for the y and z directions and add them all up to get the total change in mass within the volume.

$$\begin{aligned}
\tfrac{\partial \rho}{\partial t}\delta x \delta y \delta z &= \rho v_x \delta y \delta z - (\rho + \delta\rho)(v_x + \delta v_x)\delta y \delta z \\
&+ \rho v_y \delta x \delta z - (\rho + \delta\rho)(v_y + \delta v_y)\delta x \delta z \\
&+ \rho v_z \delta x \delta y - (\rho + \delta\rho)(v_z + \delta v_z)\delta x \delta y
\end{aligned}$$

Multiplying and expanding terms in the second column of the right side of the equation yields:

$$\begin{aligned}\tfrac{\partial\rho}{\partial t}\delta x\delta y\delta z &= \rho v_x\delta y\delta z-(\rho v_x+\rho\delta v_x+\delta\rho v_x+\delta\rho\delta v_x)\delta y\delta z\\ &+ \rho v_y\delta x\delta z-(\rho v_y+\rho\delta v_y+\delta\rho v_y+\delta\rho\delta v_y)\delta x\delta z\\ &+ \rho v_z\delta x\delta y-(\rho v_z+\rho\delta v_z+\delta\rho v_z+\delta\rho\delta v_z)\delta x\delta y\end{aligned}$$

Pulling out the area terms gives:

$$\begin{aligned}\tfrac{\partial\rho}{\partial t}\delta x\delta y\delta z &= (\rho v_x-\rho v_x-\rho\delta v_x-\delta\rho v_x-\delta\rho\delta v_x)\delta y\delta z\\ &+ (\rho v_y-\rho v_y-\rho\delta v_y-\delta\rho v_y-\delta\rho\delta v_y)\delta x\delta z\\ &+ (\rho v_z-\rho v_z-\rho\delta v_z-\delta\rho v_z-\delta\rho\delta v_z)\delta x\delta y\end{aligned}$$

Canceling out ρv_f in the first two columns results in:

$$\begin{aligned}\tfrac{\partial\rho}{\partial t}\delta x\delta y\delta z &= (-\rho\delta v_x-\delta\rho v_x-\delta\rho\delta v_x)\delta y\delta z\\ &+ (-\rho\delta v_y-\delta\rho v_y-\delta\rho\delta v_y)\delta x\delta z\\ &+ (-\rho\delta v_z-\delta\rho v_z-\delta\rho\delta v_z)\delta x\delta y.\end{aligned}$$

As we take the limit $\delta p, \delta v$ we can neglect higher order terms. Because of this the last terms multiplied by $\delta y\delta x$ are dropped resulting in:

$$\begin{aligned}\tfrac{\partial\rho}{\partial t}\delta x\delta y\delta z &= (-\rho\delta v_x-\delta\rho v_x)\delta y\delta z\\ &+ (-\rho\delta v_y-\delta\rho v_y)\delta x\delta z\\ &+ (-\rho\delta v_z-\delta\rho v_z)\delta x\delta y,\end{aligned}$$

or, via the chain rule:

$$\begin{aligned}\tfrac{\partial\rho}{\partial t}\delta x\delta y\delta z &= -\delta(\rho v_x)\delta y\delta z\\ &+ -\delta(\rho v_y)\delta x\delta z\\ &+ -\delta(\rho v_z)\delta x\delta y.\end{aligned}$$

Divide by the area of the volume.

$$\begin{aligned}\frac{\partial\rho}{\partial t} &= -\frac{\partial\rho v_x}{\partial x}\\ &+ -\frac{\partial\rho v_y}{\partial y}\\ &+ -\frac{\partial\rho v_z}{\partial z}\end{aligned}$$

Rearrange and set equal to zero.

$$\frac{\partial\rho}{\partial t} + \frac{\partial\rho v_x}{\partial x} + \frac{\partial\rho v_y}{\partial y} + \frac{\partial\rho v_z}{\partial z} = 0$$

The density within the volume is constant, so the first term is zero. The remainder is the density of the volume multiplied by the divergence of the velocity field, which is the total change of the mass within the volume. This change is zero, therefore mass is conserved.

$$\rho(\nabla \cdot v_f) = 0$$

2.7 Conservation of Momentum Derivation

First, recognize that the change in momentum equation can be denoted as the following:

$$\frac{\partial}{\partial t}Momentum = \frac{\partial}{\partial t}\rho v_f \delta x \delta y \delta z.$$

With that in mind, the following derivation will focus on momentum in the x direction. The complete set of equations will be derived later, by symmetry. The momentum flux in the x direction is the product of the mass flux and the velocity in the x direction, therefore the momentum flux in the x direction of the surface of a volume whose normal is $< -1, 0, 0 >$ can be written as

$$\rho v_x v_x \delta y \delta z.$$

The flux through the opposite surface of the volume, with normal $< 1, 0, 0 >$, would then be written as

$$-\left(\rho v_x v_x + \frac{\partial}{\partial x}\rho v_x v_x \delta x\right)\delta y \delta z.$$

The momentum flux through the y and z surfaces can be symmetrically described as the equation above and after summing them all together with external forces in the x directions, $\sum f_x$, the result is the following:

$$\begin{aligned}
\tfrac{\partial}{\partial t}\rho v_x \delta x \delta y \delta z &= \rho v_x v_x \delta y \delta z - \left(\rho v_x v_x + \tfrac{\partial}{\partial x}\rho v_x v_x \delta x\right)\delta y \delta z \\
&+ \rho v_y v_x \delta x \delta z - \left(\rho v_y v_x + \tfrac{\partial}{\partial y}\rho v_y v_x \delta y\right)\delta x \delta z \\
&+ \rho v_z v_x \delta x \delta y - \left(\rho v_z v_x + \tfrac{\partial}{\partial z}\rho v_z v_x \delta z\right)\delta x \delta y \\
&+ \sum f_x.
\end{aligned}$$

The first two columns cancel out. This simplifies to

$$\left(\frac{\partial \rho v_x}{\partial t} + \frac{\partial \rho v_x v_x}{\partial x} + \frac{\partial \rho v_y v_x}{\partial y} + \frac{\partial \rho v_w v_x}{\partial z}\right)\delta x \delta y \delta z = \sum f_x.$$

Next, expand with the product rule to yield the following equation:

$$v_x\frac{\partial \rho}{\partial t} + v_x\frac{\partial \rho v_x}{\partial x} + v_x\frac{\partial \rho v_y}{\partial y} + v_x\frac{\partial \rho v_z}{\partial z} + \rho\frac{\partial v_x}{\partial t} + \rho v_x\frac{\partial v_x}{\partial x} + \rho v_y\frac{\partial v_x}{\partial y} + \rho v_z\frac{\partial v_x}{\partial z} = \sum f_x.$$

Use conservation of mass equation to cancel out the four terms on the left.

$$\begin{aligned}
v_x\tfrac{\partial \rho}{\partial t} + v_x\tfrac{\partial \rho v_x}{\partial x} + v_x\tfrac{\partial \rho v_y}{\partial y} + v_x\tfrac{\partial \rho v_z}{\partial z} + \quad & \rho\tfrac{\partial v_x}{\partial t} + \rho v_x\tfrac{\partial v_x}{\partial x} + \rho v_y\tfrac{\partial v_x}{\partial y} + \rho v_z\tfrac{\partial v_x}{\partial z} = \sum f_x \\
\rho v_x\left(\tfrac{\partial}{\partial t} + \tfrac{\partial v_x}{\partial x} + \tfrac{\partial v_y}{\partial y} + \tfrac{\partial v_z}{\partial z}\right) + \quad & \rho\tfrac{\partial v_x}{\partial t} + \rho v_x\tfrac{\partial v_x}{\partial x} + \rho v_y\tfrac{\partial v_x}{\partial y} + \rho v_z\tfrac{\partial v_x}{\partial z} = \sum f_x \\
\rho v_x(\nabla \cdot v_f) + \quad & \rho\tfrac{\partial v_x}{\partial t} + \rho v_x\tfrac{\partial v_x}{\partial x} + \rho v_y\tfrac{\partial v_x}{\partial y} + \rho v_z\tfrac{\partial v_x}{\partial z} = \sum f_x \\
0 + \quad & \rho\tfrac{\partial v_x}{\partial t} + \rho v_x\tfrac{\partial v_x}{\partial x} + \rho v_y\tfrac{\partial v_x}{\partial y} + \rho v_z\tfrac{\partial v_x}{\partial z} = \sum f_x
\end{aligned}$$

Following this same logic for the y and z directions yields the equations

$$\rho\frac{\partial v_x}{\partial t} + \rho v_x\frac{\partial v_x}{\partial x} + \rho v_y\frac{\partial v_x}{\partial y} + \rho v_z\frac{\partial v_x}{\partial z} = \sum f_x,$$

$$\rho\frac{\partial v_y}{\partial t} + \rho v_x\frac{\partial v_y}{\partial x} + \rho v_y\frac{\partial v_y}{\partial y} + \rho v_z\frac{\partial v_y}{\partial z} = \sum f_y,$$

and

$$\rho\frac{\partial v_x}{\partial t} + \rho v_x \frac{\partial v_x}{\partial x} + \rho v_y \frac{\partial v_z}{\partial y} + \rho v_z \frac{\partial v_z}{\partial z} = \sum f_z,$$

which is

$$\rho(\frac{\partial v_f}{\partial t} + v_f \cdot \nabla v_f) = \sum f.$$

2.8 External Forces Derivation

The conservation of momentum equation requires knowledge of the external forces. There are two types of external forces that can be applied to a fluid. The first type, body forces, are applied to the entire volume. Gravity, for example, would be described as

$$f_g = g\rho\delta x\delta y\delta z.$$

The second type, surface forces, are described as the sum of stresses on a given sub-surface of the volume of fluid. This stress must be calculated for each sub-surface of the volume. Stress, denoted σ_{ij}, is an outward force, where i is the normal direction of the surface and j is the direction of the stress. Knowing this, the stress on a surface with normal $< -1, 0, 0 >$ in the x direction can be described as

$$f_{s_x1} = -\sigma_{xx}\delta y\delta z,$$

and the stress on a surface with normal $< 1, 0, 0 >$ in the same direction as

$$f_{s_x2} = \left(\sigma_{xx} + \frac{\partial}{\partial x}\sigma_{xx}\delta x\right)\delta y\delta z.$$

Sum these two to get the following:

$$f_{s_x1} + f_{s_x2} = -\sigma_{xx}\delta y\delta z + \left(\sigma_{xx} + \frac{\partial}{\partial x}\sigma_{xx}\delta x\right)\delta y\delta z$$

or

$$f_{s_x 1} + f_{s_x 2} = \frac{\partial}{\partial x}\sigma_{xx}\delta x\delta y\delta z.$$

Now, to get the total amount of stress force in the x direction, add forces due to shearing. Which are obtained in a similar fashion as just shown. Adding them all up gives us the total amount of stress in the volume:

$$\sum f_x = \frac{\partial}{\partial x}\sigma_{xx}\delta x\delta y\delta z + \frac{\partial}{\partial y}\sigma_{yx}\delta x\delta y\delta z + \frac{\partial}{\partial z}\sigma_{zx}\delta x\delta y\delta z$$

or

$$\sum f_x = \left(\frac{\partial}{\partial x}\sigma_{xx} + \frac{\partial}{\partial y}\sigma_{yx} + \frac{\partial}{\partial z}\sigma_{zx}\right)\delta x\delta y\delta z.$$

Stress can be further broken up into stress due to pressure, p, and stress due to the viscosity of the fluid, τ_{ij}. Pressure, acting inward, is negative, while viscosity forces push out. τ_{xx} describes the normal viscosity forces in the x direction, while τ_{yx} and τ_{zx} describes viscous stress due to shearing.

$$\sum f_x = \left(\frac{\partial}{\partial x}\tau_{xx} + \frac{\partial}{\partial y}\tau_{yx} + \frac{\partial}{\partial z}\tau_{zx} - \frac{\partial p}{\partial x}\right)\delta x\delta y\delta z$$

For the purposes of this derivation, it is assumed that the fluid in question is an incompressible newtonian fluid. In such a case the viscous stresses are described as the following:

$$\tau_{xx} = 2\mu\frac{\partial v_x}{\partial x},$$

$$\tau_{yx} = \mu\left(\frac{\partial v_y}{\partial x} + \frac{\partial v_x}{\partial y}\right),$$

and

$$\tau_{zx} = \mu\left(\frac{\partial v_z}{\partial x} + \frac{\partial v_x}{\partial z}\right).$$

Plug these values into the stress equation to get:

$$\sum f_x = \left(\frac{\partial}{\partial x}\left[2\mu\frac{\partial v_x}{\partial x}\right] + \frac{\partial}{\partial y}\left[\mu\left(\frac{\partial v_y}{\partial x} + \frac{\partial v_x}{\partial y}\right)\right] + \frac{\partial}{\partial z}\left[\mu\left(\frac{\partial v_z}{\partial x} + \frac{\partial v_x}{\partial z}\right)\right] - \frac{\partial p}{\partial x}\right)\delta x\delta y\delta z.$$

Now expand and rearrange the equation to make it more manageable:

$$\sum f_x = \mu\left(\frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2} + \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_y}{\partial x\partial y} + \frac{\partial^2 v_z}{\partial x\partial z}\right) - \frac{\partial p}{\partial x}.$$

The last three terms inside the parentheses can be dropped due to conservation of momentum.

$$\begin{array}{lccr}
\sum f_x = \mu(& \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2} + & \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_y}{\partial x\partial y} + \frac{\partial^2 v_z}{\partial x\partial z} &) - \frac{\partial p}{\partial x} \\
\sum f_x = \mu(& \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2} + & \frac{\partial}{\partial x}\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right) &) - \frac{\partial p}{\partial x} \\
\sum f_x = \mu(& \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2} + & \frac{\partial}{\partial x}(0) &) - \frac{\partial p}{\partial x} \\
\sum f_x = \mu(& \frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2} + & 0 &) - \frac{\partial p}{\partial x}
\end{array}$$

And the sum of all external forces due to stress in the x direction is:

$$\sum f_x = \mu\left(\frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2}\right) - \frac{\partial p}{\partial x}.$$

The y and z stresses can be symmetrically derived to yield the final set of equations:

$$\sum f_x = \mu\left(\frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2}\right) - \frac{\partial p}{\partial x},$$

$$\sum f_y = \mu\left(\frac{\partial^2 v_y}{\partial x^2} + \frac{\partial^2 v_y}{\partial y^2} + \frac{\partial^2 v_y}{\partial z^2}\right) - \frac{\partial p}{\partial y},$$

and

$$\sum f_z = \mu\left(\frac{\partial^2 v_z}{\partial x^2} + \frac{\partial^2 v_z}{\partial y^2} + \frac{\partial^2 v_z}{\partial z^2}\right) - \frac{\partial p}{\partial z}$$

.

Syntax of the above equations can be simplified to the following single equation:

$$\sum f_x = \mu \nabla^2 v_f - \nabla p.$$

2.9 The Navier-Stokes Equations (Putting it all together)

So far the derivation of the conservation of mass and momentum, as well as the equations that describe external forces have been shown. Putting all these equations together gives us the following:

$$\rho(\frac{\partial v_f}{\partial t} + v_f \cdot \nabla v_f) = \mu \nabla^2 v_f - \nabla p + g$$

or

$$\frac{\partial v_f}{\partial t} = -(v_f \cdot \nabla) v_f + \frac{\mu}{\rho} \nabla^2 v_f - \frac{1}{\rho} \nabla p + g$$

Where $-(v_f \cdot \nabla) v_f$ describes the advection, $\frac{\mu}{\rho} \nabla^2 v_f$ describes the stress forces due to viscosity, $\frac{1}{\rho} \nabla p$ describes the stress forces due to pressure, and g describes external body accelerations such as gravity. That coupled with the conservation of mass equation:

$$\nabla \cdot v_f = 0$$

gives us the complete derivation of the Navier-Stokes equations – named after Claude-Luis Navier and George Gabriel Stokes for their mathematical contributions to the equations.

3 Toward Interactive-Rate Simulation of Fluids with Moving Obstacles Using Navier-Stokes Equations

Chen and Lobo simulated a three-dimensional liquid surface in [3]. However, they only apply the fluid dynamics equations in two-dimensional space. After solving the equations for a two dimensional grid, they use the pressure of each cell to determine a height, and thus yielding the third dimension. This is justified by the fact that when incompressible water rushes into a single area that is bounded on the bottom, the pressure forces the column of water to rise vertically.

3.1 Implementation

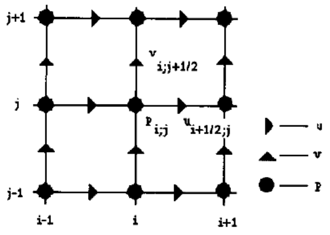
Chen and Lobo solve the Navier-Stokes equations using a finite-differencing solution, that uses penalization. Essentially they change the conservation of mass equation to:

$$\varepsilon p + \nabla \cdot v_f = 0, \varepsilon > 0, \varepsilon \to 0$$

Where ε is the penalty parameter. In addition to this they describe the viscosity term of the Navier-Stokes equation as:

$$\frac{1}{Re}\nabla^2 v_f$$

Where Re is the Reynolds number, a parameter that indicates the flow regime of the fluid. The parameter is defined as $Re = \frac{\rho v L}{\mu}$, where L and v are a characteristic length and velocity. For spacial descritization they make use of a staggered marker and cell mesh, illustrated by the image below.



Where u and v are the velocities in the x and y directions, and p is the pressure at that point. With this data structure in mind, the process of updating the velocity field is the following:

$$u^{n+1}_{i+\frac{1}{2};j} = u^{n}_{i+\frac{1}{2};j} + \left(-a^{n}_{i+\frac{1}{2};j} - \triangle^1_x p^n_{i+\frac{1}{2};j} + \frac{1}{Re}\nabla^2_h u^n_{i+\frac{1}{2};j}\right)\triangle t$$

$$v^{n+1}_{i;j+\frac{1}{2}} = v^{n}_{i;j+\frac{1}{2}} + \left(-b^{n}_{i;j+\frac{1}{2}} - \triangle^1_y p^n_{i;j+\frac{1}{2}} + \frac{1}{Re}\nabla^2_h v^n_{i;j+\frac{1}{2}}\right)\triangle t$$

$$p_{i;j}^{n+1} = -\frac{\triangle_x^1 u_{i;j}^{n+1} + \triangle_y^1 v_{i;j}^{n+1}}{\varepsilon}$$

Where i and j are indexes into the mesh field, n is the current state of the field, and $n+1$ is the state of the mesh field after an amount of time has passed, $\triangle t$. The operators $\triangle_x^1$, $\triangle_y^1$, and ∇_h^2 are defined as:

$$\triangle_x^1 f_{l;m} = \frac{1}{\triangle x}\left(f_{l+\frac{1}{2};m} - f_{l-\frac{1}{2};m}\right)$$

$$\triangle_y^1 f_{l;m} = \frac{1}{\triangle y}\left(f_{l;m+\frac{1}{2}} - f_{l;m-\frac{1}{2}}\right)$$

$$\nabla_h^2 f_{l;m} = \triangle_{xx} f_{l;m} + \triangle_{yy} f_{l;m}$$

$$\triangle_{xx} f_{l;m} = \frac{f_{l+1;m} - 2f_{l;m} + f_{l-1;m}}{\triangle x^2}$$

$$\triangle_{yy} f_{l;m} = \frac{f_{l;m+1} - 2f_{l;m} + f_{l;m-1}}{\triangle y^2}$$

That leaves the final terms still not defined, $a_{i+\frac{1}{2};j}^n$ and $b_{i;j+\frac{1}{2}}^n$, which are described by the following:

$$a_{i+\frac{1}{2};j}^n = u_{i+\frac{1}{2};j}^n \triangle_x^0 u_{i+\frac{1}{2};j}^n + V_{i+\frac{1}{2};j}^n \triangle_y^0 u_{i+\frac{1}{2};j}^n$$

$$b_{i;j+\frac{1}{2}}^n = U_{i;j+\frac{1}{2}}^n \triangle_x^0 v_{i;j+\frac{1}{2}}^n + v_{i;j+\frac{1}{2}}^n \triangle_y^0 v_{i;j+\frac{1}{2}}^n$$

where

$$U_{i;j+\frac{1}{2}} = \frac{1}{4}\left(u_{i+\frac{1}{2};j} + u_{i+\frac{1}{2};j+1} + u_{i-\frac{1}{2};j+1} + u_{i-\frac{1}{2};j}\right)$$

$$V_{i+\frac{1}{2};j} = \frac{1}{4}\left(v_{i+1;j+\frac{1}{2}} + v_{i;j+\frac{1}{2}} + v_{i+1;j-\frac{1}{2}} + u_{i;j-\frac{1}{2}}\right)$$

$$\triangle_x^0 f_{l;m} = \frac{1}{2\triangle x}\left(f_{l+1;m} - f_{l-1;m}\right)$$

$$\triangle_y^0 f_{l;m} = \frac{1}{2\triangle y}\left(f_{l;m+1} - f_{l;m-1}\right)$$

After calculating the velocities of the waters surface at each edge of the mesh, the final flow field at each point can be calculated by:

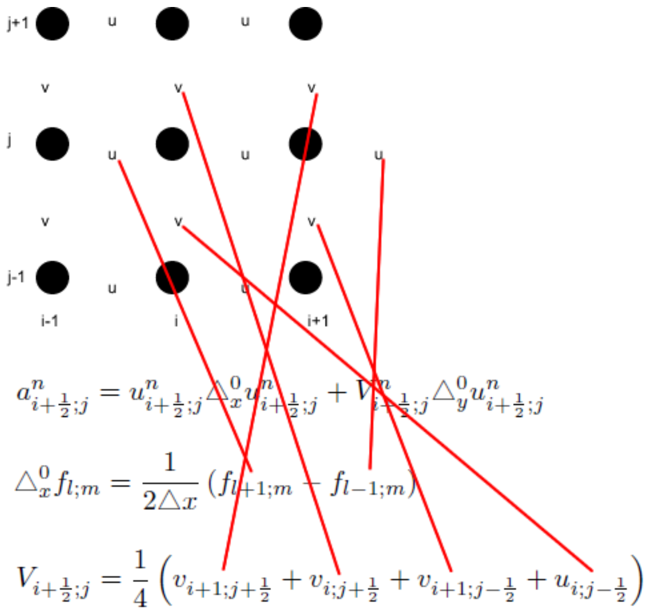
$$u_{i,j} = \frac{u_{i+\frac{1}{2};j} + u_{i-\frac{1}{2};j}}{2}$$

$$v_{i,j} = \frac{v_{i;j+\frac{1}{2}} + v_{i;j-\frac{1}{2}}}{2}$$

Finally, a vector can be drawn from a single point in the grid (i, j) using the velocity at that point. Scaling that vector in the third dimension allows for three dimensional rendering of the fluid.

3.2 Visual Steps

To aid in the visualization of the above equations, here are a few diagrams that map out what values are modified for which terms of the update equation.

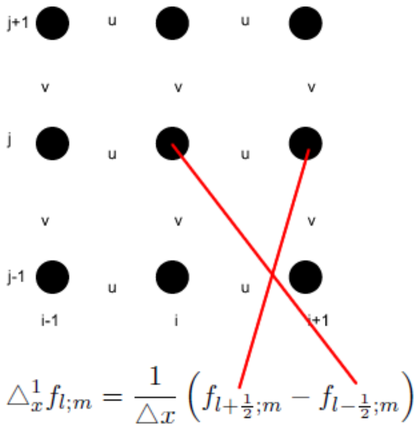


$$a^n_{i+\frac{1}{2};j} = u^n_{i+\frac{1}{2};j}\triangle^0_x u^n_{i+\frac{1}{2};j} + V^n_{i+\frac{1}{2};j}\triangle^0_y u^n_{i+\frac{1}{2};j}$$

$$\triangle^0_x f_{l;m} = \frac{1}{2\triangle x}\left(f_{l+1;m} - f_{l-1;m}\right)$$

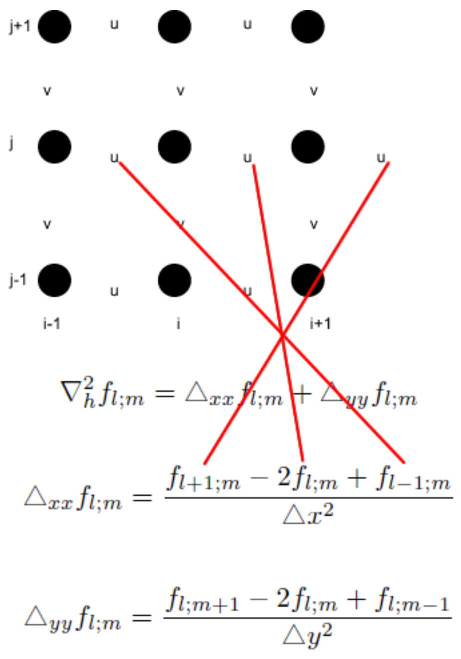
$$V_{i+\frac{1}{2};j} = \frac{1}{4}\left(v_{i+1;j+\frac{1}{2}} + v_{i;j+\frac{1}{2}} + v_{i+1;j-\frac{1}{2}} + u_{i;j-\frac{1}{2}}\right)$$

Figure 1: Advection term of the velocity update.



$$\triangle^1_x f_{l;m} = \frac{1}{\triangle x}\left(f_{l+\frac{1}{2};m} - f_{l-\frac{1}{2};m}\right)$$

Figure 2: Pressure term of the velocity update.

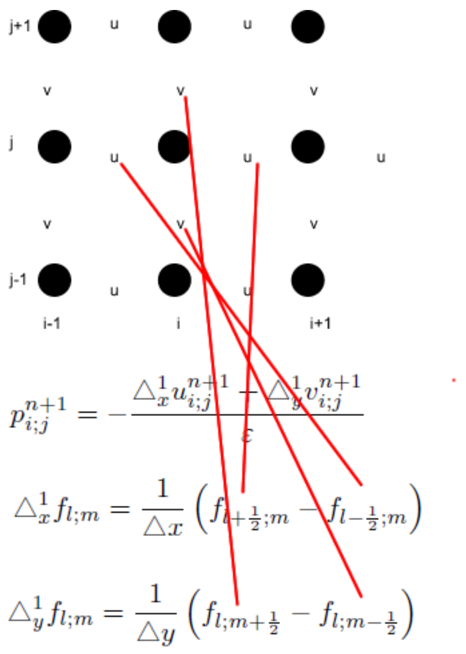


$$\nabla_h^2 f_{l;m} = \triangle_{xx} f_{l;m} + \triangle_{yy} f_{l;m}$$

$$\triangle_{xx} f_{l;m} = \frac{f_{l+1;m} - 2f_{l;m} + f_{l-1;m}}{\triangle x^2}$$

$$\triangle_{yy} f_{l;m} = \frac{f_{l;m+1} - 2f_{l;m} + f_{l;m-1}}{\triangle y^2}$$

Figure 3: Viscosity term of the velocity update.



$$p_{i;j}^{n+1} = -\frac{\triangle_x^1 u_{i;j}^{n+1} + \triangle_y^1 v_{i;j}^{n+1}}{\varepsilon}$$

$$\triangle_x^1 f_{l;m} = \frac{1}{\triangle x}\left(f_{l+\frac{1}{2};m} - f_{l-\frac{1}{2};m}\right)$$

$$\triangle_y^1 f_{l;m} = \frac{1}{\triangle y}\left(f_{l;m+\frac{1}{2}} - f_{l;m-\frac{1}{2}}\right)$$

Figure 4: Pressure update.

3.3 Boundary Conditions

Chen and Lobo describe two types of fixed boundary conditions in this paper. The first, external boundaries, are those that encase the fluid volume, or are at least on some edge of the volume, a wall or riverbed for example. The second, internal boundaries, are those that are encased by the fluid, like a post or a bridge. In addition to this, Chen and Lobo also implemented moving boundaries in their simulation. Moving boundaries would be needed to show the reaction of water to objects moving through the water, like a boat or buoy. Applying these boundary conditions to the velocity field requires a simple set velocity to a specific cell in the surface mesh.

3.4 Streak Lines and Floating Objects

To simulate streak lines in the fluid, particles are introduced into the system at several different origins and are allowed to flow freely on the surface. Their positions are updated by the value of the velocity field at their current positions. This results in streak lines due to particles following the flow of the fluid at the surface. This same logic can be used for free floating objects in the water. Their positions are simply updated by the surface velocity of the fluid at their position.

3.5 Stability

Essentially, this solver becomes more stable as its time steps and Reynolds numbers shrink, and as its penalty parameters grow. However, smaller time steps result in a slower simulation, smaller Reynolds numbers result in less turbulent or laminar waters, and large penalty numbers result in an inaccurate simulation. Balancing these values are essential to the stability and accuracy of the simulation.

4 Rapid, Stable Fluid Dynamics for Computer Graphics

The work of Kass and Miller [16] is based on three assumptions. The first assumption is that the water's surface can be represented by a height field. This limits the simulation to fluid without splashing or wave breaks. The second assumption is that the velocity of the water in the vertical direction can be ignored. This means that the more vertical or steep a wave gets the less accurate the simulation will be. The final assumption is that the horizontal velocity of the flow is nearly constant. This takes away the possibility for turbulent water. Though this approach is somewhat limited when it comes to certain fluidic phenomena, it makes up for these limitations with speed and stability.

4.1 Shallow Water Equations

The three assumptions stated above allow Kass and Miller to make use of the widely used Shallow Water Equations. These equations are extremely simplified versions of the Navier-Stokes equations. The equations are described as such:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + g\frac{\partial h}{\partial x} = 0 \tag{4.1}$$

$$\frac{\partial d}{\partial t} + \frac{\partial}{\partial x}(ud) = 0 \tag{4.2}$$

Where g is the acceleration due to gravity, $z = h(x)$ is the height of the water surface, $z = b(x)$ is the height of the ground, $d(x) = h(x) - b(x)$ is the water depth, and $u(x)$ is the horizontal velocity of a vertical column of water. From here, Kass and Miller make another assumption that the velocity of the liquid is small and the depth varies slowly. With these additional assumptions in mind, the equations can be further simplified. First, the fluid velocity is considered small enough that the second value of equation 4.1 can be ignored resulting in:

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} = 0$$

Now focusing on equation 4.2, expand out all terms to get:

$$\frac{\partial(h-b)}{\partial t}+d\frac{\partial u}{\partial x}+u\frac{\partial d}{\partial x}=0$$

or

$$\frac{\partial h}{\partial t}-\frac{\partial b}{\partial t}+d\frac{\partial u}{\partial x}+u\frac{\partial d}{\partial x}=0$$

The height of the surface on which the fluid rests never changes with respect to time, so the second term can be ignored, thus yielding:

$$\frac{\partial h}{\partial t}+d\frac{\partial u}{\partial x}+u\frac{\partial d}{\partial x}=0$$

Finally, the third term can be ignored, because velocity is very small and the depth is slowly varying. Equations 4.1 and 4.2 now become:

$$\frac{\partial u}{\partial t}+g\frac{\partial h}{\partial x}=0 \tag{4.3}$$

$$\frac{\partial h}{\partial t}+d\frac{\partial u}{\partial x}=0 \tag{4.4}$$

Combining equations 4.3 and 4.4 result in a final single equation that can later be descritized. To do so, first differentiate equation 4.3 with respect to distance.

$$\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial t}+g\frac{\partial h}{\partial x}=0\right)$$

$$\frac{\partial}{\partial x}\frac{\partial u}{\partial t}+\frac{\partial g}{\partial x}\frac{\partial h}{\partial x}+g\frac{\partial^2 h}{\partial x^2}=0 \tag{4.5}$$

$$\frac{\partial}{\partial x}\frac{\partial u}{\partial t}+g\frac{\partial^2 h}{\partial x^2}=0$$

The second term in equation 4.5 drops, because gravity is constant and, thus, its derivative is zero. Now differentiate equation 4.4 with respect to time.

$$\frac{\partial}{\partial t}\left(\frac{\partial h}{\partial t} + d\frac{\partial u}{\partial x} = 0\right)$$

$$\frac{\partial^2 h}{\partial t^2} + \frac{\partial d}{\partial t}\frac{\partial u}{\partial x} + d\frac{\partial}{\partial t}\frac{\partial u}{\partial x} = 0 \tag{4.6}$$

$$\frac{\partial^2 h}{\partial t^2} + d\frac{\partial}{\partial t}\frac{\partial u}{\partial x} = 0$$

The second term in equation 4.6 drops, because one of the assumptions is that the horizontal velocity is relatively constant, therefore its derivative can be considered zero. Next simply substitute for the cross derivatives to get the final equation describing the height of the fluid.

$$\frac{\partial^2 h}{\partial t^2} - gd\frac{\partial^2 h}{\partial x^2} = 0$$

or

$$\frac{\partial^2 h}{\partial t^2} = gd\frac{\partial^2 h}{\partial x^2}$$

4.2 Discretization

The equation above must be transform into something discrete if we are to calculate a solution computationally. To do this, Kass and Miller use a forward-differencing approach described by the following set of equations.

$$\frac{\partial h_i}{\partial t} = \left(\frac{d_{i-1} + d_i}{2\triangle x}\right)u_{i-1} - \left(\frac{d_i + d_{i+1}}{2\triangle x}\right)u_i$$

$$\frac{\partial u_i}{\partial t} = \frac{-g\left(h_{i+1} - h_i\right)}{\triangle x}$$

Where $\triangle x$ is the separation of the samples along the x direction. When these equations are put together they produce the discrete approximation of the final shallow water equation. To do so, first take the second derivative of the first equation:

$$\begin{aligned}\frac{\partial^2 h_i}{\partial t^2} = & \left(\frac{\partial}{\partial t}\right)\left(\frac{d_{i-1}+d_i}{2\triangle x}\right)u_{i-1} + \left(\frac{d_{i-1}+d_i}{2\triangle x}\right)\left(\frac{\partial u_{i-1}}{\partial t}\right) \\ & -\left(\frac{\partial}{\partial t}\right)\left(\frac{d_i+d_{i+1}}{2\triangle x}\right)u_i - \left(\frac{d_i+d_{i+1}}{2\triangle x}\right)\left(\frac{\partial u_i}{\partial t}\right).\end{aligned}$$

Assume the change in depth is constant and then substitute the change in velocity to get

$$\frac{\partial^2 h_i}{\partial t^2} = -g\left(\frac{d_{i-1}+d_i}{2(\triangle x)^2}\right)(h_i - h_{i-1}) + g\left(\frac{d_i+d_{i+1}}{2(\triangle x)^2}\right)(h_{i+1} - h_i).$$

4.3 Integration

The discrete fluid equation must be integrated to give the height of the fluid at each position in the map. Kass and Miller use a first-order implicit method, where $h(n)$ is the height at the nth iteration and dots above h denote differentiation with time. They write the first-order implicit equations as:

$$\frac{h(n) - h(n-1)}{\triangle t} = \dot{h}(n)$$

$$\frac{\dot{h}(n) - \dot{h}(n-1)}{\triangle t} = \ddot{h}(n)$$

Rearrange these equations to get the following:

$$\begin{aligned} h(n) = & \quad h(n-1) + \triangle t\dot{h}(n-1) && +(\triangle t)^2\ddot{h}(n) \\ h(n) = & \quad 2h(n-1) - h(n-2) && +(\triangle t)^2\ddot{h}(n) \\ h_i(n) = & \quad 2h_i(n-1) - h_i(n-2) && -g(\triangle t)^2\left(\frac{d_{i-1}+d_i}{2(\triangle x)^2}\right)(h_i(n) - h_{i-1}(n)) \\ & && +g(\triangle t)^2\left(\frac{d_i+d_{i+1}}{2(\triangle x)^2}\right)(h_{i+1}(n) - h_i(n)) \end{aligned}$$

The main goal here is to come up with a linear equation that can be rapidly solved. The equation above, however, is still not linear because d depends on h. To solve this problem, Kass and Gaven treat d as a constant. This fixes the velocity as a function of x and allows the height to be calculated using a symmetric tri-diagonal linear system. Before deriving the symmetric tri-diagonal linear matrix from the equation above lets first respectively abstract $g(\triangle t)^2$, $\left(\frac{d_{i-1}+d_i}{2(\triangle x)^2}\right)$, and $\left(\frac{d_i+d_{i+1}}{2(\triangle x)^2}\right)$ to A, B, and C. Then the equation becomes:

$$h_i(n) = 2h_i(n-1) - h_i(n-2) - AB\left(h_i(n) - h_{i-1}(n)\right) + AC\left(h_{i+1}(n) - h_i(n)\right)$$

Distribute A's, B's, and C's:

$$h_i(n) = 2h_i(n-1) - h_i(n-2) - ABh_i(n) + ABh_{i-1}(n) + ACh_{i+1}(n) - ACh_i(n)$$

Rearrange, so that all heights based off current step n are on the left side of the equality:

$$-ABh_{i-1}(n) + h_i(n) + ABh_i(n) + ACh_i(n) - ACh_{i+1}(n) = 2h_i(n-1) - h_i(n-2)$$

Now pull out like terms:

$$-ABh_{i-1}(n) + (1 + AB + AC)\,h_i(n) - ACh_{i+1}(n) = 2h_i(n-1) - h_i(n-2)$$

Notice that the left side of the equation is now organized in such a way that adding up all the terms is the same process as multiplying the height field by the following matrix:

$$\begin{pmatrix} (1+AB+AC)_0 & -AC_0 & & & \\ -AB_0 & (1+AB+AC)_1 & -AC_1 & & \\ & \ddots & \ddots & \ddots & \\ & & -AB_{n-3} & (1+AB+AC)_{n-2} & -AC_{n-2} \\ & & & -AB_{n-2} & (1+AB+AC)_{n-1} \end{pmatrix}$$

Label the above matrix as matrix M. Now the final discrete equation for animating the fluid can be described as the following linear equation:

$$Mh_i(n) = 2h_i(n-1) - h_i(n-2)$$

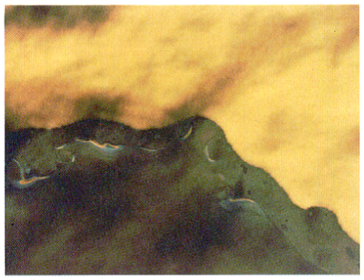
4.4 The Third Dimension

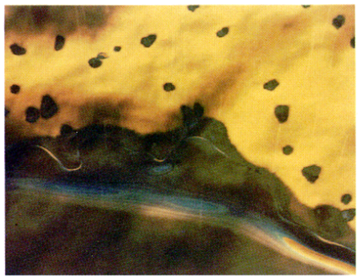
This approach can be used to simulate the animation of fluid in three dimensions as well as two. This is done by now solving a series of two dimensional equations and changing the second derivative of h with the Laplacian:

$$\frac{\partial^2 h}{\partial x^2} = gd\left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2}\right) = gd\nabla^2 h$$

4.5 Results

With this implementation, Kass and Miller have come up with a stable, real-time simulation for fluid animation. The algorithm is linear and highly parallel.





5 Stable Fluids

In this paper Stam [37] presents a model for solving the full Navier-Stokes set of equations with real-time-interactive frame rates and unconditional stability. The main idea behind this model is the use of the Helmholtz-Hodge Decomposition on the velocity field to project it onto its divergence free part.

5.1 Basic Equations

The Helmholtz-Hodge Decomposition states that any vector field w can uniquely be decomposed into the form:

$$w = \nabla \times (G(\nabla \times w)) - \nabla(G(\nabla \cdot w))$$

Where the first term has zero divergence and the second is a scalar field – G is a special type of integral called the Newtonian Potential. Using this the Navier-Stokes equations can be solved and the resultant velocity field can be projected onto its divergence free part – the projection operator being P:

$$P\left(\frac{\partial u}{\partial t}\right) = P\left(-(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \nabla^2 u + g\right)$$

or,

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

Notice that the pressure term dropped from the right side of the equation. This is due to the fact that the curl of the gradient of a scalar field is equal to zero. Also u is already divergence free, so applying the P operator to the left side of the equation will result in itself.

5.2 Method of Solution

Stam solves the above equation in four main steps, add forces, advect, diffuse, then project. If w_i is an intermediate state of the velocity field then this process can be illustrated by the following diagram.

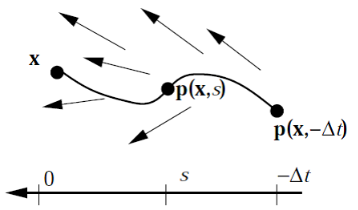
$$\mathbf{w}_0(\mathbf{x}) \overset{\text{add force}}{\overbrace{\longrightarrow}} \mathbf{w}_1(\mathbf{x}) \overset{\text{advect}}{\overbrace{\longrightarrow}} \mathbf{w}_2(\mathbf{x}) \overset{\text{diffuse}}{\overbrace{\longrightarrow}} \mathbf{w}_3(\mathbf{x}) \overset{\text{project}}{\overbrace{\longrightarrow}} \mathbf{w}_4(\mathbf{x}).$$

Add Force Updating the external force term is a simple addition between the previous state force and the current force multiplied by the change in time.

$$w_1(x) = w_0(x) + \triangle t f(x,t)$$

Advect To compute the advection step, Stam uses the Method of Characteristics. Essentially, a particle is traced from the current position back in time to a previous position. The new velocity at the current point is then set to the velocity that the particle had at its previous location $\triangle t$ ago.

$$w_2(x) = w_1(p(x, -\triangle t))$$



To prove that this approach is valid see section 5.2.1.

Diffuse The next step in the solution is the diffusion step. To solve this Stam uses an implicit method.

$$\frac{w_3(x) - w_2(x)}{\triangle t} = \nu \nabla^2 w_3(x)$$

$$w_3(x) - w_2(x) = \nu \triangle t \nabla^2 w_3(x)$$

$$w_3(x) - \nu \triangle t \nabla^2 w_3(x) = w_2(x)$$

$$(I - \nu \triangle t \nabla^2) w_3(x) = w_2(x)$$

Where I is the identity operator. This is known as a Poisson equation. To solve this first make the Laplacian operator discrete by changing it from

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

to

$$\frac{\partial^2 v}{\partial x^2} = \frac{v_{i-1,j,k} + 2v_{i,j,k} + v_{i+1,j,k}}{\triangle x^2}$$

$$\frac{\partial^2 v}{\partial y^2} = \frac{v_{i,j-1,k} + 2v_{i,j,k} + v_{i,j+1,k}}{\triangle y^2}$$

$$\frac{\partial^2 v}{\partial z^2} = \frac{v_{i,j,k-1} + 2v_{i,j,k} + v_{i,j,k+1}}{\triangle z^2}$$

where v is some three dimensional vector field, in this case w_3. If the grid sizes are equal in length then the equation can be further simplified to

$$\frac{\partial^2 v}{\partial r^2} = \left(\frac{1}{\triangle r^2}\right) [v * + 6v_{i,j,k}]$$

$$v* = v_{i-1,j,k} + v_{i+1,j,k} + v_{i,j-1,k} + v_{i,j+1,k} + v_{i,j,k-1} + v_{i,j,k+1}$$

Now that a discrete solution has been found, apply it to the diffusion step.

$$(I - \nu \triangle t \nabla^2) w_3(x) = w_2(x)$$

$$w_3(x) - \nu \triangle t \nabla^2 w_3(x) = w_2(x)$$

$$w_{3i,j,k}(i,j,k) - \left(\frac{\nu \triangle t}{\triangle r^2}\right)[w_3(i,j,k)^* + 6w_3(i,j,k)] = w_2(i,j,k)$$

$$\left(\frac{\triangle r^2}{\nu \triangle t}\right) w_3(i,j,k) - w_3(i,j,k)^* + 6w_3(i,j,k) = \left(\frac{\triangle r^2}{\nu \triangle t}\right) w_2(i,j,k)$$

$$\left(\frac{\triangle r^2}{\nu \triangle t} - 6\right) w_3(i,j,k) + w_3(i,j,k)^* = \left(\frac{\triangle r^2}{\nu \triangle t}\right) w_2(i,j,k)$$

The final equation could be thought of as a series of two dimensional linear equations of the form

$$Aw_3(x) = b$$

where

$$A = \begin{bmatrix} \left(\frac{\triangle r^2}{\nu \triangle t} - 6\right) & 1 & & & \\ 1 & & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & \left(\frac{\triangle r^2}{\nu \triangle t} - 6\right) \end{bmatrix}$$

$$b = \left(\frac{\triangle r^2}{\nu \triangle t}\right) w_2(x)$$

Now with a discrete and linear diffusion equation, apply one of many techniques for solving Poisson equations. For example, Gauss-Seidel Relaxation could be used to find a solution:

$$w_i^k = \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j} w_j^k - \sum_{j=i+1}^{n} a_{i,j} w_j^{k-1}\right)$$

Project The solution to the projection step is very similar to the diffusion step. This is because, using Helmholtz-Hodge Decomposition,

$$w_4(x) = w_3(x) - \nabla q$$

or

$$0 = \nabla \cdot w_3(x) - \nabla^2 q$$

Again, a Poisson equation of the form

$$0 = \nabla \cdot w_3(x) - \left(\frac{1}{\triangle r^2}\right)[q^* + 6q_{i,j,k}]$$

must be solved. Once q is found, subtract its gradient from $w_3(x)$ to get the final projection.

5.2.1 Method of Characteristics

The first order wave equation,

$$c\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0,$$

describes the movement of a wave in one direction with no change of shape. Using the method of characteristics, the partial differential equation can be converted into several ordinary differential equations, by expressing the curve in the $x - t$ plane parametrically. To do so, let

$$x = x(r)$$

and

$$t = t(r)$$

where r is distance along the curve. Therefore,

$$u(x,t) = u(x(r), t(r))$$

making u a function of r. u's derivative with respect to r would then be

$$\frac{du}{dr} = \frac{dx}{dr}\frac{\partial u}{\partial x} + \frac{dt}{dr}\frac{\partial u}{\partial t}.$$

Comparing this to the first order wave equation,

$$\frac{du}{dr} = 0,$$

when,

$$\frac{dx}{dr} = c$$

and,

$$\frac{dt}{dr} = 1.$$

This shows that u is constant along the characteristic curve, but may be different on different characteristic curves. Now integrate these equations to get

$$u = constant,$$

$$x = cr + x_0,$$

and

$$t = r.$$

Which implies that $x - ct = x_0$.

5.3 Diffusion and Projection in the Fourier Domain

Stam also proposes a more elegant solution, by solving the diffusion and projection steps in the Fourier domain. However, this solution will only work for fluids with periodic boundaries (boundaries that wrap around). Though fluids with periodic boundaries do not show up in nature, this technique still has its applications. For example, a fluid simulation could be solved for one specific portion of a plane, and then be tiled to cover the rest.

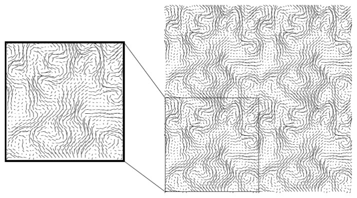


Figure 5: Because of the periodic boundaries of this solver, fluid flows can be tiled along surfaces.

For this approach the method of solution goes as follows:

1. Add force field
2. Advect velocity
3. Transform to Fourier domain
4. Diffuse velocity
5. Project velocity
6. Inverse transform from Fourier domain

Because the first two steps of this approach are the same as what has been described before they will not be described again here.

5.3.1 The Fast Fourier Transform (FFT)

Stam does not implement his own FFT solver, due to the fact that there are many "black box" solvers that can be easily found and do the job efficiently. Instead, he makes use of MIT's FFTW [5].

For this very reason further detail about the FFT will be omitted from this survey. Essentially, the FFT can take a set of values (a velocity field for example) from the time domain to the frequency domain where certain calculations become much easier. For example, in the Fourier domain most differential equations become algebraic.

5.3.2 Diffusion in the Fourier Domain

After transforming into the Fourier domain, higher spacial frequencies are filtered out by a filter whose decay depends on the magnitude of the wave direction, viscosity and the time step denoted

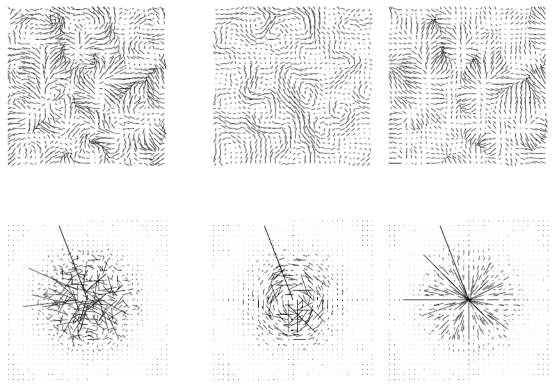
$$-k^{2*visc*\delta t}.$$

Here is a code snippet from Stam's solver that does just this:

```
for ( i=0 ; i<=n ; i+=2 ) {
   x = 0.5*i;
   for ( j=0 ; j<n ; j++ ) {
      y = j<=n/2 ?  j :  j-n;
      r = x*x+y*y;
      if ( r==0.0 ) continue;
      f = exp(-r*dt*visc);
      U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
      U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
      u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
      u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
      v0[i+  (n+2)*j] = f*(   -y*x/r *U[0] + (1-y*y/r)*V[0] );
      v0[i+1+(n+2)*j] = f*(   -y*x/r *U[1] + (1-y*y/r)*V[1] );
   }
}
```

5.3.3 Projection in the Fourier Domain

Just as explained earlier Stam uses the Helmholtz Hodge Decomposition to project onto a final, stable solution for the fluid. Only now the decomposition is applied in the Fourier domain. Here is an example of how decomposition might compare between the two domains:



Helmholtz Hodge Decomposition in time (top row) and frequency (bottom row) domains. The left most column being the initial velocity field, middle - the divergence free field, and right most - the gradient field.

Notice how each velocity vector in the gradient field is parallel to the wave direction and in the mass conserving field it is perpendicular. This means that projecting the velocity onto the mass conserving field requires a simple projection onto the plane parallel to the direction of the wave. This is shown by the following code snippet:

```
U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
v0[i+  (n+2)*j] = f*(   -y*x/r *U[0] + (1-y*y/r)*V[0] );
v0[i+1+(n+2)*j] = f*(   -y*x/r *U[1] + (1-y*y/r)*V[1] );
```

After projection, the velocity field is then inversely transformed from the frequency domain back into the time domain and is used as the solution field for the current time step.

5.4 Results

Stam has implemented both a two and three dimensional fluid solver at interactive rates. Figures 6 and 7 were taken from the three dimensional solver on an SGI Octane workstation with R10K processor and 192 Mbytes of memory.

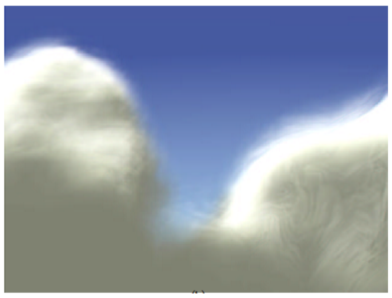


Figure 6

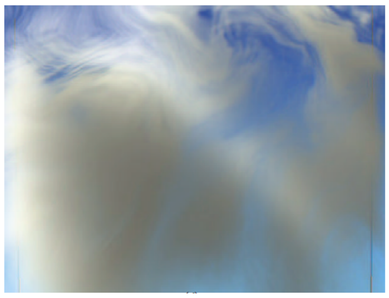


Figure 7

6 Particle-Based Fluid Simulation for Interactive Applications

In [27], Muller et al. propose a particle-based approach, based on Smoothed Particle Hydrodynamics (SPH) to animate arbitrary fluid motion. Before this paper, SPH had been used to depict fire and other gaseous phenomena, as well as highly deformable bodies. This paper extends this method focusing on the simulation of fluids.

6.1 Smoothed Particle Hydrodynamics

SPH is an interpolation method for particle systems. This method calculates some value for a particle at a specific point in space by distributing quantities from neighboring particles using radial symmetrical smoothing kernels. A scalar quantity A of a particle is interpolated at location r by a weighted sum of contributions from all particles. This is denoted

$$A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} W\left(r - r_j, h\right)$$

where j iterates over all particles, m_j is the mass of particle j, r_j its position, ρ_j the density and A_j the field quantity at r_j. The function $W\left(r, h\right)$ is called the smoothing kernel with core radius h. The smoothing kernel is essentially a function that, when applied, dictates how much a value from a certain particle at r_j will impact the queried position r, based off their distance from each other and an arbitrary falloff point h. The Gaussian bell curve would be a good example of this (see following figures).

$$W(r, h) = \frac{1}{(2\pi h^2)^{\frac{3}{2}}} e^{-\frac{r^2}{2h^2}}, h > 0$$

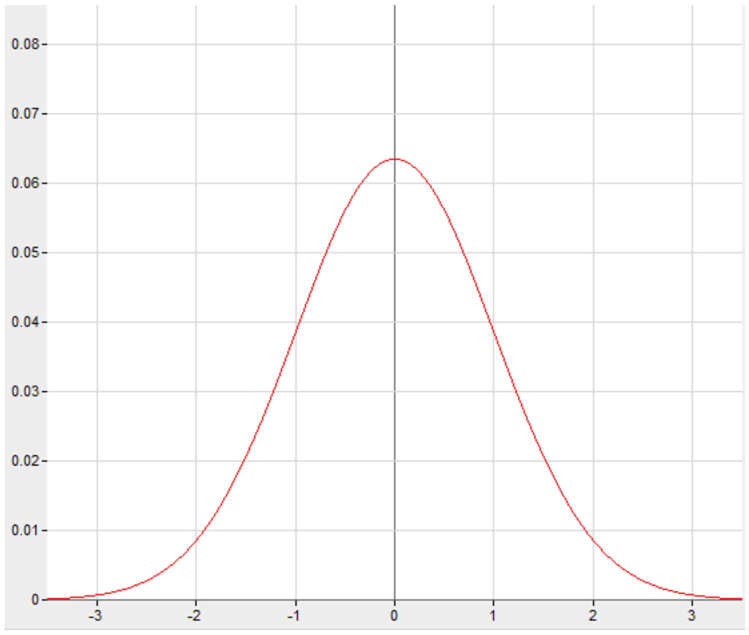


Figure 8: A standard normalized Gaussian bell curve.

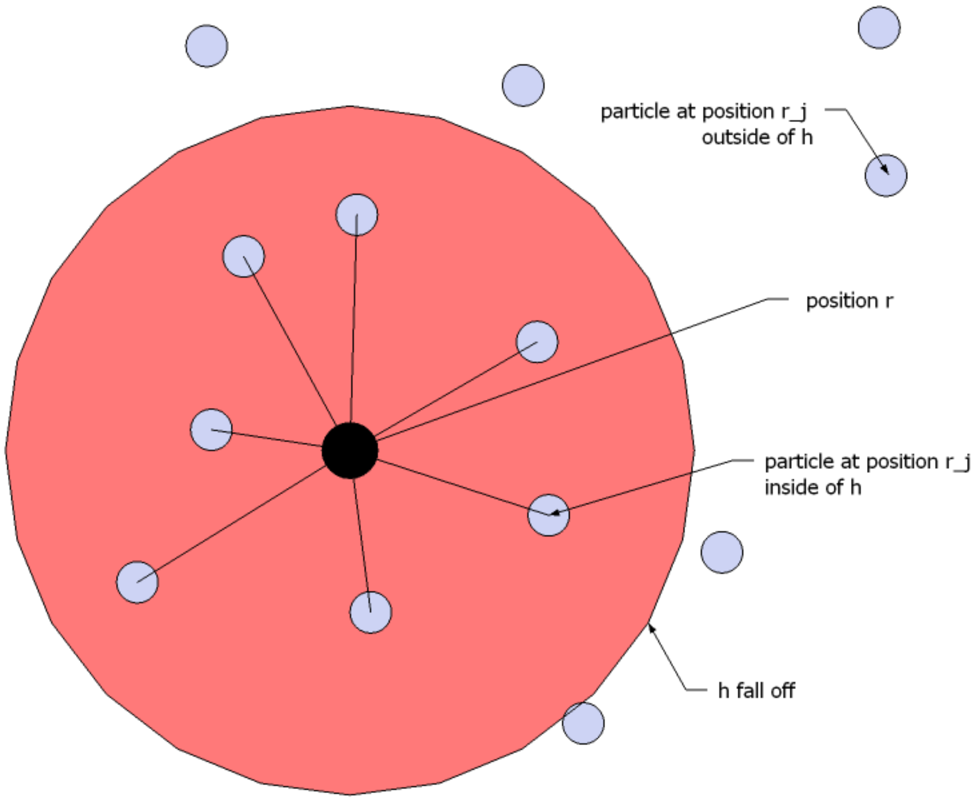


Figure 9: A visual representation of the interpolation of some value at position r, between particles within a smoothing kernel's radius h.

An example of calculating the density of a specific particle using this technique would look like the following:

$$\rho_S(r) = \sum_j m_j \frac{\rho_j}{\rho_j} W\left(r - r_j, h\right) = \sum_j m_j W\left(r - r_j, h\right).$$

The derivative of a value would look like this:

$$\nabla A_S(r) = \sum_j \nabla m_j \frac{A_j}{\rho_j} W\left(r - r_j, h\right)$$

$$\nabla A_S(r) = \sum_j \left(\nabla m_j \frac{A_j}{\rho_j} W\left(r - r_j, h\right) + m_j \frac{A_j}{\rho_j} \nabla W\left(r - r_j, h\right) \right)$$

$$\nabla A_S(r) = \sum_j \left(0 \times W\left(r - r_j, h\right) + m_j \frac{A_j}{\rho_j} \nabla W\left(r - r_j, h\right) \right)$$

$$\nabla A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W\left(r - r_j, h\right)$$

6.2 Modeling Fluids with Particles

Like most fluid simulation approaches, start with the Navier-Stokes equations:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho v) = 0$$

and,

$$\rho \left(\frac{\partial v}{\partial t} + v \cdot \nabla v \right) = -\nabla p + \rho g + \mu \nabla^2 v.$$

And like other approaches simplify the equations down to something more manageable. Because this approach is particle based, mass is guaranteed not to changed and therefore the first equation can be ignored completely. Also, because the particles move with the fluid and the velocity and density of the fluid at any given point is tied to each particle, momentum is guaranteed to be conserved and the advection portion of the equation $\left(\frac{\partial v}{\partial t} + v \cdot \nabla v\right)$ can be reduced to just the time derivative $\frac{\partial v}{\partial t}$. This means that

$$\rho\left(\frac{\partial v}{\partial t}\right) = -\nabla p + \rho g + \mu\nabla^2 v,$$

which implies that the acceleration of a particle i is:

$$a_i = \frac{\partial v_i}{\partial t} = \frac{f_i}{\rho_i}$$

where f_i is a summation of all the forces on the right hand side of the original equation. The following sections will define how to go about calculating f_i.

6.3 Pressure

When calculating pressure with SPH, one would think to simply plug in the pressure term $-\nabla p$ into the SPH equation, yielding:

$$-\nabla p(r_i) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W\left(r_i - r_j, h\right).$$

However, this is not symmetric (think of the case with only two particles of different pressure values). To solve this use:

$$f_i^{pressure} = -\nabla p(r_i) = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W\left(r_i - r_j, h\right)$$

where the SPH equation now yields the mean of the pressures of interacting particles.

6.4 Viscosity

Using SPH for the viscosity term, also, is not symmetrical.

$$\mu\nabla^2 v(r_i) = \mu\sum_j m_j \frac{v_j}{\rho_j} \nabla^2 W\left(r_i - r_j, h\right).$$

An intuitive solution is to change the above equation to

$$f_i^{viscosity} = \mu \nabla^2 v(r_i) = \mu \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W\left(r_i - r_j, h\right).$$

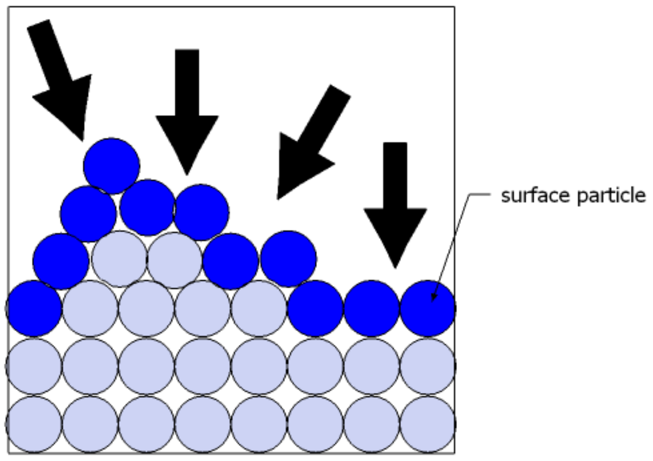
This makes sense because viscosity is solely based on the difference between velocities. Using this a single particle is accelerated by the velocities relative to its surrounding position.

6.5 External Forces

For calculating accelerations due to external forces, such as gravity, SPH is not necessary. For this model the accelerations are simply added directly to the affected particle itself.

6.6 Surface Tension

Though surface tension does not show up in the Navier-Stokes equations, it is important to include in the model. Surface tension is a force that pushes into the fluid volume and in doing so minimizes its curvature.



Unstable forces push down on the surface particles – minimizing their curvature.

To apply this force first identify the surface of the fluid using the following SPH equation:

$$c_S(r) = \sum_j m_j \frac{1}{\rho_j} W\left(r - r_j, h\right).$$

Now apply the gradient operator to convert to a field of vectors pointing into the surface.

$$n = \nabla c_S$$

Then find the divergence of the vector field to obtain a field of scalars that define the curvature of the surface (the negative ensures positive values for convex fluid volumes).

$$\kappa = \frac{-\nabla \cdot \nabla c_S}{|n|}$$

or

$$\kappa = \frac{-\nabla^2 c_s}{|n|}$$

Put these two equations together to get the surface tension:

$$f^{surface} = \sigma \kappa n$$

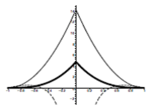
where σ is a surface tension coefficient that depends on the two fluids that form the surface.

6.7 Smoothing Kernels

Just as most grid-based fluid solvers use different differencing schemes for updating velocities, particle-based fluid solvers use what are called smoothing kernels. Muller et al. in [27] use three smoothing kernels for all of its SPH equations.

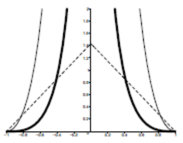
For pressure computations use the following:

$$W_{spiky}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h-r)^3 & 0 \le r \le h \\ 0 & otherwise \end{cases}$$



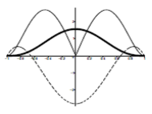
For viscosity computations:

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3}\begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \le r \le h \\ 0 & otherwise \end{cases}$$



All other SPH computations:

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9}\begin{cases} \left(h^2 - r^2\right)^3 & 0 \le r \le h \\ 0 & otherwise \end{cases}$$



It is important to note that in the W_{poly6} kernel r only shows up as r^2 – reducing the number square-root calculations. However, W_{poly6}'s gradient goes to zero when r is zero, which isn't a very good characteristic for measuring pressure in the fluid. This is because for very close particles, the forces that push them away goes to zero resulting in a clumping of particles. To solve this Muller et al. use W_{spiky} which has a gradient that is always positive.

The gradient is not the only problem with the W_{poly6} kernel. Another problem is that its Laplacian changes sign. This becomes a problem when diffusing the fluid. If this kernel were to be used, some particles, depending on their relative distances, would become accelerated, which is not the proper behavior. To solve this problem Muller et al. make use of what they call $W_{viscosity}$, the Laplacian of which is always positive.

6.8 Surface Tracking and Visualization

In order to visualize the fluid surface some sort of mesh must be defined from the mass of points. In [27] Muller et al. implement two different algorithms, point splatting and marching cubes. Between the two, the point splatting approach is fast, while the marching cubes approach is accurate.

6.9 Point Splatting

Point splatting can by done numerous ways, but in general the algorithm entails projecting the surface points into screen space and blending colors according to basis functions. Muller et al. reference techniques by Zwicker et al. in [45].

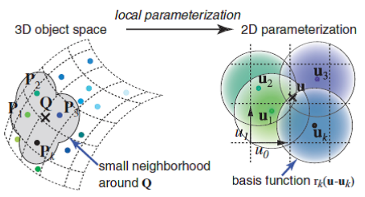

local parameterization
3D object space
2D parameterization
small neighborhood around Q
basis function $r_k(\mathbf{u}-\mathbf{u}_k)$

Figure 10

As seen in Figure 10 each point has a small radius where neighboring colors are sampled and filtered. Once the circle is projected into screen space, however, its shape becomes an ellipse. Because of this an Elliptical Weighted Average formulation is used (EWA) from [10].

6.10 Marching Cubes

The marching cubes algorithm [21] constructs a mesh from as set of points identified as the surface of the object being rendered. This is done by grabbing eight neighboring points and forming a cube around them. Depending on where the points are located in the cube, one of 256 possible surface intersections can be possible ($2^8 = 256$, where 8 is the number of sides of the cube). Using rotational and translational symmetry the possible surface intersections can be further simplified to the 15 cases shown in Figure 11.

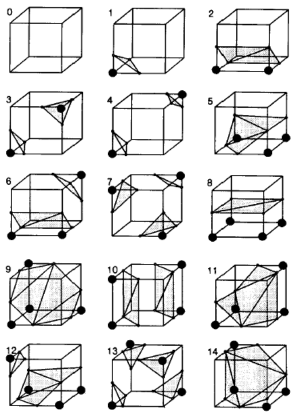


Figure 11

As the algorithm 'marches' from cube to cube it determines which of these 15 cases best fits how the surface intersects each cube. The best case is chosen and the proper vertices and surface normals are calculated to represent the final mesh.

6.11 Results

The liquid in Figure 12 was sampled with 2200 particles. Image (a) shows the individual particles, while (b) and (c) are rendered using the point splatting and marching cubes techniques, respectively. Running on a 1.8 GHz Pentium IV PC with a GForce 4 graphics card, (a) and (b) achieve 20 frames per second, while (c) runs at 5. Though using the marching cubes technique is much slower than point splatting, it yields a much more realistic representation of the fluid.

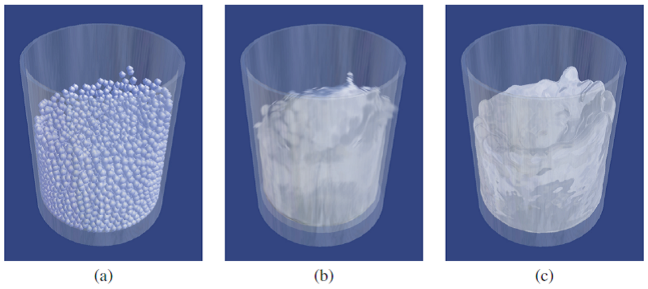


Figure 12

7 Hypertexture

Another approach for rendering fluids is with the use of noise-based hypertextures. A hypertexture is essentially a volumetric texture animated by some arbitrary algorithm. Perlin et al. [31] define their hypertextures as density regions between the surface of an object and the non-surface. They do this because infinitesimally thin surfaces are insufficient to properly define realistic surfaces. Furry, or corroded surfaces are good examples of this, see Figure 13.

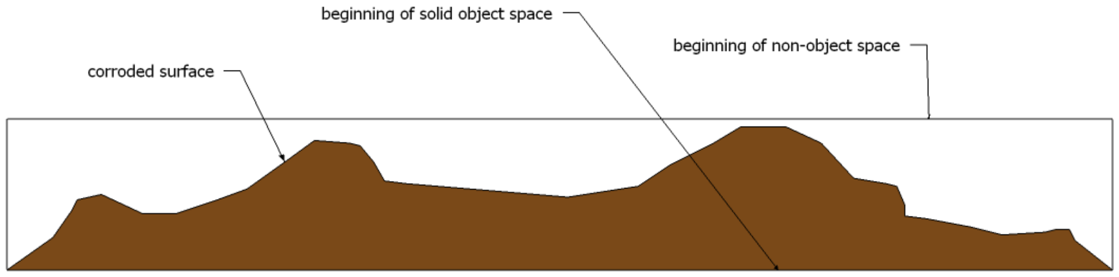

beginning of solid object space
beginning of non-object space
corroded surface

Figure 13: Example of a hypertexture volume that represents a corroded surface.

With this approach objects are no longer modeled as connected surfaces, but as a distribution of density. The base shape of the density distribution is broken up into a completely solid region and a malleable region where, in the malleable region, the surfaces can be deformed. This can be applied to fluids, by defining functions that deform the malleable surface over time according to the CFD equations.

7.1 Modeling Hypertexture

Perlin et al. hypertextures are defined by two functions. The first being an Object Density Function, $D(x)$, with range $[0, 1]$, which describes the 3D shape of the object for all positions x in R^3 space. In other words the malleable region of a given object consists of all x such that $0 < D(x) < 1$. The second is a Density Modulation Function f_i, which is used to modulate the density of a given object within its malleable region. Using these two equations the hypertexture can be defined like so:

$$H(D(x),x) = f_n(...f_2(f_1(f_0(D(x)))))$$

An example of the density function that might define a spherical geometry with position, radius, and softness c, r, and s could look something like this:

$$D_{[c,r,s]}(x) = \begin{cases} r_1^2 := (r - s/2)^2 \\ r_0^2 := (r + s/2)^2 \\ r_x^2 := (x_x - c_x)^2 + (x_y - c_y)^2 + (x_z - c_z)^2 \\ D := \begin{array}{c} if(r_x^2 \leq r_1^2) then\{1.0\} else \\ if(r_x^2 \geq r_0^2) then\{0.0\} else \left\{ \frac{(r_0^2 - r_x^2)}{(r_0^2 - r_1^2)} \right\} \end{array} \end{cases}$$

where r_0 is the outer ($D = 0$) boundary, r_1 is the inner ($D = 1$) boundary and r_x is the radius of the sphere at the point x.

7.2 Base Density Modulation Functions

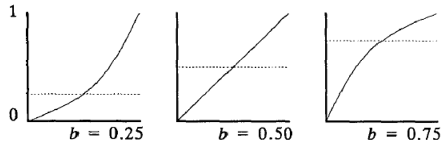
Perlin et al. base all higher level modulation functions off of four main base modulation functions – bias, gain, noise, and turbulence.

7.3 Bias

Bias is used to push up or pull down density values between zero and one. If the desired bias is b then $bias_b(0) = 0$, $bias_b(\frac{1}{2}) = b$, and $bias_b(1) = 1$. Using $bias_{\frac{1}{2}}$ as the identity for the bias function it can be defined as the power function

$$t^{\frac{ln(b)}{ln(0.5)}}.$$

See the following figure for a visual representation of different bias functions.



Different *bias* curves

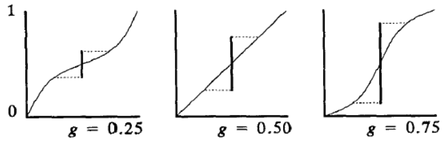
7.4 Gain

Gain is used to flatten or steepen the a density gradient. Perlin et al. define their gain function as $gain_g(0) = 0$, $gain_g(1/4) = (1-g)/2$, $gain_g(1/2) = 1/2$, $gain_g(3/4) = (1+g)/2$, and $gain_g(1) = 1$. This can be defined as a spline of two bias curves:

$$if(t < 0.5)then\{bias_{1-g}(2t)/2\}$$

$$else\{1 - bias_{1-g}(2 - 2t)/2\}.$$

As shown in the following figure.



Different *gain* curves

7.5 Noise

The third base density modulation function is noise. Perlin et al. sum pseudo-random spline knots of each point on the integer lattice in R^3. The knot $\Omega_{i,j,k}$ consists of a pseudo-random linear gradient $\Gamma_{i,j,k}$ smoothed by a drop off function $\omega(t)$.

$$\Omega_{i,j,k}(u, v, w) = \omega(u)\omega(v)\omega(w)(\Gamma_{i,j,k} \cdot (u, v, w))$$

where

$$\omega(t) = if(|t| < 1)then\{2|t|^3 - 3|t|^2 + 1\}else\{0\}.$$

Using this the noise at point (x, y, z) is equal to

$$\sum_{i=x}^{x+1}\sum_{j=y}^{y+1}\sum_{k=z}^{z+1}\Omega_{i,j,k}(x - i, y - j, z - k).$$

$\Gamma_{i,j,k}$ is found by hashing (i, j, k) to create an index into a precomputed gradient table G

$$\Gamma_{i,j,k} = G\{\phi(i + \phi(j + \phi(k)))\}$$

where $\phi(i) = P[i_{mod(n)}]$, and P is a precomputed array containing a pseudo-random permutation of the first n integers, G is a precomputed array of n pseudo-random vectors, uniformly distributed on the unit sphere, and n is the length of the P and G arrays.

7.6 Turbulence

Using the previously described noise function, the last base function, turbulence, can be described like so:

$$\sum_i abs\left(\frac{1}{2^i} noise\left(2^i x\right)\right)$$

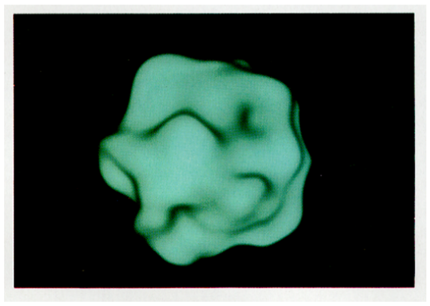
Perlin et al. point out that this is not a true turbulence function, but merely a method of simulating turbulent activity.

7.7 Higher Level Functions

Using these base functions, different phenomena can be simulated. For example, the density equation

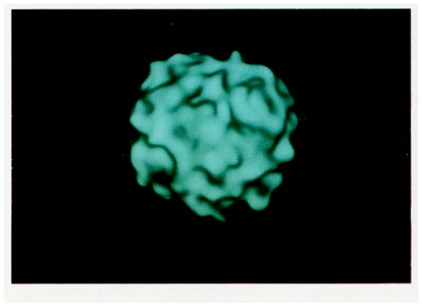
$$D(x) = sphere\left(x\left(1 + \frac{1}{f} noise\left(fx\right)\right)\right),$$

where f is the frequency and $\frac{1}{f}$ is the amplitude, results in a 'noisy sphere'; see following figure.

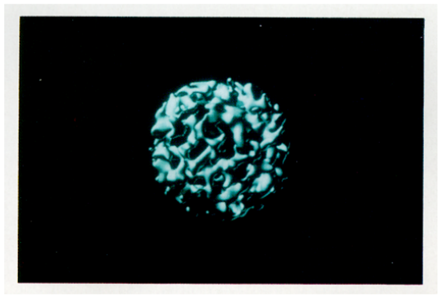


noisy sphere

Changing the frequency and amplitude results in the following:

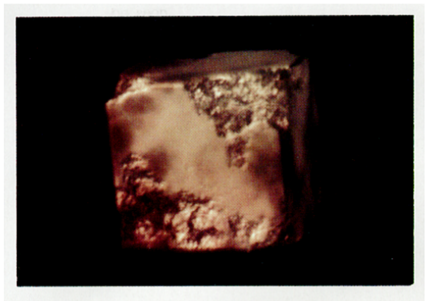


high-frequency noisy sphere

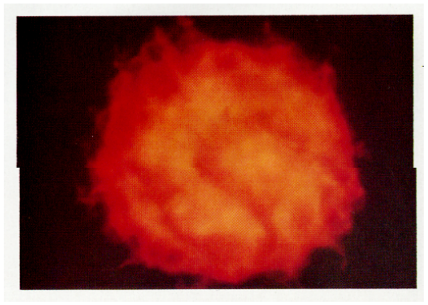


high-amplitude noisy sphere

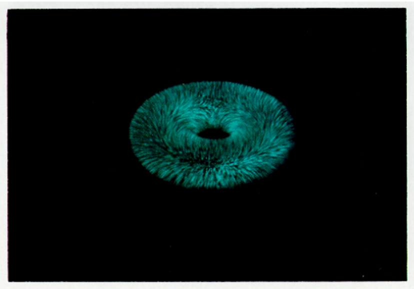
Perlin et al. composed different base functions to result in many different phenomena. Here are a few more examples of what can be simulated using this technique.



eroded cube



fire ball


furry donut

7.8 Results

Hypertextures make it easy to model objects that cannot be simply defined by their surface alone. Unlike traditional models, where objects are defined by their surface alone, hypertextures define the density of the object at every position in R^3. Though the run-time of this model is $O(n^3)$, it is highly parallel, and an increase of processor count results in a linear decrease in execution time of the algorithm.

8 Worthy Mentions

The field of fluid simulation is vast and there are many papers describing different solvers for fluids under different conditions. Most of which build upon the fundamentals of other research papers highlighted in this survey; nevertheless, they are worth mentioning for future research pursuits.

Shallow Waves In [42] Wang et al. describe a framework for solving General Shallow Wave Equations (GSWE). This paper extends the GSWE to solve for external forces such as gravity a surface tension, as well as interactions with rigid bodies.

Deep Water The deep water solver of Jensen et al. in [15] is presented as a solution to simulating deep ocean waves and interactions. This solver is a hybrid of three different solvers; Fast Fourier Transform (FFT), Navier-Stokes Equations (NSE), and Shallow Water Equations (SWE).

Viscus Fluids In this paper [44] Yongning Zhu and Robert Bridson extending an existing particle-based fluid simulator to accommodate inter-grain and boundary friction thus yielding a sand simulator.

Turbulence Many papers focus on solver fluid in its most general cases. This paper [18], however, focuses on fluid when it's in its most turbulent form specifically focused on producing small-scale detail.

Streams and Flows Here there is a focus on moving water, whether it be as stream flowing down a bank [23] or poured through the air onto a surface [14].

Part II

Implementations

The rest of the paper will be focusing on simulations based off of the implementation of solvers mentioned in the survey part of the paper. Two types of simulations (3D liquid surface and 3D liquid volume) will be discussed, using three different solvers (height field, grid, and particle).

All implementations were written using DirectX 10.1 using HLSL 4.0

9 3D Liquid Surface

9.1 Height Field Solver

The Height field solver is by far the simplest of the three implementations described in this paper. This solver is based on the work presented by Matthias Müller-Fischer as a part of a Game Developers Conference (GDC) in 2008 [28]. Given an initial state, the change in the height at each point in the field can be calculated by taking into account the constant speed at which the waves travel and the current height of each point being calculated. The equation looks something like the following:

$$f = c^2 * (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j})/h^2$$

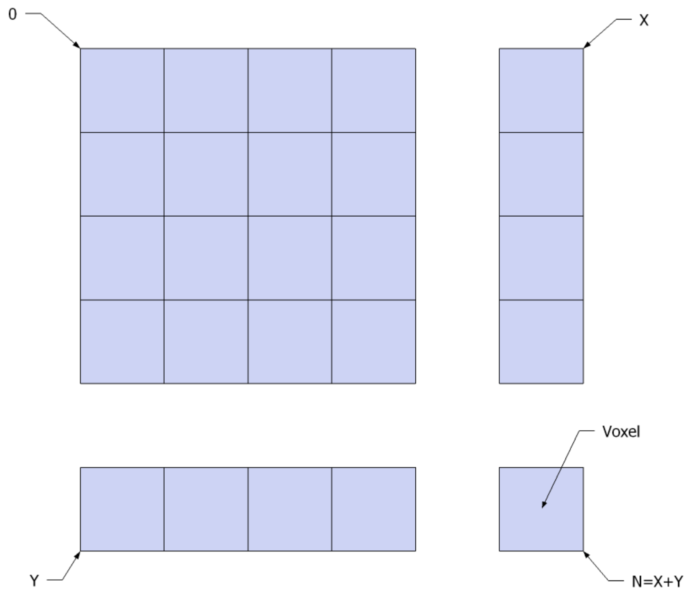
where c is the constant speed at which waves in the simulation move, u is the height field, h is the width of each fluid column, and f is the resultant force.

9.2 Grid-Based Solver

This, like the height-field solver, is a simulation of a three dimensional liquid surface. However, for this implementation, the liquid surface height will not be calculated from it's height at a previous time step, but will be inferred from the characteristics of each voxel in the solver's domain as fluid passes through it. The overarching approach for this implementation is very similar to the Chen and Lobo [3] approach where they implement a two dimensional grid-based solver and achieve the third dimension of height from the pressure at each voxel in the domain. This implementation will follow this approach but will replace the Chen and Lobo solver with Stam's Stable Solver.

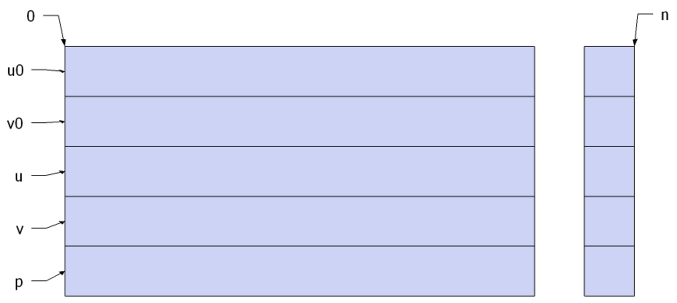
9.2.1 Data and Structures

This implementation takes up linear space in memory relative to the solution grid's domain. The liquid solver needs five buffers of size n, where n is the number of voxels in the domain of the solver.



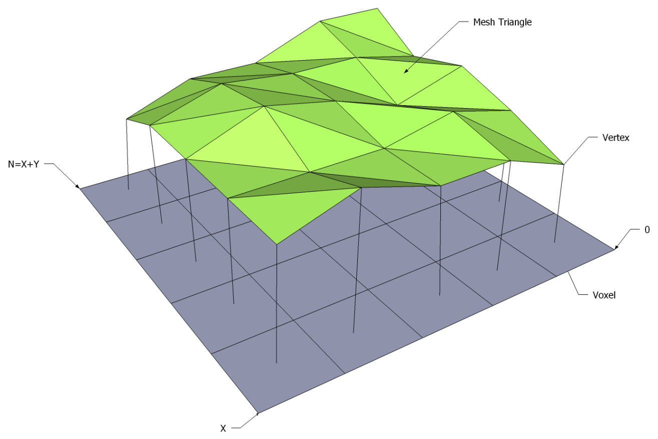
A top-down view of the voxelized two dimensional surface.

These buffers are u_0, v_0, u, v, and p, where u_0 and v_0 are the velocity in the x and y direction (respectively) at time n, and u, v and p are the velocity in the x and y direction and the pressure at time $n+1$. Two buffers are needed for both the velocity in the x direction and velocity in the y direction, because the velocity of each voxel at time $n+1$ is calculated based of it's velocity at time n. A simple 'ping-pong' method is used for each time step to avoid copying of the velocity data. Instead, the buffer pointers are simply swapped each time step.



Buffers for the simulator.

To properly visualize the liquid surface additional data is required – namely mesh data. The structure of Stam's solver is very simplistic in that it allows easy integration from a one-to-one mapping of vertices to voxels, where each vertex position is a representation of the pressure and velocity at that point in the field (see image below).



Mesh representation of pressure and velocity at each position (The green is a visualization of the liquid surface mesh, while the blue is a visualization of the solver's two dimensional domain)

9.2.2 Implementation of Stam's Solver

Though near linear time when implementing this solver in the Forier domain is possible to achieve, the ability to have boundaries is lost. This isn't very realistic when trying to simulate a liquid surface, so for this implementation Forier domain calculations are not used. The downside of not simulating in the Forier domain is that the simulation is now a $O(n^2)$ solver.

As shown in the survey section of this paper, Stam's solver consists of four steps that are executed each frame of the simulation – add force, advect, diffuse, and project. The following goes into the implementation of said steps.

External Forces Adding external forces is easily the simplest step in the solver and involves adding in the desired external forces multiplied by the time step of the solver. Note that though the liquid surface is three dimensional he liquid solver is a two dimensional impementation so all external forces will be represented as two dimensional vectors.

```
void AddForce(int N, float *x, float *s, float dt) {
  for(int i = 1; i <= N; ++i) {
    for(int j = 1; j <= N; ++j) {
      x[IX(i, j)] += s[IX(i, j)] * dt;
    }
  }
}
```

Advection For the advection step implement the Method of Characteristics as described in Stam's paper. The new velocity at any given voxel can be calculated by tracing a particle back a time step by its current velocity. The new velocity will be the interpolation of the surrounding voxels.

```
void FluidSolver::Advect(
  int N,
  int b,
  float * d,
  float * d0,
  float * u,
  float * v,
  float dt) {

  int i, j, i0, j0, i1, j1;
  float x, y, s0, t0, s1, t1, dt0;
  dt0 = dt*N;
  for(i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
      x = i-dt0*u[IX(i,j)];
      y = j-dt0*v[IX(i,j)];

      if(x<0.5) x = 0.5;
      if(x>N+0.5) x = N + 0.5;
      i0 = (int)x;
      i1 = i0 + 1;

      if(y<0.5) y = 0.5;
      if(y>N+0.5) y = N + 0.5;
      j0 = (int)y;
      j1 = j0 + 1;

      s1 = x-i0; s0 = 1-s1; t1 = y-j0; t0 = 1-t1;
      d[IX(i,j)] =
        s0*(t0*d0[IX(i0,j0)] +
          t1*d0[IX(i0,j1)]) +
        s1*(t0*d0[IX(i1,j0)] +
          t1*d0[IX(i1,j1)]);
    }
  }
  HandleBoundary(N, b, d);
}
```

Once the new velocity is found boundary conditions must be checked.

Diffusion To properly implement the diffusion step a Poisson solver must be used. For this implementation the most straight forward of solutions is used – Gauss-Seidel Relaxation.

```
1  void Diffuse(int N, int b, float *x, float *x0, float diff, float dt) {
2    int i, j, k;
3    float a=dt*diff*N*N;
4    for ( k=0 ; k<2 ; k++ ) {
5      for ( i=N ; i>=1 ; i-- ) {
6        for ( j=N ; j>=1 ; j-- ) {
7          temp[IX(i,j)] = (x0[IX(i,j)] + a * (
8            x[IX(i-1,j)]+
9            x[IX(i+1,j)]+
10           x[IX(i,j-1)]+
11           x[IX(i,j+1)]
12         ))/(1+4*a);
13       }
14     }
15     HandleBoundary ( N, b, temp );
16     SWAP(this->temp, x);
17   }
18 }
```

On line four of the source code above one can see that k is set to two. This is the number or 'relaxation' steps the solver takes to diffuse. This number can be tweaked to the liking to the user, but the higher the number the slower the diffusion step. Higher than two k values yielded insignificant accuracy gains.

Projection Most Eulerian solvers are subject to the same drawback in that under certain conditions they can become unstable and diverge. One of the unique qualities of Stam's Stable Solver is that it is unconditionally stable. This is due to its projection step where the solution grid is projected onto its divergence free part using Helmholtz-Hodge Decomposition.

Recall that the Helmholtz-Hodge Decomposition states:

$$w = u + \nabla q$$

where w is an initial vector field and u is mass conserving. Using Helmholtz-Hodge Decomposition solve the vector field for u in three steps. First q must be solved for and to do that the divergence operator needs to be applied to both sides of the decomposition equation:

$$\nabla \cdot w = \nabla \cdot (u + \nabla q)$$

```
for(i = 1; i <= N; i++) {
  for (j = 1; j <= N ; j++) {
    // Store result in this as a temp buffer
    v0[IX(i,j)] = -0.5f * h * (
      u[IX(i+1,j)] - u[IX(i-1,j)] +
      v[IX(i,j+1)] - v[IX(i,j-1)]
    );

    // Zero this buffer so we can 'ping-pong' with u0.
    u0[IX(i,j)] = 0;
  }
}
```

Because u is mass conserving, $\nabla \cdot u = 0$. This means that after applying the gradient operator a Poisson equation is left over, which can be solved, once again, with a Gauss-Seidel relaxation scheme.

$$\nabla \cdot w = \nabla^2 q$$

```
for(k = 0; k < 2; k++) {
  for (i = N; i >= 1; i--) {
    for (j = N; j >= 1; j--) {
      temp[IX(i,j)] = (
        v0[IX(i,j)] +
        u0[IX(i-1,j)] +
        u0[IX(i+1,j)] +
        u0[IX(i,j-1)] +
        u0[IX(i,j+1)]
      ) / 4.0f;
    }
  }
  HandleBoundary ( N, 0, temp );
  SWAP(this->temp, u0);
}
```

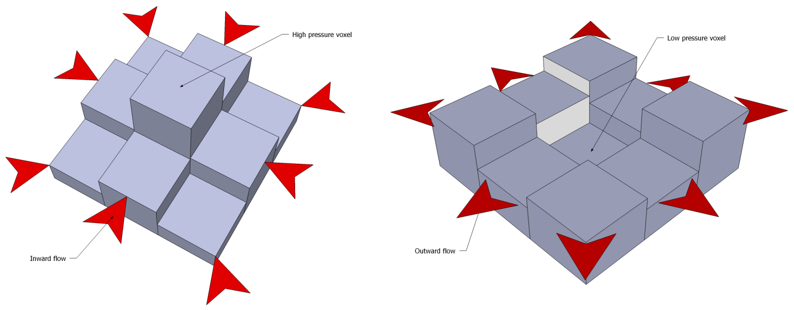
Now that q has been solved for its gradient can be subtracted from w leaving a mass conserving vector field.

$$w - \nabla q = u$$

```
for(i = 1; i <= N; i++) {
  for(j = 1; j <= N; j++) {
    u[IX(i,j)] -= 0.5f * (u0[IX(i+1,j)] - u0[IX(i-1,j)]) / h;
    v[IX(i,j)] -= 0.5f * (u0[IX(i,j+1)] - u0[IX(i,j-1)]) / h;
  }
}
```

9.2.3 Inferring Third Dimension

In the Chen and Lobo solver, the height of the liquid surface calculated based off the pressure at each voxel in the solution domain. Stam's solver, however, does not keep track of a pressure value so the height must be obtained from the velocity field itself. To obtain the height at any given voxel, observe the flow of liquid in it's neighboring voxels. If the velocity field is flowing inward then the pressure must be high resulting in a higher liquid surface. Conversely, if the velocity field is flowing outward then the pressure must be low resulting in a low surface.



(Left) liquid flowing to center voxel (Right) liquid flowing away from center voxel

Source for calulating pressure field:

```
for(int i = 1; i <= N; i++) {
  for(int j = 1; j <= N; j++) {
    // Pressure in the x direction.
    p[IX(i,j)] = p[IX(i,j)] + u[IX(i - 1, j)];
    p[IX(i,j)] = p[IX(i,j)] - u[IX(i + 1, j)];

    // Pressure in the y direction.
    p[IX(i,j)] = p[IX(i,j)] + v[IX(i, j - 1)];
    p[IX(i,j)] = p[IX(i,j)] - v[IX(i, j + 1)];
  }
}
```

9.3 Visualization

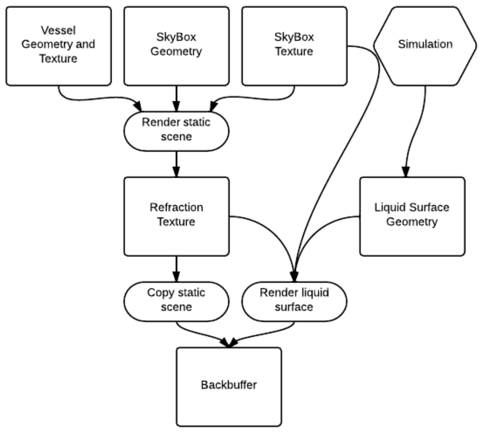
There are many visual phenomena that occur when observing liquid. Two of the most obvious phenomena being the reflection and refraction of light on its surface. In this implementation a Phong shading model is used along with a three-step rendering pipeline in an attempt to recreate the reflection and refraction of light on the liquid surface. The steps are as follows:

1. Render static objects to intermediate 'refraction' buffer

2. Copy refraction buffer to the back-buffer

3. Render the liquid surface to the back-buffer with the 'refraction' buffer as an input and while making sure to reuse the depth buffer from step 1

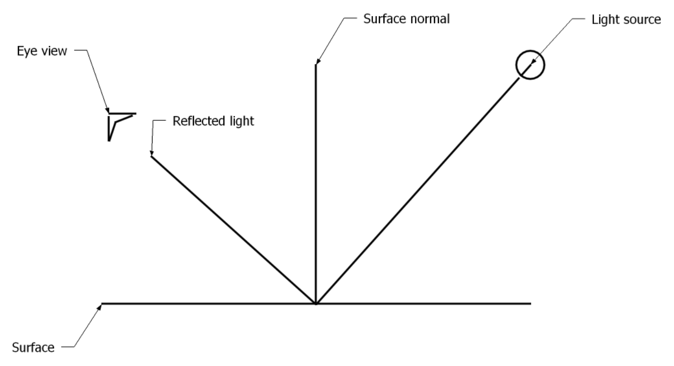
It might seem odd to first render the scene to an intermediate buffer and then copy that same buffer to the back-buffer, but DirectX 10.1 (as most graphics APIs) does not allow reading from and writing to the same render target. This setup is necessary for proper refraction of the scene behind the liquid surface.



Flow chart of the three step rendering process

9.3.1 Specular Color

The liquid surface is made up of three different effects of which will be explained in more detail here. The first effect is specular lighting. Specular lighting is observed on reflective materials (liquid for example) when light from a source hits the surface and reflects at an angle parallel to the viewer's eye.



Specular lighting diagram

The value of the specular lighting can be defined as the following:

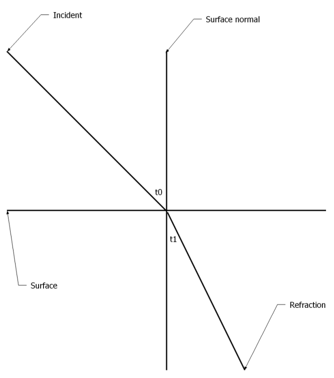
$$i = (r \cdot e)^p$$

Where r is the direction the light is reflecting off of the surface, e is the direction from the surface to the viewing eye, p is the intensity property of the surface material, and i the final specular value. Here is some example shader code that demonstrates how to calculate this value:

```
float3 projection_of_initial_on_normal =
  (dot(light_dir, input.normal) / pow(length(input.normal), 2.0f)) * input.normal;

float3 reflection =
  light_dir + (2.0f * (proj_of_initial_on_normal - light_dir));

float reflection_attenuation =
  pow(max(0.0f, dot(reflection, camera_dir)), 100.0f);
```

9.3.2 Refractive Color

The second visual effect simulated in this implementation is the refraction of light of the liquid surface. Refraction occurs when a light wave leaves one substance and enters another. For example, when a light wave from the sun hits the surface of a body of water, the light wave leaves the standard atmosphere and enters the fluid substance. Depending on the angle of incidence between the light wave and the surface of the liquid, the light wave will change directions.



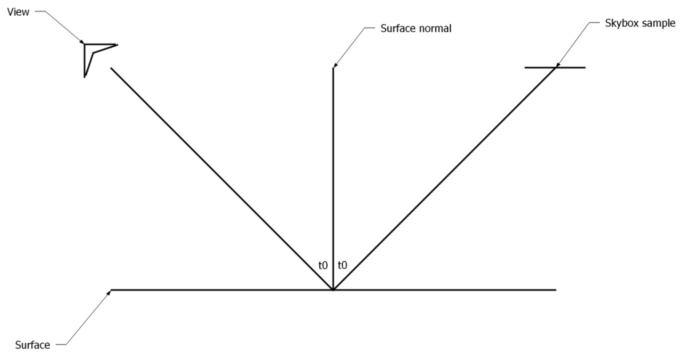
Two dimensional diagram depicting a light wave refracting as it enters another medium.

This effect alters how the viewer sees what is behind the fluid surface and is what is imitated in this simulation. A very simplistic way of imitating this is by simply offsetting the UV coordinate of the refraction buffer by the angle of the liquid surface.

```
incident = normalize(position_pixel - position_camera);
refraction = incident + surface_normal * offset;
color_refraction = SampleWithVector(refraction);
```

9.3.3 Reflective Color

The final effect implemented is the reflection of the environment off the liquid surface. For this effect simply reuse the sky-box texture cube and sample from it using a reflection vector calculated by reflecting the camera view direction off the surface normal.



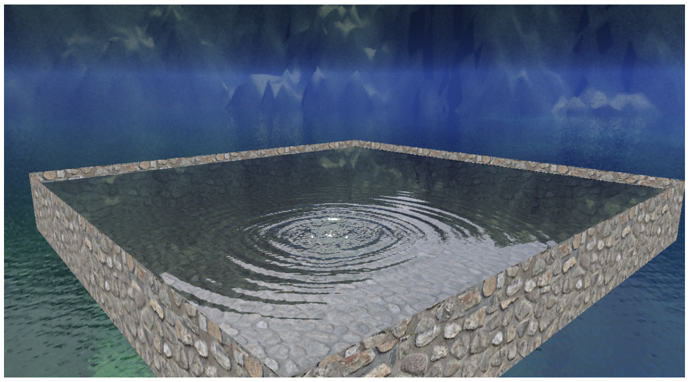
9.3.4 Color Composition

Now that the three colors that make up the fluid surface have been found (specular, refraction and reflection), combine them. To do so, linearly interpolate between the refraction and reflection color based off the viewing angle – the closer to ninety degrees the more refractive color shows up, and the closer to zero the more reflective. Finally, add specular color to the result.

```
// Reflect sample
float3 camera_to_pixel = normalize(input.world_position.xyz - camera_position);
float camera_dot_normal = dot(camera_to_pixel, normalize(input.normal));
float3 cube_sample_coords =
  camera_to_pixel - 2.0f * camera_dot_normal * input.normal;
float4 texture_cube_sample = texture_cube.Sample(sampler3, cube_sample_coords);
```

9.4 Results

The following two images are screen captures taken from grid and height-field solvers. Both solvers are running on a machine with an Intel Core i7-3610QM CPU at 2.30GHz and a NVIDIA GeForce GTX 670M graphics card.



62,500 heights simulated at over 290 frames per second.

Mostly reflective coloring
Mostly refractive coloring
Specular highlight

90,000 voxels simulated at just over 40 frames per second.

10 3D Liquid Volume

Unlike the 3D liquid surface implementations, this will simulate a free flowing volume of liquid which means calculating the velocities, pressures and densities of the liquid at all points in the volume, not just at the liquid surface. This is done by implementing a particle-based solver which behaves on the principals of the Navier-Stokes equations using an SPH interpolation scheme.

10.1 Particle-Based Solver

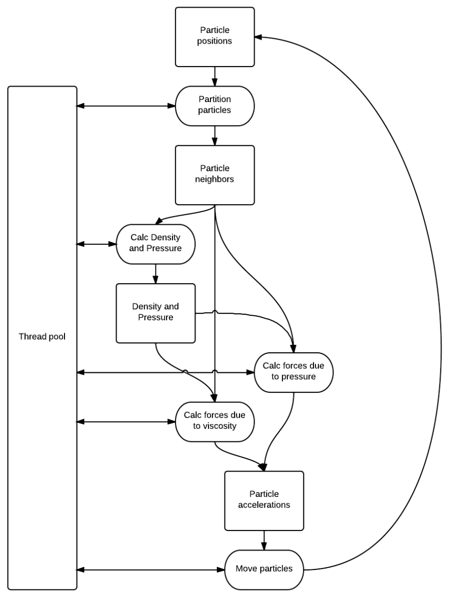
10.1.1 Overview

Implementing a ‘brute force’ or ‘naive’ Navier-Stokes solver using SPH would result in a very slow simulation. This is because there is set of n particles to simulate and each particle must be compared against each other particle in the simulation, resulting in a $O(n^2)$ solution time. Luckily there are a few things that one can take advantage of to speed up the run-time of the algorithm. This implementation has two key features that do just that.

The first feature is spatial partitioning. Spatial partitioning is very important when it comes to this type of solver. This is because the vast majority of particles don’t need to be calculated against each other. Some particles are so far away that they have no (or negligible) impact. Using this knowledge one can construct a spatial partitioning system that organizes each particle so that it only compares particles against those that are close and would have a meaningful impact.

The second feature implemented for this solver is a simple thread pool pattern. Particle-based fluid solvers are extremely parallel in that for each step updating a particle's state does not depend on the result of any other particle in the same time step. Taking advantage of this fact can yield big gains depending on how well the implementation hardware can accommodate the number of threads needed.

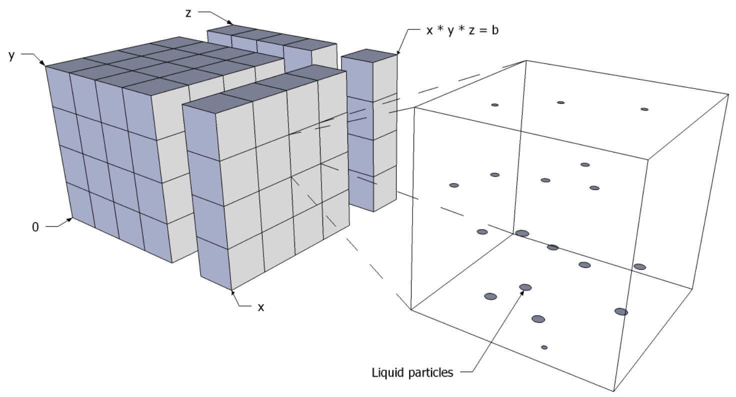
Visualization of how all this works together with the core SPH solver, see the following diagram:



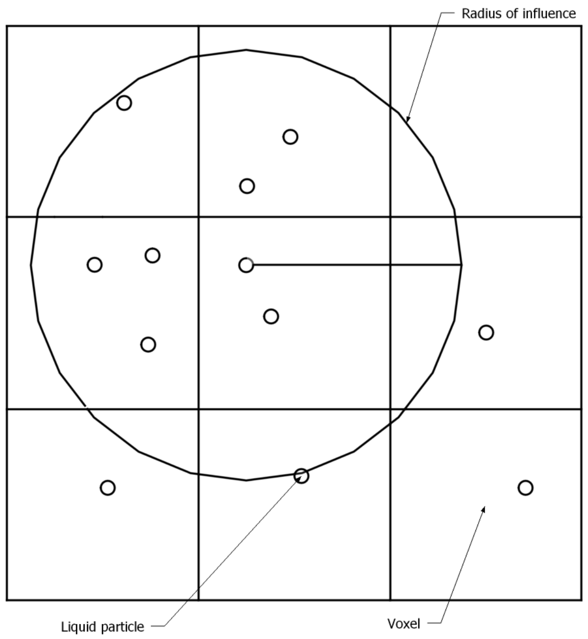
This flow chart illustrates how the core SPH solver works with the spatial partitioning and the thread pool systems.

10.1.2 Grid Partition

As described in the survey portion of this paper pertaining to SPH. Each particle has a fall off radius h which is used to determine how close a neighboring particle needs to be for it to be considered as an influence. For this solver the solution domain is split up into voxels, each sized so that the x, y, and z extents are exactly equal to a liquid particle's h value. A simple way to think of the voxelized domain is to see each voxel as a bucket, and each time step some number of particles will fall in the bounds of some number of buckets. When it comes time to collide all the particles within the simulation, one can simply collide all the particles within a single bucket. This makes finding neighboring particles very simple, see the following two diagrams:



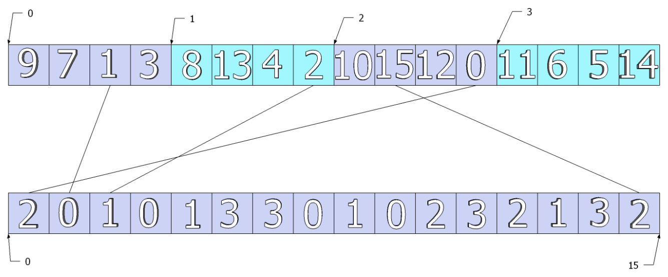
Voxelized solution domain of variable length (x) width (z) and height (y)



2D slice of solution domain. Note that the radius of influence of each particle is the unit length of each voxel. This guarantees that the only particles needed for updating a single particle's state are ones found in its voxel or voxels directly neighboring its own.

This spatial partitioning scheme is implemented using two arrays. The first array is of size

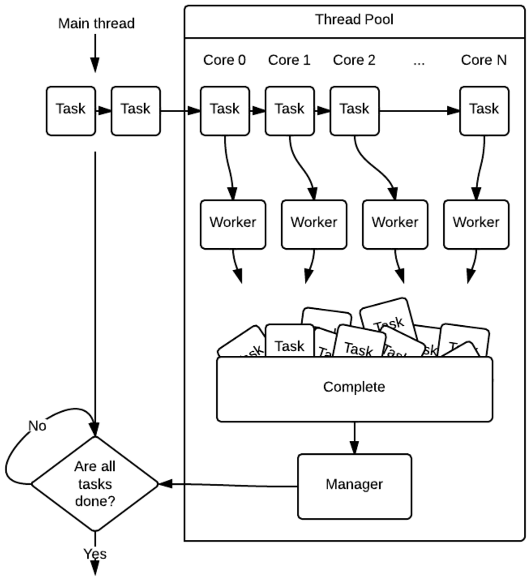
$l + m$, where l is the number of voxels, called buckets, in the solution domain and m is the maximum number of particles that could fit in each bucket. The second array is of size n, where n is the number of particles in the simulation. The idea behind these two arrays is to easily be able to take any particle and be able to find which bucket it is currently in, and, on the other hand, be able to take any bucket and quickly be able to find which particles are in side it.



Simple example of spatial partitioning data structure. The top array consists of four buckets each with four particle IDs. The second array consists of sixteen bucket IDs, where the index is the ID of the particle.

10.1.3 Thread Pool

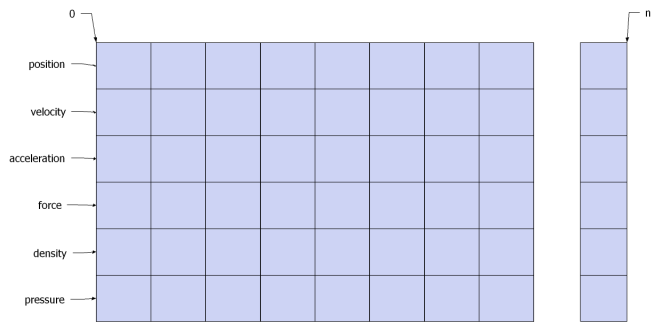
The thread pool implementation for this solver is actually very simple. There are two types of threads, one manager thread and n worker threads, where n is the number of processors on the implementation hardware. The thread pool is given a queue of tasks to complete and, when told to begin, the workers will grab tasks from the queue and carry them out. Once done with their task each worker thread will increment a count, which is monitored by the manager thread. Once the manager thread counts all tasks complete it sends out an event so that the main application knows to continue in its execution.



Tasks are sent from the main thread to the pool and blocks until the manager thread counts all tasks complete.

10.1.4 The Core SPH Solver

Data The most important data for this implementation is that of which makes up the liquid particles themselves. For this six buffers are needed, each of size n where n is the number of particles in the simulation. These buffers hold the state of each particle in the simulation for a particular time step. The state of a particle consists of four vectors: position, velocity, acceleration, force and two scalars: density and pressure. The contents of these buffers are calculated each update step of the solver.



Core data buffers for solver.

Algorithm Implementation of the core SPH solver can be broken up into two steps.

1. Calculate density and pressure of each particle based off closeness to neighboring particles
2. Calculate the forces on each particle due to pressure and viscosity from neighboring particles

Density and Pressure Step

$$\rho_S(r) = \sum_j m_j \frac{\rho_j}{\rho_j} W\left(r - r_j, h\right)$$

$$p_S(r) = G * \rho_i - R$$

As one can see from the equation above when density is plugged into the SPH equation the two density variables cancel out leaving a very simple summation of all particle position interpolations. This will calculate the density at each particle position. Once the density for a single particle is calculated, its pressure can then be obtained.

```
void DensityPressureMain(void *args) {
  unsigned particle_i = reinterpret_cast<unsigned>(args);

  // Zero density from last time
  density[particle_i] = 0.0f;

  // Which bucket is this particle in,
  // and how many other particles do we need to calculate against?
  signed bucket_index_i = bucket_index[particle_i];
  signed *bucket_i = bucket + bucket_index_i;
  unsigned bucket_count_i = bucket_count[bucket_index_i / BUCKET_CAPACITY];

  // Get position of the i'th particle
  float position_i_x = position[X(particle_i)];
  float position_i_y = position[Y(particle_i)];
  float position_i_z = position[Z(particle_i)];

  // Iterate over all neighboring particles and sum density
  for(unsigned j = 0; j < bucket_count_i; ++j) {
    signed particle_j = bucket_i[j];

    // Get position of the j'th particle
    float position_j_x = position[X(particle_j)];
    float position_j_y = position[Y(particle_j)];
    float position_j_z = position[Z(particle_j)];

    // Find difference between the two particle positions
    float difference_x = position_j_x - position_i_x;
    float difference_y = position_j_y - position_i_y;
    float difference_z = position_j_z - position_i_z;

    // Squared distance
    float squared_distance = abs(
      difference_x * difference_x +
      difference_y * difference_y +
      difference_z * difference_z
    );

    // Add in smoothed density
    if(0 <= squared_distance && squared_distance <= KERNEL_WIDTH*KERNEL_WIDTH) {
      density[particle_i] += PARTICLE_MASS * (SQUARED_POLY_CONSTANT *
        (KERNEL_WIDTH * KERNEL_WIDTH - squared_distance) *
        (KERNEL_WIDTH * KERNEL_WIDTH - squared_distance) *
        (KERNEL_WIDTH * KERNEL_WIDTH - squared_distance));
    }
  }

  // Use density to calculate pressure
  pressure[particle_i] = GAS_CONSTANT * (density[particle_i] - RESTING_DENSITY);
}
```

Forces Due to Pressure and Viscosity Step

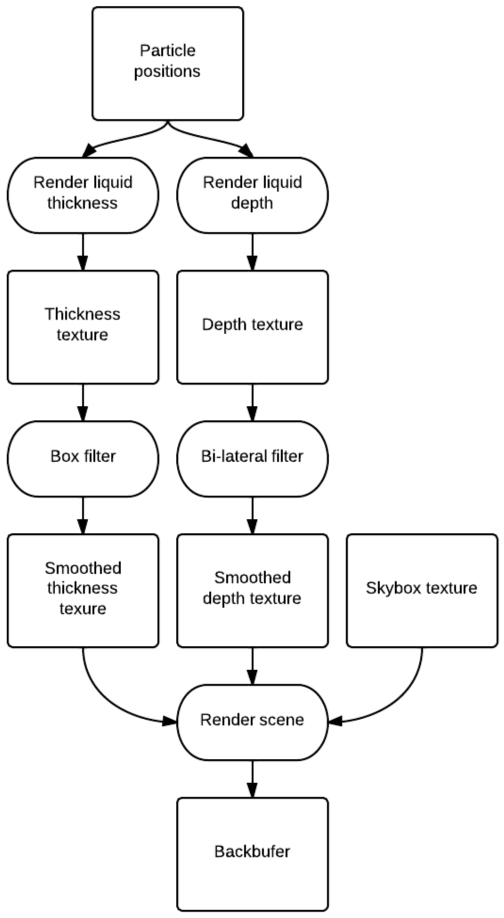
$$\nabla p_s(r) = \textstyle\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(r - r_j, h) \quad , \quad \nabla^2 v_s(r) = \textstyle\sum_j m_j \frac{v_i - v_j}{\rho_j} \nabla W(r - r_j, h)$$

```
void ForcesMain(void) {
  unsigned particle_i = reinterpret_cast<unsigned>(args);

  /* Calculate forces due to pressure and viscosity*/ {
  // If this particle doesn't have any density then skip it
  if(!density[particle_i]) return;

  // Which bucket is this particle in,
  // and how many other particles do we need to calculate against?
  signed bucket_index_i = bucket_index[particle_i];
  signed *bucket_i = bucket + bucket_index_i;
  unsigned bucket_count_i = bucket_count[bucket_index_i / BUCKET_CAPACITY];

  // Get position of the i'th particle
  float position_i_x = position[X(particle_i)];
  float position_i_y = position[Y(particle_i)];
  float position_i_z = position[Z(particle_i)];

  // Iterate over all neighboring particles and sum density
  for(unsigned j = 0; j < bucket_count_i; ++j) {
    signed particle_j = bucket_i[j];

    if(!density[particle_j]) return;

    // Get position of the j'th particle
    float position_j_x = position[X(particle_j)];
    float position_j_y = position[Y(particle_j)];
    float position_j_z = position[Z(particle_j)];

    // Find difference between the two particle positions
    float difference_x = position_j_x - position_i_x;
    float difference_y = position_j_y - position_i_y;
    float difference_z = position_j_z - position_i_z;

    // Calculate actual distance (not squared)
    float distance = sqrt(
      difference_x * difference_x +
      difference_y * difference_y +
      difference_z * difference_z
    );

    float magnitude = abs(distance);

    // Force due to pressure (GradSpiky)
    if(0 < magnitude && magnitude <= KERNEL_WIDTH) {
      float smoothing_x =
        GRAD_SPIKY_CONSTANT * (difference_x / magnitude) *
          (KERNEL_WIDTH - magnitude) * (KERNEL_WIDTH - magnitude);

      float smoothing_y =
        GRAD_SPIKY_CONSTANT * (difference_y / magnitude) *
          (KERNEL_WIDTH - magnitude) * (KERNEL_WIDTH - magnitude);

      float smoothing_z =
        GRAD_SPIKY_CONSTANT * (difference_z / magnitude) *
          (KERNEL_WIDTH - magnitude) * (KERNEL_WIDTH - magnitude);

      float pressure_over_density =
        PARTICLE_MASS * (
          (pressure[particle_j] + pressure[particle_i]) /
            (2.0f * density[particle_j])
        );

      force[X(particle_i)] += pressure_over_density * smoothing_x;
      force[Y(particle_i)] += pressure_over_density * smoothing_y;
      force[Z(particle_i)] += pressure_over_density * smoothing_z;
    }

    // Force due to viscosity (LapViscosity)
    if(0 < magnitude && magnitude <= KERNEL_WIDTH) {
      float smoothing =
        LAP_VISCOSITY_CONSTANT * (KERNEL_WIDTH - magnitude);

      force[X(particle_i)] += VISCOSITY * PARTICLE_MASS * (
        (velocity[X(particle_j)] - velocity[X(particle_i)]) /
          density[particle_j]
      ) * smoothing;

      force[Y(particle_i)] += VISCOSITY * PARTICLE_MASS * (
        (velocity[Y(particle_j)] - velocity[Y(particle_i)]) /
          density[particle_j]
      ) * smoothing;

      force[Z(particle_i)] += VISCOSITY * PARTICLE_MASS * (
        (velocity[Z(particle_j)] - velocity[Z(particle_i)]) /
          density[particle_j]
      ) * smoothing;
    }
  }
}
```

10.2 Visualization

Rendering the liquid volume is actually rather complicated with respect to more traditional scenes where there are meshes to work with instead of a mass of points. Many implementations, including the SPH implementation mentioned in this survey, use either a marching cubes method where one creates a mesh by 'voxelizing' the liquid volume and building a mesh around the outer surface, or by doing some sort of point splatting method by rendering oriented quads parallel to the liquid surface. The method for this implementation is similar to the point splatting approach but is done in screen space – a technique mentioned in the presentation given by NVIDIA at the 2010 Game Developers Conference.

The overall algorithm can be broken up into four steps each of which will be discussed in further detail, but first see the following flow chart for a general idea of how it works.



10.2.1 Thickness

This step calculates the thickness of the liquid volume at each pixel in the scene. This step is very important for deciding how the liquid surface coloring and opacity will fade off as the viewer looks through varying numbers of liquid particles. For this step the GPU is set to a color blending mode where each bit of geometry drawn is additively blended with the last. The idea is to render a bunch of billboard quads of some alpha value to the thickness texture - the more quads rendered on top of each other the higher the value stored at that position of the texture.

```
ThickVertexOutput ThickVertexMain(ThickVertexInput input) {
  ThickVertexOutput output;
    output.position_world = mul(float4(input.position_local, 1.0f), world_matrix);
  output.color = input.color;
  return output;
}

[maxvertexcount(4)]
void ThickGeometryMain(
  point ThickVertexOutput input[1],
  inout TriangleStream<ThickGeometryOut> stream
) {
  ThickGeometryOut output = (ThickGeometryOut)0;

  float3 offset_up = camera_normal * 4.0f;
  float3 offset_right = camera_cross * 4.0f;

  float3 center = input[0].position_world.xyz;
  float4 vertex0 = float4(center - offset_up - offset_right, 1.0f);
  float4 vertex1 = float4(center - offset_up + offset_right, 1.0f);
  float4 vertex2 = float4(center + offset_up - offset_right, 1.0f);
  float4 vertex3 = float4(center + offset_up + offset_right, 1.0f);

  float4 vertex0_view = mul(vertex0, view_matrix);
  float4 vertex1_view = mul(vertex1, view_matrix);
  float4 vertex2_view = mul(vertex2, view_matrix);
  float4 vertex3_view = mul(vertex3, view_matrix);

  vertex0 = mul(vertex0_view, projection_matrix);
  vertex1 = mul(vertex1_view, projection_matrix);
  vertex2 = mul(vertex2_view, projection_matrix);
  vertex3 = mul(vertex3_view, projection_matrix);

  output.position_screen = vertex0;
  output.position_view = vertex0_view;
  output.position_texture = float2(0.0f, 1.0f);
  output.color = input[0].color;
  stream.Append(output);

  output.position_screen = vertex1;
  output.position_view = vertex1_view;
  output.position_texture = float2(1.0f, 1.0f);
  output.color = input[0].color;
  stream.Append(output);

  output.position_screen = vertex2;
  output.position_view = vertex2_view;
  output.position_texture = float2(0.0f, 0.0f);
  output.color = input[0].color;
  stream.Append(output);

  output.position_screen = vertex3;
  output.position_view = vertex3_view;
  output.position_texture = float2(1.0f, 0.0f);
  output.color = input[0].color;
  stream.Append(output);

  stream.RestartStrip();
}

ThickPixelOutput ThickPixelMain(ThickGeometryOut input) {
  ThickPixelOutput output;

  float3 normal =
    float3(input.position_texture * 2.0f - 1.0f, 0.0f) * float3(1.0f, -1.0f, 1.0f);
  float squared_radius = dot(normal.xy, normal.xy);
  if(squared_radius > 1.0f) discard;

  output.color = float4(0.50f, 0.0f, 0.0f, 1.0f);
  return output;
}

technique10 Thick {
  pass P0 {
    SetRasterizerState(DisableCull);
    SetDepthStencilState(DisableDepth, 0);
    SetBlendState(ColorBlend, float4(0.0f, 0.0f, 0.0f, 0.0f), 0xfffffff);
    SetVertexShader(CompileShader(vs_4_0, ThickVertexMain()));
    SetGeometryShader(CompileShader(gs_4_0, ThickGeometryMain()));
    SetPixelShader(CompileShader(ps_4_0, ThickPixelMain()));
  }
}
```

10.2.2 Depth

This step calculates the screen space depth of the liquid volume at each pixel. This is achieved by first generating billboard quads from the particle positions. The idea is to think of these quads as spheres and render their depth out accordingly. Using the UV coordinates of each quad and the fact that spheres are symmetrical, this can be achieved. The following is the pixel shader code for this step.

```
QuadPixelOutput QuadPixelMain(QuadGeometryOut input) {
  QuadPixelOutput output;

  // Calculate normal from uv coordinates.
  float3 normal = float3(input.position_texture * 2.0f - 1.0f, 0.0f) *
    float3(1.0f, -1.0f, 1.0f);
  float squared_radius = dot(normal.xy, normal.xy);

  // Throw out any pixels outside of the sphere radius
  if(squared_radius > 1.0f) discard;
  normal.z = -sqrt(1.0f - squared_radius);

  // Use surface normal to calculate pixel depth
  float4 position_view = float4(input.position_view.xyz + normal * 4.0f, 1.0f);
  float4 position_projection = float4(mul(position_view, projection_matrix));
  float depth = position_projection.z / position_projection.w;

  output.color = float4(depth, 0.0f, 0.0f, 0.5f);
  return output;
}
```

10.2.3 Smoothing

Smoothing is perhaps the most important part of this surface rendering technique. Without any smoothing the liquid volume would simply look like a bunch of spheres. For this implementation there are to stages of smoothing. The first smoothing stage is needed for the thickness texture. For this a simple box filter is used. The second stage, however, is a little more complicated, because it involves smoothing positional depth values. For the depth texture a bilateral filter is used.

The Box Filter The box filter is perhaps one of the simplest filters one can implement. The box filter is a very fast and inexpensive filter, but what it gains in speed and simplicity it lacks in quality. The lack of quality is acceptable in this case, however, because smoothing the thickness texture beyond fairly low quality thresholds yields virtually unnoticeable gains.

The algorithm is very simple and involves deciding on a kernel width k and summing all texel values within k and then dividing by the number of texels sampled.

```
SmoothThickPixelOutput SmoothThickPixelMainX(SmoothThickVertexOut input) {
  SmoothThickPixelOutput output;

  float thickness = texture0.Sample(pointClampSampler, input.position_texture).r;
  float texel_width = 1.0f / screen_width * 0.5f;
  float texel_height = 1.0f / screen_height * 0.5f;

  float kernel_width = ThickSIZE / 2.0f;

  for(int i = -kernel_width; i < kernel_width; ++i) {
    for(int j = -kernel_width; j < kernel_width; ++j) {
      float2 position_texture_neighbor =
        input.position_texture + float2(i * texel_width, j * texel_height);
      float thickness_neighbor =
        texture0.Sample(pointClampSampler, position_texture_neighbor).r;
      if(thickness_neighbor) {
        thickness += thickness_neighbor;
      }
    }
  }

  thickness /= (ThickSIZE * ThickSIZE);

  output.color = float4(thickness, 0.0f, 0.0f, 1.0f);
  return output;
}
```

The Bilateral Filter For the depth texture, apply a bilateral filter. A bilateral filter is used because it has the quality of a Gaussian filter, which weighs samples based on how close they are to the center pixel, but also preserves edges. The problem with simply applying a screen space Gaussian filter to the depth texture is that particles that are close to the camera but far away from any other particles in the liquid will still get blended with the ones behind them resulting in an unrealistic liquid surface.

This implementation is a separable bilateral filter, which means only $k+k$ texels per pixel need to be sampled instead of the normal $k * k$. To accomplish this, break the smoothing into two render passes – one for the x direction and one for the y. The following is shader code for a single pass in the x direction.

```
SmoothPixelOutput SmoothPixelMainX(SmoothVertexOut input) {
  SmoothPixelOutput output;

  float depth_center = texture0.Sample(pointClampSampler, input.position_texture).r;
  float depth = 0.0f;
  float texel_width = 1.0f / screen_width;
  float texel_height = 1.0f / screen_height;
  float sum_count = 0.0f;

  for(int i = -KERNEL_WIDTH2; i < KERNEL_WIDTH2; ++i) {
    float2 position_texture_neighbor =
      input.position_texture + float2(texel_width * i, 0.0f);

    float depth_neighbor =
      texture0.Sample(pointClampSampler, position_texture_neighbor).r;

    if(depth_neighbor) {
      float spatial_contribution =
        1.0f - max(min(abs(depth_center - depth_neighbor) * 1000.0f, 1.0f), 0.0f);
      float depth_contribution = FlatKernel2[i + KERNEL_WIDTH2];
      depth += depth_neighbor * spatial_contribution * depth_contribution;
      sum_count += spatial_contribution * depth_contribution;
    }
  }

  if(sum_count) {
    depth /= sum_count;
  }

  output.color = float4(max((min(depth, 1.0f)), 0.0f), 0.0f, 0.0f, 1.0f);
  return output;
}
```

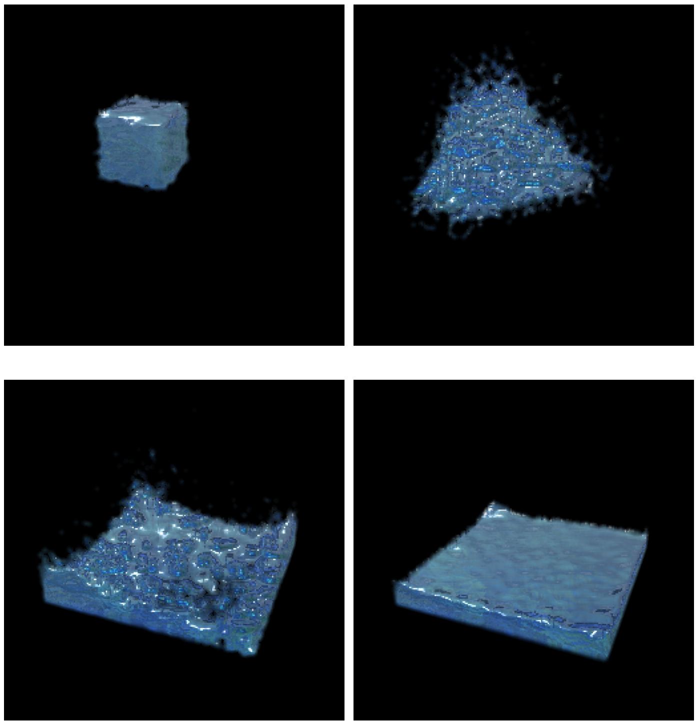
10.2.4 Final Composition

Once the thickness and the depth of the liquid volume at every pixel is known the scene can finally be rendered. The final composition consists of calculating three vectors - the liquid surface normal, reflection vector, and refraction vector. The trickiest of all three being the surface normal. This is done by calculating the change in depth at each pixel in comparison to its neighboring pixels. Using this technique the surface normal can be approximated. Then the surface normal along with the viewing angle can be used to calculate the reflection and refraction vectors used to index into the sky-box.

```
ScreenPixelOutput ScreenPixelMain(ScreenVertexOut input) {
  ScreenPixelOutput output;

  float2 half_screen = float2(screen_width, screen_height) * 0.5f;

  float position_projection_z =
    texture0.Sample(pointClampSampler, input.position_texture).r;

  if(position_projection_z == 0.0f) discard;

  // Reconstruct projection position
  float2 position_projection_xy =
    ((input.position_screen.xy - half_screen) / half_screen) * float2(1.0f, -1.0f);

  float position_projection_w =
    -camera_near / (
      -1.0f +
      position_projection_z *
      ((camera_far - camera_near) / camera_far)
    );

  float4 position_projection = float4(
    position_projection_xy,
    position_projection_z,
    1.0f
  ) * position_projection_w;

  // Transform projection to world
  float4 position_view = mul(position_projection, projection_matrix_inv);
  float4 position_world = mul(position_view, view_matrix_inv);

  // Calculate surface normal from change in world position
  float3 surface_normal =
    normalize(cross(ddx(position_world.xyz), ddy(position_world.xyz)));

  // Reflection skybox sample
  float3 camera_to_pixel = normalize(position_world.xyz - camera_position);
  float camera_dot_normal = dot(-camera_to_pixel, surface_normal);
  float3 cube_sample_coords =
    camera_to_pixel - 2.0f * -camera_dot_normal * surface_normal;
  float4 color_reflection =
    texture_cube.Sample(linearWrapSampler, cube_sample_coords);

  // Refraction skybox sample
  camera_to_pixel =
    normalize(position_world.xyz - camera_position) + surface_normal * 0.05f;
  float4 color_refraction = texture_cube.Sample(linearWrapSampler, camera_to_pixel);

  // Interpolate between reflection and refraction sample.
  float4 color_refref =
    camera_dot_normal * color_refraction +
    (1.0f - camera_dot_normal) * color_reflection;

  // Specular lighting
  float3 projection_of_initial_on_normal =
    (dot(light_direction, surface_normal) /
    pow(length(surface_normal), 2.0f)) * surface_normal;

  float3 reflection =
    light_direction + (2.0f * (projection_of_initial_on_normal - light_direction));
  float specular_value = pow(max(0.0f, dot(reflection, camera_direction)), 20.0f);

  // Sample thickness of liquid for alpha blending.
  float thickness = texture1.Sample(linearWrapSampler, input.position_texture).r;

  output.color = float4(
    color_refref.rgb + float3(1.0f, 1.0f, 1.0f) * specular_value,
    thickness
  );
  return output;
}
```

10.3 Results

The images below are screen captures of the particle solver simulating 5000 particles at just over 40 frames per second. The simulation was run on a machine with an Intel Core i7-3610QM CPU at 2.30GHz and a NVIDIA GeForce GTX 670M graphics card.



One might think that maximizing the number of threads working on all possible computations that could be done in parallel would result in the best results. However, the following data found shows that not to be the case. There are many factors when it comes to multi-threading an application, one big one being the OS' implementation and management of threads. The following charts are of data collected while simulating at different particle counts, different thread counts, and with different stages of the code using the thread pool.

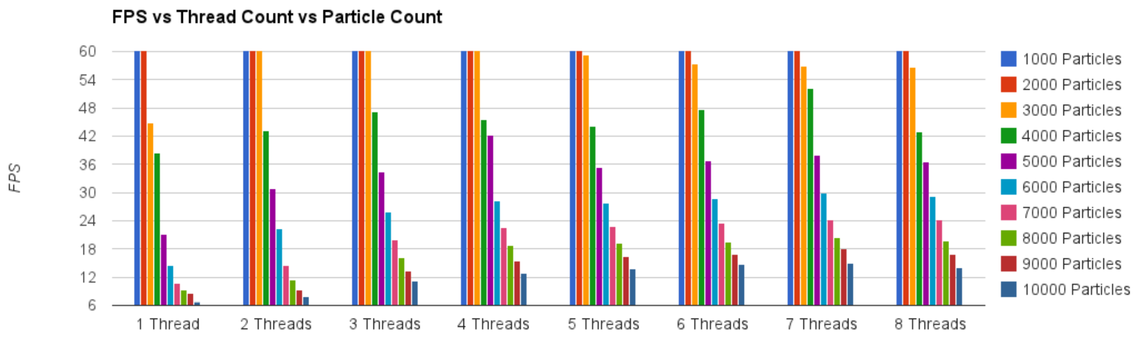


Figure 14

Looking at Figure 14 shows that as the number of particles go up the frames per second go down. This makes sense, the more particles the more calculations. However, the more threads that are introduced into the simulation the slower the frame rate drops due to particle count. So the question becomes, how many threads should be used in a simulation in order to maximize both particles and frame rate? Surprisingly, the answer is not as many as possible.

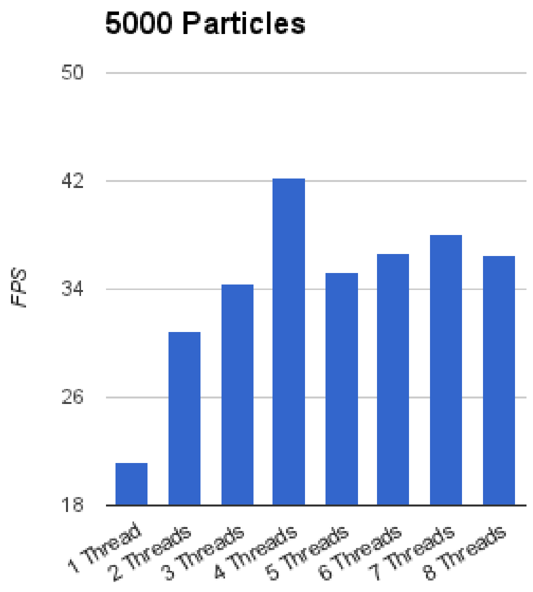


Figure 15

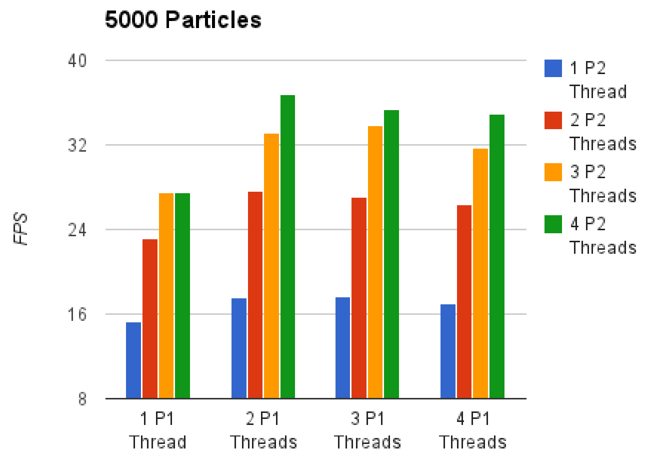


Figure 16

For this implementation, frame rates dropping below 30 are considered to be out of the realm of real-time. Using this as the absolute bottom line it becomes easy to see the best choice. The largest number of particles that gets over 30 fps is 5000, and that is when using 4 threads. This becomes more obvious when isolating the 5000 particle bars into their own graph as in Figure 15.

Seeing this data brought the question of whether or not there were other non-obvious configurations that would lead to better frame rates. Up until now the thread count had been modified for the solver as a whole, but what if different steps used different thread counts? Figure 16 shows statistics on just that (P1: Density and pressure step, P2: Forces due to viscosity and pressure step). The figure shows this does have an impact on the frame rate, the most optimal configuration being two threads for P1 and four threads for P2. As interesting as that is, the point is moot, due to the fact that the highest frame rate achieved was still slower than the highest frame rate in Figure 15.

Part III

Conclusions and Future Work

Hopefully this paper has clearly illustrated the complications and fascinations inherent in fluid simulation. There are challenges but when overcome the visuals can be very rewarding. Even under real-time constraints physically-based fluid simulation can be achieved, as shown in the implementations of this thesis.

The following section will cover some future work that can be done to further improve the implementations described in this paper.

General GPU Programming Due to the advancement of programmable graphics pipelines it's becoming less and less abnormal to offload general (non-graphics) computations from the CPU to the GPU. Gains have already been shown in the particle-based solver by introducing a simple thread pool allowing for c computations to be calculated at the same time where c is the number of cores on the implementation hardware. Imagine the performance gains when being able to calculate virtually p calculations at the same time where p is the number of pixel shaders being executed. With general GPU programming this can be achieved.

All three implementations shown in the paper could, in fact, benefit from significant performance boosts if implemented on the GPU. For more information on this approach there are several papers that should be investigated for height-fields [30], grid [39] and particle [12, 32] implementations.

Caustics Just as important as the physics that drive the simulation, the visual representation of the fluid can be the thing that makes or breaks the illusion. This is why it is very important to apply the proper graphics techniques to the simulation. Most importantly ones that trick the brain into thinking its viewing the animation of real fluid. One very important effect when it comes to simulating liquids is the visualization of caustics.

Caustics are a lighting effect that can be observed when photons from a light source gather on a surface as an effect from passing through some substance and reflecting or refracting. When talking about liquids the most obvious case of this is when light passes through the liquid surface causing photons of light to refract in patterns on surfaces behind the liquid volume. See Figure 17 from [34] for an example of this effect. This effect is also shown in [2] and [43].

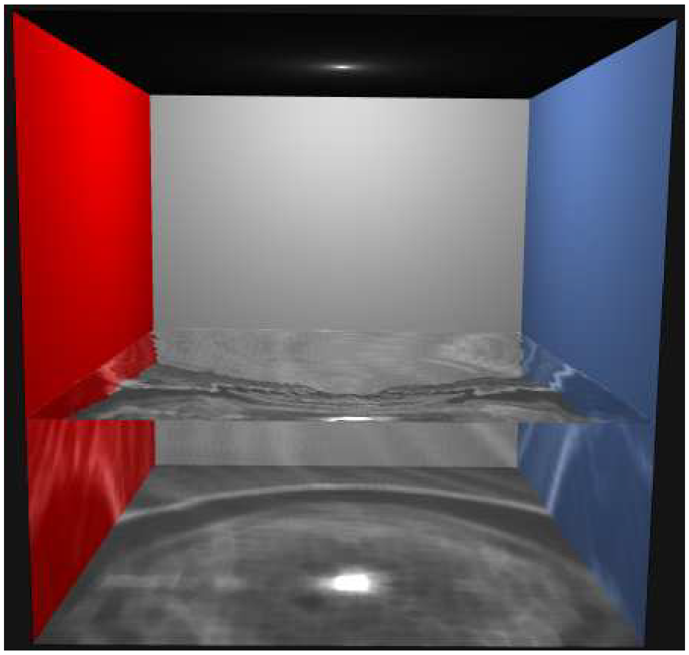


Figure 17: Light from the ceiling refracts when entering the liquid and gathers on the floor

Rigid Body Interactions All fluid simulations are only as realistic as the interactions between the fluid itself and the scene in which the fluid is represented. This makes it very important for interactions with external objects, such as rigid bodies, to behave properly. Realistic external interactions are a natural next step in the implementation of a real-time liquid solver like the implementations shown in this thesis. A good example of this is seen in [13] (also see Figure 18) where Hirada et al. voxelize rigid bodies into particles so that physical interactions seamlessly integrate with a particle-based liquid solver.

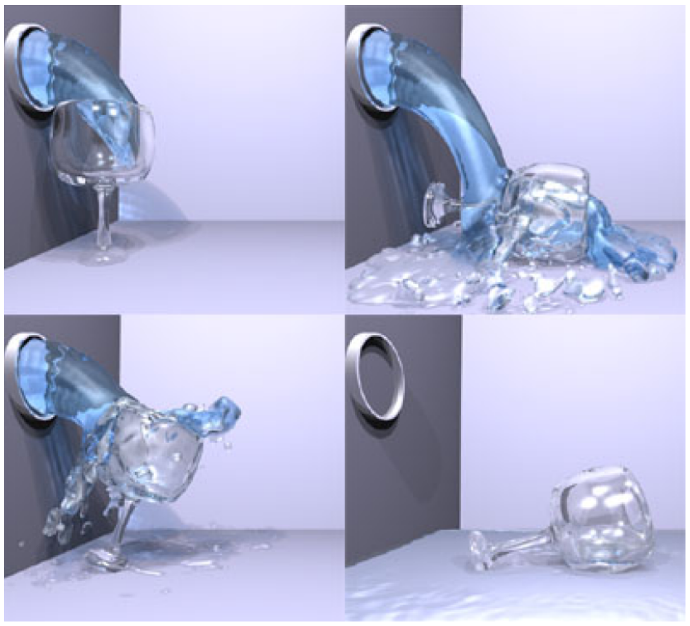


Figure 18: Physically-based liquid interacting with external rigid-bodies

References

[1] Allstar network. aeronautics - fluid dynamics - level3 - flow equations. http://www.allstar.fiu.edu/aero/Flow2.htm, January 29th, 2007.

[2] Barczak, Josh. Olano, Marc. Interactive Shadowed Caustics Using Hierarchical Light Volumes. University of Maryland Baltimore County.

[3] Chen, Jim X. and Lobo, Niels Da Vitoria. Toward Interactive-Rate Simulation of Fluids with Moving Obstacles Using Navier-Stokes Equations. Graphical Models and Image Processing Vol. 57, No. 2, March, pp. 107-116, 1995

[4] Desbrun, Mathieu. Gascuel, Marie-Paule. Smoothed Particles: A new paradigm for animating highly deformable bodies. Grenoble codex 09.

[5] Fastest Fourier Transform in the West. http://www.fftw.org, March 24th, 1997.

[6] Gauss, Carl Friedrich. Theoria attractionis corporum sphaeroidicorum ellipticorum homogeneorum methodo nova tractata. 1813.

[7] Green, Simon. Particle-based Fluid Simulation. NVIDIA 2008.

[8] Green, Simon. Screen Space Fluid Rendering for Games. Game Developers Conference, 2010.

[9] Green, George. An Essay on the Application of Mathematical Analysis to the Theories of Electricity and Magnetism, pp 10-12. 1838.

[10] Greene, Ned. Creating Raster Omnimax Images from Multiple Perspective Views Using the EllipticalWeighted Average Filter. IEEE Computer Graphics & Applications, June 1986.

[11] Grossman, J.P. Dally, W. Point Sample Rendering. Massachusetts Institute of Technology, 1999.

[12] Harada, Takahiro. Koshizuka, Seiichi. Kwaguchi, Yoichiro. Smoothed Particle Hydrodynamics on GPUs. The Visual Computer manuscript.

[13] Harada, Takahiro. Tanaka, Masayuki. Koshizuka, Seiichi. Kwaguchi, Yoichiro. Realtime Coupling of Fluids and Rigid Bodies. APCOM 07 in conjunction with EPMESC XI, December 3-6, 2007, Kyoto, JAPAN.

[14] Hoetzlein, Rama. Hollerer, Tobias. Interactive Water Streams with Sphere Scan Conversion. Association for Computing Machinery, Inc. 2009.

[15] Jensen, Lasse Staff. Golias, Robert. Deep-Water Animation and Rendering. Funcom Oslo AS.

[16] Kass, Michael and Miller, Gavin. Rapid, Stable Fluid Dynamics for Computer Graphics. Computer Graphics, Volume 24, Number 4, August 1990.

[17] Kealager, Micky. Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics. Department of Computer Science, University of Copenhagen, 2006.

[18] Kim, Janghee. Cha, Deukhyun. Chang, Byungjoon. Koo, Bonki. Ihm, Insung. Practial Animation of Turbulent Splashing Water. SIGRAPH 2006.

[19] Kuhbacher, Christian. Shallow Water Derivation and Applications. Technische Universitat Dortmund, 2009.

[20] Lagrange, Joseph-Luis, Nouvelles recherches sur la nature et la propagation du son, 1762.

[21] Lorensen, William E. Marching Cubes: A High Resolution Surface Construction Algorithm. ACM Press, 1987.

[22] L. B. Lucy. A numerical approach to the testing of the sion hypothesis. The Astronomical Journal, 82:10131024, 1977.

[23] Maes, Marcelo M. Fujimoto, Tadahiro. Chiba, Norishige. Efficient Animation of Water Flow on Irregular Terrains. Association for Computing Machinery, Inc. 2006.

[24] Martin, T.J. Pearce, F.R. Thomas P.A. An Owner's Guide to Smoothd Particle Hydrodynamics. Astronomy Centre, Susser University, 1993.

[25] McLean, William. Poisson Solvers. April 21, 2004.

[26] Monaghan, J. J. Smoothed Particle Hydrodynamics. Annu. Rev. Astron. Astrophys. 1992. 30:543-74, 1992.

[27] Muller, Matthias. Charypar, David. Gross, Markus. Particle-Based Fluid Simulation for Interactive Applications. Eurographics/SIGGRAPH Symposium on Computer Animation. Pages 154-159. Year 2003.

[28] Muller-Fischer, Matthias. Fast Water Simulation for Games Using Height Fields. Game Developers Conference, 2008.

[29] Muller-Fischer, Matthias. Real Time Fluids in Games. PhysX by ageia, SIGGRAPH 2006.

[30] Noe, Karsten O. Implementing Rapid, Stable Fluid Dynamics on the GPU. Fall 2004.

[31] Perlin, Ken. Hoffert, Eric M. Hypertexture. Computer Graphics, Volume 23, Number 3, July 1989.

[32] Oat, Christopher. Barczak, Joshua. Shopf, Jeremy. Efficient Spatial Binning on the GPU. AMD Technical Report. February 2009.

[33] Perlin, Ken. An Image Synthesizer. SIGGRAPH, Volume 19, Number 3, 1985.

[34] Shah, Musawir. Pattanaik, Sumanta. Caustics Mapping: An Image-space Technique for Real-time Caustics. School of Engineering and Computer Science, University of Central Florida, CS TR 50-07.

[35] Stam, Jos. A Simple Fluid Solver Based on the FFT. Alias | wavefront

[36] Stam, Jos. Real-Time Fluid Dynamics for Games. Proceedings of the Game Developer Conference, March 2003.

[37] Stam, Jos. Stable Fluids. SIGGRAPH, Pages 121-128, year 1999.

[38] Tessendorf, Jerry. Simulating Ocean Water. SIGGRAPH 2004.

[39] Thibault, Julien C. Senocak, Inanc. CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Deskctop Platforms for Incompressible Flows. Boise State University, Boise, Idaho, 83725.

[40] Trevethan, Brian. Physically-Based Fluid Simulation for Computer Graphics. DigiPen Institute of Technology, 2004.

[41] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In Proceedings of the 14th annual conference on Computer graphics and interactive techniques, pages 163169. ACM Press, 1987.

[42] Wang, Huamin. Miller, Gavin. Turk, Greg. Solving General Shallow Wave Equations on Surfaces. SIGGRAPH 2007.

[43] Yu, Xuan. Li, Feng. Yu, Jingyi. Department of Computer and Information Sciences University of Delaware Newark.

[44] Zhu, Yongning. Bridson, Robert. Animating Sand as a Fluid. Association for Computing Machinery, Inc. 2005.

[45] Zwicker, Matthias. Pfister, Hanspeter. Baar, Jeroen van. Gross, Markus. Surface Splatting. SIGGRAPH 2001.

[46] Zwicker, Matthias. Pster, Hanspeter. Baar, Jeroen van. Gross, Markus. Surface splatting. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pages 371378. ACM Press, 2001.