# Physical Simulation of Fire and Smoke Masters Thesis

David Minor

N.C.C.A Bournemouth University

September 10, 2007

# Contents

**1 Introduction** **1**

**2 Previous Work** **2**

**3 Modeling The Dynamics of a Flame** **3**
3.1 The Euler Equations . . . . . 3
3.2 Solving the Euler Equations . . . . . 4
3.3 The Flame Model . . . . . 10

**4 Implementation** **15**
4.1 The Pressure Solver . . . . . 16
4.2 Solid Boundaries . . . . . 17
4.3 Optimization . . . . . 20
4.4 Buoyancy and User Defined Curves . . . . . 21

**5 Rendering Smoke and Flames** **21**
5.1 Computing Self Shadowing . . . . . 22
5.2 Rendering Light from the Flames . . . . . 23

**6 Results** **24**

**7 Current Problems** **29**

**8 Conclusion and Future Work** **29**

# 1 Introduction

Fire has fascinated us ever since early humans had the idea of rubbing two sticks together. It is a dangerous, otherworldly phenomenon, which perhaps our brains, billions of years in the making, are unused to dealing with in daily life, and therefore pay extra attention to. Fire symbolizes rage, conflict and destruction, and stories of rage, conflict and destruction have always interested people. Nowadays, we tell these stories with sounds and moving pictures. Many movies require the depiction of fire, particularly action movies, and the danger and expense of working with fire mean a director often needs to use tricks and sleight of hand to make the viewer believe something happened which would be too dangerous or expensive to actually shoot.

Like with many things, there are certain shots which would be difficult or impossible to create by shooting real fire - the director may want a slow motion shot of a petrol flame, following a tiny piece of debris as it tumbles away from the conflagration at high speed. They may want an impossibly fast flight out of a mine shaft just as a coal dust explosion gathers momentum, with flames licking round the edge of the visual field. Along with this, real fire can be difficult and expensive to work with for safety reasons, and a director may want an alternative means of creating a shot. Because of this, software which renders convincing images and animations of fire can be a useful tool. This project focuses on the development of such a tool.

## 2 Previous Work

There are two approaches one can take when creating CG fire. When creating a highly stylized shot, for a cartoon, or perhaps a magic spell in a live action movie, it may be preferable to eschew physical accuracy and create a system designed with artistic control in mind. Foster and Lamorlette take this approach in [3], where they explicitly model the structure of a flame by using control curves carried through a wind field. Portions of these curves stochastically split off from the main control curve to create the puffs and plumes seen in a natural flame, and the flame is rendered volumetrically using noise fields. This system was used to create the fire effects in the animated film Shrek.

If more realism and less artistic control is required, for example, if a fire is to take place as a background element of a live action shot, it may be better to create an animation based on the actual physics of fire. This can be achieved using fluid simulation software, something which has become commonplace in 3D animation lately, and was used to create the oil fire in the lobby in the film Poseidon (2006), using the fluid simulation package FlowLine. This is the approach taken by this project.

The simulation of real world materials like solids, liquids and gases, which typically requires large amounts of memory and computing time, has become ever more common in visual effects, due to the increasing power of modern computers. This kind of simulation normally involves taking the equations governing the behavior of the material and using a computer to calculate its motion based on equations. The engineering community has worked on this kind of problem for decades, and has developed a great deal of expertise, and many very accurate techniques, but until recently these techniques have required top end supercomputers, making them unsuitable for creating animations.

Fire is a gaseous phenomenon, technically a fluid, and the field of numerical fluid simulation is known as Computational Fluid Dynamics (CFD). CFD is largely underpinned by the so called Navier Stokes equations, which describe, on a very basic level, how the motion of a fluid changes in time. The first published attempt to use these equations in computer graphics was a paper by Foster and Metaxas[26], who used earlier work by Harlow and Welch[29] to create animations of liquids. Since then, a large amount of work has been done in the field of fluid simulation for graphics, producing software capable of animating liquids of varying viscosity, viscoelastic fluids like chewing gum, sand, gases, fire, and the interactions between these different media, as well as two way interactions with rigid bodies.

Many commercially available software packages for the visual simulation of fluids have come from this work. Examples of packages which can handle fire are Maya's inbuilt fluid simulator, the software package FlowLine from German VFX house ScanLine, and the 3ds max plugin FumeFX.

# 3 Modeling The Dynamics of a Flame

Fire is a gaseous phenomenon, consisting of turbulent, incandescent gas and soot, driven by a chemical reaction between an oxidizer and a fuel[7]. A flame can be satisfactorily modeled as a very fast reaction in a gaseous state, with a region of vaporised fuel, separated from a region of hot exhaust gases by a thin reaction surface. This reaction surface spreads into the fuel region according to certain rules, causing a rapid expansion of the gas it crosses. To correctly model the behavior of a flame, one must model the dynamics of the gas on either side of this surface, the motion of the surface itself, and the expansion caused by the reaction. This requires the modeling of fluids.

Fluids can be simulated in a number of ways, each with their advantages and disadvantages. Broadly speaking, fluid models can be divided into two categories: Eulerian and Lagrangian models[17]. The two approaches differ, essentially, in that Eulerian models store fluid properties like pressure and fluid velocity on a structure which is fixed in space, and track the evolution of those properties in time, and Lagrangian models use structures which move with the fluid, like a moving tetrahedral mesh or a collection of particles. A third approach, the Arbitrary Lagrangian Eulerian (ALE) approach[18], lies between the two. Currently, the most widely used technique in computer graphics uses an Eulerian cubic grid, and is based on the so called "Stable Fluids" algorithm[23]. This algorithm is explicitly based on the partial differential equations governing fluid dynamics, and solves them numerically. An implementation of this algorithm forms the core of this project.

## 3.1 The Euler Equations

The simplest equations describing fluid dynamics are known as the Euler equations, which are a special case of the Navier Stokes equations. They describe fluids with no internal forces other than pressure, which acts radially on pairs of particles in the fluid and conserves energy. The Euler equations are a good model for the behavior of fire, as other internal forces like viscosity are usually very small under the conditions being considered, and also tend to damp the turbulent effects which make fire and smoke visually interesting. The Euler equations describe the evolution of the fluid velocity field $\underline{u}$ in time, in terms of the pressure $p$, density $\rho$ and external forces acting on the fluid per unit volume, $\underline{f}$. The equations are summarized in equation 1:

$$\frac{\partial \underline{u}}{\partial t} + (\underline{u}.\nabla)\underline{u} + \frac{\nabla p - \underline{f}}{\rho} = 0 \tag{1}$$

In terms of individual components of $\underline{u} = (u, v, w)$, equation 1 can be written:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} + \frac{1}{\rho}\left(\frac{\partial p}{\partial x} - f_x\right) = 0$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} + \frac{1}{\rho}\left(\frac{\partial p}{\partial x} - f_y\right) = 0$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} + \frac{1}{\rho}\left(\frac{\partial p}{\partial x} - f_z\right) = 0$$

Most fluid phenomena we see can be modeled adequately as incompressible flows, meaning that any given fluid element preserves its volume as it is carried along with the flow. Compressible effects only become important under extreme conditions, for example an explosion, the flight of a bullet or the mechanics of a supersonic jet engine (and of course sound, which is completely invisible), and can therefore be ignored for most applications in computer graphics. This is convenient, as compressibility leads to complex phenomena like shock waves, which require sophisticated, computationally expensive algorithms.

The incompressibility condition is equivalent to saying that the total fluid flowing out of any volume is equal to the fluid flowing into that volume. This can be stated as follows:

$$\oint_s \underline{u}.\underline{\hat{n}}dA = 0$$

Where $S$ is any arbitrary closed surface, $\underline{\hat{n}}$ is a unit outward facing normal to the surface and $dA$ is an area element. The infinitesimal quantity $\underline{u}.\underline{\hat{n}}dA$ is the volume of fluid flowing out of an infinitesimally small patch of the surface $S$ per unit time - summing this over the surface gives the total fluid flowing out of the surface, which must be equal to zero for no fluid compression to occur. If $\underline{u}$ is sufficiently continuous, then by the Divergence Theorem[27] this is equivalent to:

$$\int_V \nabla.\underline{u}dV = 0$$

Where, in Cartesian coordinates, $\nabla.\underline{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$, $V$ is the volume enclosed by the surface $S$, and $dV$ is a volume element. Since the volume is arbitrary, and therefore the equation must hold for all volumes, it follows that the integrand must be zero everywhere, and we arrive at the incompressibility equation:

$$\nabla.\underline{u} = 0 \tag{2}$$

## 3.2 Solving the Euler Equations

The Euler equations can lead to very complex behavior, which along with the very complicated and unpredictable driving forces the fluid is subjected to, make them impossible to solve symbolically. We instead search for approximate numerical solutions to the equations, by "discretizing" them: we split the spatial domain into cubic cells, and the time domain into discrete steps. There are many ways of doing this, but we settle for a so called "finite difference" approach, where quantities are defined at regularly spaced points, and derivatives are estimated by taking differences.

The simplest, most common pattern used for defining quantities in finite difference simulations is on a cubic grid. We refer to quantities on such a grid using three integers, denoting the Cartesian coordinates of the cell on the grid, which we write as subscripts separated by commas after the quantity. For example, $c_{i,j,k}$ refers to the value of the quantity c stored at the grid point i cells along the x axis, j cells along the y axis and k cells along the z axis. We store some quantities at points between grid cells, for example at the center of one of the cell's faces, which we will represent using fractional indices. For example, the quantity $q_{i+\frac{1}{2},j,k}$ is located on the positive x face of the cell whose coordinates are i,j,k.

Foster and Metaxas[26] did the first published work on using the full 3D equations of fluid dynamics for computer animation, and essentially did so by writing down the equations governing the fluid, and replacing all derivatives with differences, for example:

$$\frac{\partial \underline{u}}{\partial x} \simeq \frac{\underline{u}_{i+1,j,k} - \underline{u}_{i,j,k}}{\triangle x}$$

$$\frac{\partial \underline{u}}{\partial t} \simeq \frac{\underline{u}_{t+1} - \underline{u}_{t}}{\triangle t}$$

Where $\triangle t$ is our time sampling interval, $\triangle x$ is the width of one cell on our spatial sampling grid, and the subscripts $i$, $j$, $k$ and $t$ refer to integer locations on the sampling grid. They then used the resulting equations to estimate $\frac{\partial \underline{u}}{\partial t}$ at a given instant in time, minus the contribution due to the pressure term, multiplying that by the time step and adding the resulting quantity to the velocity field, obtaining an intermediate velocity field. They then used iterative methods to generate a pressure field which changed the motion of the fluid such that equation 2 was satisfied at every time step, and recorded the time steps to create an animation. This approach runs into difficulties: it is unstable for large time steps, meaning that small errors tend to amplify themselves and grow without limit, and within a few frames the simulation explodes, rendering it useless[1]. The time step restriction also becomes more stringent as the grid resolution is increased, making this approach potentially very expensive. The stable fluids algorithm addresses this shortcoming.

### 3.2.1 The MAC Grid

Before discussing how to solve the equations, we must establish how to arrange and interpret the data grids the algorithm operates on. This is straightforward for scalar quantities like pressure - we just store them at regular intervals on a cubic grid. We could also choose to store the components of our velocity field at the same locations - however, this can lead to problems.

Equations 1 and 2 involve the derivatives of vector quantities affecting those of scalar quantities and vice versa. For example, one of the terms in equation 1 relates the time derivative of the x component of velocity with the derivative of the pressure field in the x direction. The question is: how do we approximate

the pressure derivative on our discrete grid? If we are storing the velocities at the same location as the pressures, we can approximate it as follows:

$$\frac{\partial p}{\partial x} \simeq \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x}$$

This method of taking a derivative has a major disadvantage however - it is one sided, meaning information only propagates from one direction, and this can introduce strange biasing artifacts in a simulation. Additionally, it turns out that there are ways of estimating the derivative which give better accuracy. It can easily be seen that this particular derivative operator, known as a one sided difference, gives an exact result for the derivative of $p$ if $p(x)$ is of the form $p(0) + ax$ - a first order polynomial in x. Because of this, we say this method is first order accurate - it turns out that if we use it to estimate the derivative of any higher order polynomial, the error in the estimate will have terms proportional to $\Delta x$ and higher[1]. We can do better than this by taking a so called central difference:

$$\frac{\partial p}{\partial x} \simeq \frac{p_{i+1,j,k} - p_{i-1,j,k}}{2\Delta x}$$

It can be shown that this estimate gives an exact result for quadratic polynomials in $x$, and is second order accurate - the error in the estimate has terms in $\Delta x^2$ and higher, decreasing much more quickly as $\Delta x$ becomes very small than with the first method. It also has the advantage that information propagates both ways.

Unfortunately this method also has a serious disadvantage. If, for example, $p_i$ is defined as 0 for even $i$ and 1 for odd $i$, then taking the central difference anywhere will yield a zero derivative - something which is obviously untrue. This is known as a "null space" problem, and can lead to "parasitic" oscillations in a simulation, which in general there is no satisfactory way around.

Harlow and Welch[29] solved these problems by proposing the Marker And Cell (MAC) grid structure, in which quantities like pressure are stored on a traditional cubic lattice, but instead of storing the velocities at the same locations, their components are split and stored on separate faces of the cubic cells. For example, the x components of the velocity are stored on the +/-x faces of the cells, etc. With this structure, we can still take central differences if we wish to know the derivative of the pressure where a velocity component is stored, avoiding bias and retaining second order accuracy, but we now have no null space problem.

### 3.2.2 Operator Splitting

A central idea of the stable fluids algorithm is that of "operator splitting". In this scheme, instead of calculating $\frac{\partial u}{\partial t}$ and adding on the update all in one step,

[1] Actually, this is the case for any smooth function, for example sin(p), because any function like this can be written as a polynomial of infinite order using Taylor series.

we split $\frac{\partial \underline{u}}{\partial t}$ into its separate terms and update the velocity field separately for each one, using techniques specialized for each term. These techniques may be optimized for stability, accuracy, or computational cheapness if they are not particularly problematic. To advance the simulation forward by one time step, we perform the following steps in order:

### 3.2.3 The Advection Term

The so called advection term is dealt with first, under the assumption that the velocity field already satisfies equation 2, and we perform this step of the algorithm by solving the equation $\frac{\partial \underline{u}}{\partial t} + (\underline{u}.\nabla)\underline{u} = 0$ for a single time step. Advection refers to the passive motion of objects or quantities through a moving fluid. If a quantity $c$ (vector, scalar or otherwise) is passively carried along with a fluid having a velocity field $\underline{u}$, then it will satisfy the so called advection equation:

$$(\frac{\partial}{\partial t} + \underline{u}.\nabla)c = 0$$

This equation is derived by requiring that rate of change of $c$, measured at a point co-moving with the fluid, remains constant. In fact, the operator on the left of $c$ in this equation, $(\frac{\partial}{\partial t} + \underline{u}.\nabla)$, is known as the convective derivative, and measures this rate of change. The advection term of the Euler equations is essentially this equation, with the quantity c replaced by the velocity field $\underline{u}$ itself.

In engineering applications, where a high degree of accuracy is required, the advection term is usually solved by explicitly considering the derivatives, using sophisticated approximations. These techniques often pose restrictions on the time step, and are generally quite computationally expensive. Stam [23] proposed a way of avoiding these time step restrictions, by using the so called method of characteristics - an intuitive way of looking at the advection problem, which works by exploiting the fact that the quantity being advected is constant at a point co-moving with the fluid. The paths followed by these points are called "characteristics". To advance find the value of $c$ one time step into the future at a position $\underline{r}$, we simply walk back by a single time step along a characteristic starting at $\underline{r}$, and evaluate $c$ at the point we end up at. To walk back along a characteristic, we can use Euler's method[2] and set $\underline{r}(t-\Delta t) = \underline{r}(t) - \underline{u}\Delta t$. This is known as the "Semi-Lagrangian" method, because it borrows from the Lagrangian framework by moving effectively moving the grid points and resampling.

Using this technique, we can find an updated velocity field $\underline{u}*$ by evaluating $\underline{u}$ at the point we are interesting in, finding the point $\underline{r} - \underline{u}\Delta t$, and setting $\underline{u}*$ to the value of $\underline{u}$ at that point. On a MAC grid, there is no point at which all the components of $\underline{u}$ are all defined at once, as the components are defined separately on the faces of the cubic cells. Therefore, to evaluate $\underline{u}$ at any point,

[2] Euler's method often performs poorly for rotational motion, so we can use a more sophisticated method if we need to.

we must interpolate the components. The most straightforward way to do this is by trilinear interpolation.

### 3.2.4 Body force term

The next term we use is called the body force term, and is the easiest term to deal with. It does not lead to any instabilities, so it can be dealt with using a very simple technique. We compute this term by solving the equation $\frac{\partial \underline{u}}{\partial t} = \frac{\underline{f}}{\rho}$ for one time step, so approximating the derivative with a simple one sided difference, we can update the velocity field by simply adding $\frac{\underline{f}}{\rho}\Delta t$. This scheme is known as the Euler method.

### 3.2.5 The Pressure Term

The most complicated step in the solution procedure is calculating the pressure field $p$, so it can be used to update the velocity field, completing our time stepping procedure. We are modeling an incompressible fluid, so we must calculate $p$ such that the velocity field is divergence free following the update. The pressure term models the internal forces in the fluid, and is responsible for most of its interesting rotational motion.

By taking the divergence of the Euler equations, we can arrive at the following condition on the pressure field[1]:

$$\frac{\bigtriangledown^2 p}{\rho} = \bigtriangledown.(-\underline{u}.\bigtriangledown \underline{u} + \underline{f})$$

This is true in the continuous limit, but in our discrete simulation we will not use this, opting instead for a more direct approach. Having run through the previous two steps, we now have an intermediate velocity field, which we will call $\underline{u}*$. This will typically not satisfy the incompressibility equation. What we require is a pressure field $p$ which will make$\underline{u}*$ divergence free when we update it using the pressure term, ie we obtain:

$$\underline{u} = \underline{u}* - \frac{\bigtriangledown p}{\rho}\Delta t$$

Such that $\underline{u}$ satisfies the incompressibility equation. We can obtain an equation for $p$ by taking the divergence of both sides of this equation:

$$\bigtriangledown.\underline{u} = \bigtriangledown.\underline{u}* - \frac{\bigtriangledown^2 p}{\rho}\Delta t = 0 \tag{3}$$

Where $\bigtriangledown^2 p$ is the "laplacian" of $p$, which in Cartesian coordinates is $\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2}$. We can simplify this equation slightly by defining a variable $p' = \frac{p}{\rho}\Delta t$, and rearranging equation 3 to obtain:

$$\bigtriangledown^2 p' = \bigtriangledown.\underline{u}* \tag{4}$$

We must now find the counterpart of this continuous equation on our discrete grid. To do this, we again approximate all derivatives using finite differences, giving us a very large system of simultaneous equations. To solve this system of equations, we use an iterative method called the "conjugate gradient" method.

### 3.2.6 Boundaries

In a fluid simulator, the pressure equations need several special cases. For example, the grids we use are finite, so care must be taken when we take derivatives across the edges of the domain - along with this, the fluid must flow round and interact with objects the artist places in the domain. Problems like this, when we specify certain things on the boundaries of our domain, are known as boundary conditions. We use two kinds of boundary conditions in the simulation, which are technically known as Dirichlet and Von Neumann boundary conditions. Dirichlet boundary conditions explicitly specify the values of a quantity on a surface, whereas Von Neumann conditions specify the derivative of a quantity in a direction normal to a surface. We use Diriclet boundary conditions on the edges of the domain, requiring that the pressure be equal to zero on the edges, allowing the fluid to move freely in and out of the domain. At the surface of solid objects we use Von Neumann boundary conditions, setting the derivative of the pressure to be zero in a direction normal to the surface. This way we can set the velocity of the fluid on that surface in a previous step, and the pressure step will leave the normal component of the velocity unaltered, allowing fluid to flow round objects, and allowing solid objects to push on the fluid. This topic is covered in more detail in section 4.2.3.

### 3.2.7 Vorticity Confinement

A major problem with the stable fluids algorithm is one known as "numerical dissipation". Numerical dissipation has the effect of unintentionally adding viscous behavior to the fluid, which can damp down some of the intricate turbulent behavior seen in natural smoke and fire. This arises mainly from the advection step in the solver: a given velocity value is updated by tracing back a characteristic through the domain and then interpolating the nearby velocities to calculate the velocity where the characteristic landed. This interpolation is effectively a weighted average, and has the effect of blurring the velocity field over many time steps - this destroys rapid variations in the velocity field, and unnaturally reduces turbulence. Vorticity confinement combats this by essentially detecting vortices and forcing back in some of the rotational motion that numerical dissipation took away.

The rotational motion of a velocity field can be related to a vector quantity called "curl". Intuitively speaking, curl is a "paddle wheel" force. Imagine carrying a tiny paddle wheel around and placing it somewhere in a fluid - the fluid's motion will be slightly different on each side of the wheel, and under the right conditions it will start spin. Now orient the paddle wheel in the direction which gives it the maximum torque. Roughly speaking, the curl of the fluid's

velocity field at that point is a vector pointing along the axis of the wheel, with a magnitude proportional to how fast it is spinning. More precisely, curl is motivated and defined using a mathematical result called Stokes' theorem (see [28] for more details).

It turns out that this useful quantity can be denoted $\bigtriangledown \times \underline{u}$, where

$$\bigtriangledown \times \underline{u} = \begin{pmatrix} \frac{\partial v}{\partial z} - \frac{\partial w}{\partial y} \\ \frac{\partial w}{\partial x} - \frac{\partial u}{\partial z} \\ \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} \end{pmatrix}$$

In fluid dynamics, tubes of rotational motion or vortices form, which can be detected by looking for regions where the curl field has a high magnitude. Our task is to identify these structures and apply a force which accelerates their motion.

To generate such a force, we first find the curl of the velocity field, which we will call $\underline{\omega}$, and its magnitude, $\Omega = |\underline{\omega}|$. We can then take the normalized gradient of this quantity, which we will call $\underline{N}$:

$$\underline{N} = \frac{\bigtriangledown \Omega}{| \bigtriangledown \Omega|}$$

This is a field of unit vectors which point towards the center of the vortices in our fluid, as it points towards regions of high vorticity. We can obtain a field of vectors going round the vortices in the direction they are moving by taking the cross product of $\underline{N}$ with $\underline{\omega}$. We find our vorticity confinement force by multiplying this by a constant $\varepsilon$, and the grid spacing $\Delta x$, so that the force gets smaller as the cells get smaller, and we converge on the true solution to the equations. The final formula for the confinement force is written below:

$$\underline{f}_{conf} = \varepsilon \Delta x (\underline{N} \times \underline{\omega})$$

## 3.3 The Flame Model

Implementing these features will give us a basic fluid solver, which is capable of producing pleasing animations of passive phenomena like smoke and clouds. To model the dynamics of fire, we must make some modifications, which will now be described.

At the heart of most flames is a structure known as a deflagration front. This is a thin layer separating a region of air from a region of fuel, in which a chemical reaction takes place, and can be successfully modeled as being infinitely thin[7]. The reaction releases heat, and changes the chemical nature of the gas, leading to a rapid expansion of the reaction products - a process through which much of the turbulent, chaotic behavior of flames arises. The exhaust then rises, as it is hotter and less dense than the air around it, and eventually cools - this can be modeled as part of the body force term in the Euler equations. A fire simulator must model and couple together the dynamics of the gases either side

of the reaction front, the dynamics of the front itself, and the effects of the gas expansion.

In our model, the reaction front will move along with the fluid, and additionally, the surface will spread into the fuel region with a uniform speed normal to the surface, in a frame of reference co-moving with the fluid. Burning objects can be modeled by enclosing the object in fuel, and adding a velocity perpendicular to surface of the object onto the surrounding fluid.

### 3.3.1 The Level Set Method

It can be difficult representing the reaction front using, for example, an explicit polygon mesh[11]. The motion of a fluid can be very irregular, meaning the quality of the mesh can deteriorate rapidly if its verticies are simply swept along with the fluid. The reaction front can also experience topology changes, where two separate fronts can merge, or a piece of fuel can break off and float away, which is also difficult to handle with an explicit representation. A much more robust way of tracking the surface is by representing it using an isosurface of a *level set* function, often denoted $\phi$. This is the approach taken by the simulator, with all regions containing fuel having level set values greater than or equal to zero, and all other regions containing air.

Given a level set function defined on a grid, and a velocity field at a given time, we wish to find the level set function after one time step has elapsed. We do this by considering how the surface defined by the level set is moving at that point. Take the unit normal, $\underline{\hat{n}}$, as pointing into the region of fuel. According to our simple flame model, the surface must move with a constant, uniform speed $S$ in the direction of $\underline{\hat{n}}$, and must also move with the fluid, having a velocity field $\underline{u}$. Therefore, the total speed of the surface at that point, which we denote $\underline{w}$, is found by adding these terms together:

$$\underline{w} = S\underline{\hat{n}} + \underline{u} \tag{5}$$

We must now find an equation of motion for$\phi$such that its isosurfaces move in this way. We do this by considering a characteristic - a point which moves as a function of time in such a way that $\phi$ stays constant when evaluated at that point. We will denote the point $\underline{r}(t)$. Along this characteristic, the value of $\phi$ remains constant, ie:

$$\frac{d}{dt}\phi(\underline{r}(t)) = 0$$

We can now expand the time derivative using the chain rule to obtain:

$$\frac{\partial\phi}{\partial t} + \frac{\partial\phi}{\partial x}\frac{dx}{dt} + \frac{\partial\phi}{\partial y}\frac{dy}{dt} + \frac{\partial\phi}{\partial z}\frac{dz}{dt} = 0$$

$$\frac{\partial\phi}{\partial t} + \nabla\phi.\underline{v}(t) = 0$$

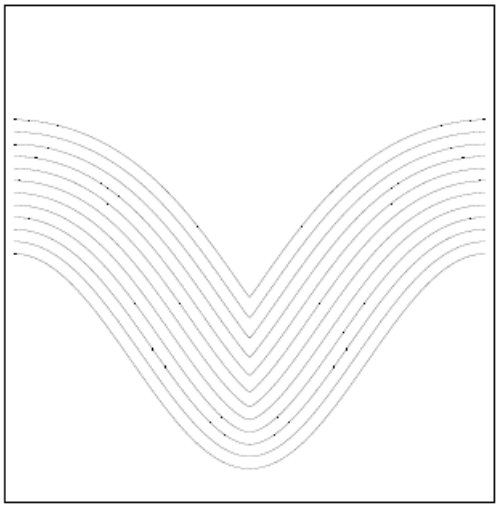

Figure 1: Pitted surfaces will eventually lead to creases and cusps, as shown above for an advancing sine wave (figure taken from [4])

Where $\underline{v}(t)$ is the time derivative of $\underline{r}(t)$, ie the characteristic's velocity. However, we know the desired behavior of this point from equation 5. Because of the way our level set is defined, the fuel-facing unit normal is given by $\hat{\underline{n}} = \frac{\nabla\phi}{|\nabla\phi|}$, and $\underline{v}(t) = \underline{w}$. Therefore:

$$\frac{\partial\phi}{\partial t} + \nabla\phi.\underline{w} = 0$$

$$\frac{\partial\phi}{\partial t} + \underline{w}.\hat{\underline{n}}|\nabla\phi| = 0 \tag{6}$$

We refer to equation 6 as our level set equation, which we must now solve numerically.

The level set equation presents a hidden difficulty, illustrated in figure 1. If the surface is pitted or grooved in any way, and the normals to the surface converge, some of the characteristics will eventually cross paths, and the surface will develop a cusp. This can present difficulties, as the gradient of the surface will become discontinuous. To correctly handle situations like this, we can use an "upwind" difference operator to discretize equation 6. A simple discretization, used by Fedkiw et al in [7] is to approximate the derivatives by taking a one sided difference in the direction $\underline{w}$ is coming from. This means, for example, that if the x component of $\underline{w}$ is greater than zero, we approximate $\frac{\partial\phi}{\partial t}$ with $\frac{1}{\Delta x}(\phi_{i,j,k} - \phi_{i-1,j,k})$

It turns out that this scheme requires we limit the time step such that $\Delta t < \frac{\Delta x}{|\underline{w}|}$ everywhere, meaning we may have to take fractional steps in the level set update part of the simulation. However, because we only need the zero

isocontour of the level set function, we need not define it everywhere, and we can therefore skip over most of the domain when applying the update.

### 3.3.2 Regenerating the Level Set

The surface tracking method described in the previous section works well if the level set is a so called Signed Distance Function (SDF). This is a function obtained by plotting the minimum distance to a surface, in this case the reaction front, throughout the domain. A numerically convenient property of an SDF is that the magnitude of its gradient is always the same - ie $|\nabla\phi| = 1$, and it is desirable to keep our level set function close to an SDF throughout the simulation. Unfortunately, the level set equation does not preserve this property, so we must periodically regenerate the level set function. However, as we only need the zero isocontour of the function, we need only do it in a narrow band either side of the isocontour.

We achieve this regeneration using the so called Fast Marching Method, first proposed by Sethian in [4]. The SDF property $|\nabla\phi| = 1$ can be treated as a partial differential equation called the eikonal equation, equivalent to $|\nabla\phi|^2 = 1$, which we discretize using an upwind finite difference formula. $|\nabla\phi|^2$ is equivalent to $\frac{\partial\phi}{\partial x}^2 + \frac{\partial\phi}{\partial y}^2 + \frac{\partial\phi}{\partial z}^2$, and we approximate each of the derivatives with a one sided finite difference in the direction which yields the largest absolute value.

The fast marching method propagates the SDF outwards from a region of known cells - it can do this in either the increasing or the decreasing direction. To save time, we can terminate the procedure for cells greater than an upper threshold, or less a lower threshold.

The algorithm keeps three lists: a list of cells for which the SDF is known, a list of cells for which it is unknown, which we initialize to the upper or lower threshold, depending on weather we are going in the increasing or decreasing direction. We also keep a list of cells with tentative values. To establish these lists in the first place, we use a method which will be described later. We then loop over the following procedure:

1. If the "tentative" list is empty, exit. Otherwise, grab the cell with smallest value in the list, and remove it from the list. Add it to the "known" list.

2. If the newly finalized cell has a value outside the permitted range, clamp it to the range and go to step one. Otherwise, scan the unknown and tentative cells in the newly finalized cell's six cell neighborhood. If they are in the unknown list, compute their values and add them to the tentative list. Otherwise recompute their values.

3. Return to step one.

We initialize our lists as follows: first, we generate signed distance values on cells adjacent to the zero isocontour. This is an important step, because we want the isosurface to move as little as possible. We achieve this by finding a gird point with neighbors on the opposite side of the isosurface, and then finding the

gradient vector of $\phi$ at that point. We then set the signed distance value of our new level set function, $\phi'$, to $\phi/|\bigtriangledown\phi|$ at that cell. We measure the components of $\bigtriangledown\phi$ by taking one sided differences across the the surface, in the direction which gives the largest absolute value if the cell has neighbors the opposite side of the surface either side of it. This was found to cause the minimum change in the isosurface. We add these cells to the list of known signed distance values.

Next we generate a band of signed distance cells on both the positive and negative sides of this. We do so at each of these cells by writing the equation $|\nabla\phi|^2 = 1$ as a quadratic equation, using the usual upwind differencing methods, and taking either the most positive solution if we are generating cells in the positive region, or the most negative solution if we are in the negative region. This is also the method we use to compute signed distance values once we enter the main loop. We can now label all other cells unvisited by setting their SDF values to the upper limit if they are on the positive side of the isosurface, or the lower limit if they are on the negative side. Finally, we start the main loop for two fronts, one moving in the positive direction and one in the negative direction.

An important step in the main loop is finding the cell with the minimum SDF value in the tentative list. This can be accomplished using a min heap - see [10] for details. A min heap uses a tree structure to keep track of its smallest element, which is located at the root of the tree. Each node has two children, each of which must be smaller than their parents. Every time an element is added to the heap or modified, the tree is rearranged so this property is preserved. The heap structure used in the algorithm also requires pointers to cells on the domain, so a heap class was implemented to code the algorithm.

### 3.3.3 Jump Conditions

The reaction front separates two regions of the gas with different densities and spreads into the fuel region, converting one to another. This, along with the conservation of mass and momentum, leads to the so called Rankine-Hugoniot jump conditions across the interface:

$$\rho'(u'_n - S) = \rho(u_n - S)$$

$$\rho'(u'_n - S)^2 + p' = \rho(u_n - S)^2 + p$$

Where unprimed quantities indicate quantities immediately inside the reaction surface, and primed quantities indicate those immediately outside, ie quantities in regions of fluid which have just reacted. The discontinuities in the velocity field at the deflagration front are responsible for a lot of the motion at the base of the flame, and must be incorporated into our numerical model. The flame speed S, and the ratio $\frac{\rho'}{\rho}$ are two parameters which have a lot of effect on the appearance of the flame, and the user should be able to specify them.

### 3.3.4 Refinements to the Original Algorithm

The original stable fluids algorithm treats the grid values as an approximation to a continuous function, and will quickly smooth out any discontinuities such as these, through interpolation and the dynamics of the system. We do not want this, and we must find a way of explicitly maintaining these discontinuities within our algorithm. One such method is known as the Ghost Fluid Method[2]. When we take the derivative of a quantity across a discontinuity using a finite difference, we get a quantity which increases indefinitely as our cell spacing decreases, rather than converging on a value. The Ghost Fluid Method fixes this problem by taking the jump conditions into account at the discontinuities. If the derivative of a quantity is desired in the fuel region for example, instead of taking a finite difference using the grid values of this quantity, it checks if any values in the finite difference stencil are on the other side of the interface, and replaces them with "ghost" values by subtracting the jump conditions from them. A similar technique is used when the simulator has to interpolate between two quantities defined at opposite sides of the interface, and when a characteristic crosses the interface in the semi Lagrangian advection phase. This method must also be used to modify the linear system in the pressure solver. For more details on how these modifications are implemented, see [12] and [11] .

# 4 Implementation

The numerical algorithm described in the previous section was implemented as part of a standalone simulator program with a graphical user interface. The program, uses the FOX toolkit[3], a cross platform GUI library with OpenGL support, which compiles under Windows and Linux. The interface allows the user to see the simulation in 3D while it is running, visualizing the objects in the simulation, along with the smoke and flames as isosurfaces of some of the volume data. The simulation caches these isosurfaces so the user can play back the animation, and it also saves some of the data grids to disk so they can be used in rendering. The interface also has a render preview feature, and allows interactive editing of parameters like the speed of the flame, the gravity vector, buoyancy and smoke curves, etc. A screen shot is shown in figure 12.

Initially, a standard stable fluids solver was implemented, using the so called preconditioned conjugate gradient algorithm to solve the pressure equations (see section 4.1). This was then extended to deal with fixed and moving boundaries. Algorithms concerning the level set were then implemented, and finally jump conditions were integrated into the solver.

The simulator operates on solid objects, which are loaded in from the disk and can interact with the fluid and emit fuel. Three kinds of object are supported: ellipsoids, static polygon meshes and dynamic polygon meshes. Each object in the simulator has a transformation matrix, which can vary over time, and a number indicating how much fuel it is emitting, if any. Along with the

[3] www.fox-toolkit.org

motion generated from varying its transformation matrix, the verticies of a dynamic mesh can move in arbitrary ways. All this motion is coupled to the fluid.

The code for the simulator was written in C++ and encapsulated in a class called CSimulator, which has member variables for the data grids, various methods for adding and editing parameters and objects in the simulation, and an "advance" method, which advances the fluid forward through one time step. The pseudo code for the advance method is shown below:

```
updateObjects();
voxelizeOccluders();
advectFields();
emitFuel();
advectVelocities();
addForces();
projectVelocities();
```

In the updateObjects stage of the algorithm, the simulator loads new values for the object transforms, and new states for the dynamic meshes, and calculates velocity data, which is used later when coupling the objects to the fluid. The voxelizeOccluders method then converts all objects to a volumetric representation and sets velocities at their surfaces. Quantities like smoke density are then swept along with the fluid in advectFields, fuel is placed in the domain in emitFuel, and the advectVelocities, addForces and projectVelocities methods deal with the advection, body force and pressure terms of the Euler equations respectively.

## 4.1 The Pressure Solver

The most difficult phase of the simulator to implement is the pressure solver. This is mainly because of the boundary conditions that have to be enforced to couple the solid objects to the fluid, which can be difficult to debug if coded incorrectly. Equation 4, when written in discrete form, gives a large number of simultaneous equations similar to the following:

$$\frac{-6*p_{i,j,k}+p_{i-1,j,k}+p_{i,j+1,k}+p_{i,j-1,k}+p_{i,j,k+1}+p_{i,j,k-1}}{\Delta x^2} = \frac{u_{i+\frac{1}{2},j,k}-u_{i-\frac{1}{2},j,k}+u_{i,j+\frac{1}{2},k}+u_{i,j-\frac{1}{2},k}+u_{i,j,k+\frac{1}{2}}+u_{i,j,k-\frac{1}{2}}}{\Delta x}$$

This is and equation for a cell at a position i,j,k, without any solid objects, domain boundaries or portions of the flame interface in its six cell neighborhood. The presence of such objects, ie boundary conditions, will lead to different equations.

Typically, for a large grid, there will be millions cells and millions of these equations. Large systems like this are commonly solved using matrix techniques. If we order our cells along one dimension, for example by running up the x axis, going back to zero and incrementing our y coordinate when we have reached the

maximum x coordinate, and doing the same with the z coordinate, we can write all the pressures and the right hand side terms of these equations as column vectors. Once we have done this, the system of equations can be summarized in the form $M\underline{p}' = \underline{d}$, where $\underline{p}'$ and $\underline{d}$ are large column vectors, containing the pressures and right hand side terms respectively, and $\underline{M}$ is a square matrix. Normally, this kind of system is solved by finding the inverse of the matrix, or using techniques like Gauss elimination. However, in this case $\underline{M}$ is enormous, and can easily have over 1,000,000,000,000 elements, meaning its inverse or row reduction cannot possibly be stored on today's computers. Fortunately, $\underline{M}$ is sparse, meaning most of its entries are zero, and it has a special structure, in that its nonzero elements lie on the diagonal, and additionally on six bands parallel to the diagonal. The matrix is also symmetric, meaning that swapping its rows and its columns does not change it, which further reduces the amount of bands in the matrix which have to be stored to 4.

The solution to this system is obtained in an approximate sense using an iterative method. An iterative solution algorithm is one which starts with an initial approximation to the solution and progressively refines it, using a procedure which, ideally, is guaranteed to bring the approximation closer to the true solution with each step. Particularly effective for this kind of system is the so called Conjugate Gradient algorithm, which treats the problem as one of minimizing a high dimensional quadratic function. It can be made to perform especially well by combining the matrix with a "preconditioner", which is an approximate inverse of the matrix - this makes the algorithm converge on a solution in a smaller number of iterations. For a detailed mathematical description of the conjugate gradient algorithm, see [8], and for practical implementation details including the implementation of a preconditioner, see [1].

## 4.2 Solid Boundaries

### 4.2.1 Volumetric Representation of Objects

There are three problems which must be overcome before objects can interact with the fluid. The first is converting the objects to a volumetric representation, so they can be understood by the solver. For this, the objects need to be converted to voxels. The simulator uses an ID buffer for this, storing the ID numbers of the objects as one byte per voxel. Voxelizing ellipsoids is a straightforward task. Ellipsoids in the simulator are defined as the image of a unit sphere under a matrix transform. To voxelize an ellipsoid, the simulator finds a conservative estimate of its bounding box, and then transforms all the points in the bounding box back to the space in which the ellipsoid is a unit sphere. If the object's normal is required, it is fetched from the unit sphere space and transformed back to world space using the transpose of the inverse of the ellipsoid's matrix. It turns out that this keeps normals orthogonal to surfaces undergoing that transform, although unsurprisingly it alters their length. Meshes are more difficult, as they are not represented implicitly, and an inside-outside test is more challenging. Initially, ray tracing was considered as a means to carry out

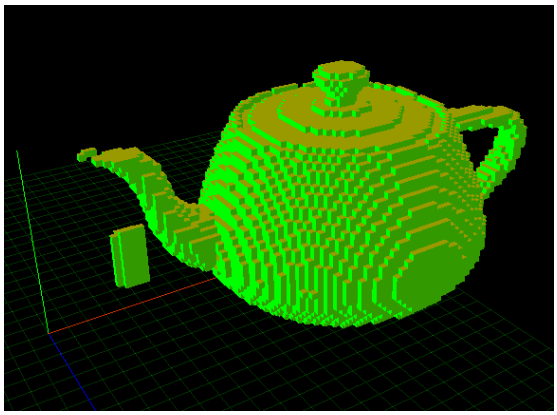

Figure 2:

this test. If a surface is closed, a point can be classified by drawing a line between that point and a point known to be outside the surface. If the line crosses the surface an odd number of times, the point is inside the object, otherwise it is outside. This can potentially be very slow for highly tessellated objects though, and would limit the complexity of the objects the sim could work with. The code would also be complex. Efficient ray tracing requires an acceleration structure like a KD tree, the creation of which has problems of its own. The solution which was adopted, following some discussion with lecturers and peers, was a scan conversion algorithm. This method is essentially the same as the ray tracing method, in that it counts intersections with the surface, but it does this by marching over the surface rather than testing ray intersections, and is therefore much faster and easy to code. Initially, all voxels in the grid are set to empty. The object is then viewed looking down the x axis and all the triangles in the object are essentially rasterized on the simulation grid - all voxels behind the point being rasterized have their states flipped, ie full cells become empty and vice versa. If the mesh is good, all grid points will be correctly classified once this procedure has been completed. For added robustness, to handle some cases when the mesh is non manifold, the procedure is performed using all three axis and a majority vote is taken. For non manifold meshes, this can still lead to artifacts, which are shown in figure 2 for a Utah teapot model. After some reading it was found that this algorithm was used by Frantic Films[22] for fluid work on the film Scooby Doo 2, and integrated into their fluid simulation software package, Flood, with several extensions involving ray tracing to make the algorithm more robust.

Two methods of scan converting the triangles were tried. The first found the rows of grid points corresponding to the top and bottom of the triangle, and then moved from top to bottom finding the left most and right most grid points in each row. It then went through all the points in each row and used the plane equation to find the position of the point being visited on the remaining

axis, and flipped the state of all voxels beyond that position. This method runs into a lot of numerical problems however, which can cause poor performance for complex meshes, and require complex code to overcome. Instead of this, a slightly slower but more robust method was used, where the bounding box of the triangle was found, and each grid point within the box was explicitly tested against the edges of the triangle. Care had to be taken that exactly the same edge tests were carried out by two triangles whose bounding boxes overlapped, and to avoid complexity when disambiguating edges, points were perturbed slightly when they landed exactly on an edge. This gave good performance, although there were still occasional problems when triangles were viewed at extreme angles.

The final step in voxelizing the objects is a flood fill. The fluid is incompressible, and therefore the pressure solver is only able to deal with situations in which the fluid is not forced to compress. If there is a solid object with a shrinking cavity from which no fluid can escape, for example, the solver will be completely unable to find a solution, and often gives the velocity field arbitrary values and ruins the simulation. To avoid this, we make sure that there is a path to the edge of the fluid domain, through which fluid can freely flow, for every empty cell, and we do this using a flood fill, starting from the edges of the domain. We first mark all the empty cells as tentatively empty. Active cells are held in a C++ STL queue, initialized to be all empty border cells in the domain, and at every step in the algorithm a cell is taken from the front of the queue, it is marked as empty, and its neighbors are examined. If any of them are marked tentative empty they are added to the back of the queue, otherwise they are skipped over. This is repeated until the queue is empty - following that, all tentative empty cells are marked occupied.

### 4.2.2 Setting an Object's Surface Velocities

The second problem when coupling solids to the fluid is setting the velocities of the fluid on the surface of the objects. Ellipsoids are the most straight forward objects to deal with, and only move via matrix transforms, meaning the velocity of any point on the ellipsoid can itself be calculated using a matrix. Consider a point $\underline{r}$ in world space, lying on the surface of the ellipsoid, whose transformation matrix at frame n is $M_n$. The corresponding point in object space, ie the space in which the ellipsoid is a unit sphere, is $M_n^{-1}\underline{r}$. The world space position of the same object space point at frame n-1 is $M_{n-1}M_n^{-1}\underline{r}$, and therefore between frame n-1 and frame n, the object has moved by the vector $\delta\underline{r}$:

$$\delta\underline{r} = \underline{r} - M_{n-1}M_n^{-1}\underline{r}$$

$$= (I - M_{n-1}M_n^{-1})\underline{r}$$

$$= V_n\underline{r}\Delta t$$

Where I is the identity matrix, and $V_n = \frac{1}{\Delta t}(I - M_{n-1}M_n^{-1})$ is the velocity matrix for frame n. This reasoning applies to any object moving due to a varying matrix transformation. The matrix is used to calculate the velocity at every grid point contained by the ellipsoid, which is then projected onto the surface normal to prevent transverse motion of the object dragging the fluid with it[4], and finally it is stored on the grid. This is performed for all points within the ellipsoid, as it is desirable that velocities be defined inside objects as well as outside them.

A similar method is used to set the velocities on the surface of static meshes. This time, the velocities are painted onto the surface of the object by rasterizing the triangles, and then extrapolated inside the object using an algorithm similar to the fast marching method, where whenever a voxel is visited it is assigned a velocity averaged from the cells the signed distance function was propagated from. Dynamic meshes are treated in a similar fashion, only in this case the velocities of each vertex are computed and interpolated across the triangles, before being added on to the velocity computed with the transformation.

### 4.2.3 Coupling to the Fluid

Solid objects can be made to communicate their motion to the fluid via the pressure field - this is accomplished by ignoring all pressures inside an object and setting all pressure gradients measured across an object's surface to zero, so that essentially any force applied to an object by the fluid is returned by the object, and there is no net acceleration of the fluid across the object's surface. This is done by directly replacing the gradients measured across the surfaces, for example $\frac{p_{i+1,j,k}-p_{i,j,k}}{\Delta x}$, with zero, both when setting up the linear system, and when adding the pressure forces later on. Note that it may not necessarily be possible to explicitly set the pressures inside the object so this is satisfied, as a solid cell can have more than one non solid neighbor, leading to two possibly conflicting conditions on the cell's pressure.

## 4.3 Optimization

Initially, a sparse matrix was used to record the pressure equations for a particular grid. This requires approximately 4 floating point numbers or 16 bytes per voxel, and takes a large amount of memory. The maximum grid size that could be simulated before this and other memory optimizations were carried out was 217 by 217 by 217. Following [22], the system of equations can be divided through by a number of factors, so that all the entries in the matrix become integers. We can then store the elements of the matrix using a single byte per voxel, as the diagonal elements only go up to six, and can be stored using 3 bits, and the entries on the three other bands we need to store are either 0 or

[4] Non viscous fluids behave as if they slide freely over the surface of solid objects, although in reality the fluid velocity actually matches that of the object at the interface. However, a boundary layer forms, where the fluid essentially lubricates itself, and in fluids with low viscosity this layer is thin, and the fluid behaves as if it flows freely past the object.

1, and only need one bit each. After this optimization was implemented, it was possible to push the grid size past 300 by 300 by 300.

### 4.4 Buoyancy and User Defined Curves

The buoyancy of a particular element of the flame is defined as a function of the time since it crossed the reaction front, which is stored on a grid and updated every frame using the semi Lagrangian advection algorithm. The time dependence of the buoyancy is controlled using a piecewise linear lookup table. This is implemented as a class, which has a list of data points passed to its constructor, and has a "val(float fT)" method, which identifies which two data points the variable lies between, and linearly interpolates them. A potentially slow part of this algorithm is determining the interval in which a point lies. To accelerate this process, the lookup table class uses a binary search strategy. In the constructor, it first sorts the data points in ascending order by their x coordinates, and then permutes this list so it forms an interleaved binary tree, stored in breadth first order. This tree has the property that any given node's left child and all its descendants have an x coordinate less than that of the parent node, and the node's right child and all its descendants have an x coordinate greater than the parent node.

To evaluate the peicewise linear function for a variable, all the class then needs to do is take the data point at the root, stored at position 0 in the array, and test weather the coordinate we are evaluating the function at is greater or less than the x coordinate of this data point. If it is greater, it moves to the right child and repeats this procedure, and if not it repeats the procedure for the left child. It repeats these steps until it reaches the bottom of the tree, at which point it can identify the interval the evaluation point lies within, and then linearly interpolate the data points. This algorithm is simple and fast, executes in O(ln(n)) time, where n is number of points in the lookup table.

This kind of table is also used to define the incandescent colours of the flame, again in terms of the time since the fluid crossed the reaction front, and also the rate of smoke generation as a function of time, with the smoke density stored on a different data grid.

To generate a plausible lookup table for the buoyancy force, the method in [7] was used. This is based on the fact that the rate of radiative heat loss from an object is proportional to its temperature to the fourth power, and the buoyancy of a volume of gas is proportional to its temperature. This gives a differential equation for the buoyancy as a function of time, which can be analytically solved.

## 5 Rendering Smoke and Flames

The most natural way to render volumetric phenomena like smoke and flames is by ray marching. In this technique, we integrate the light delivered to the eye along a line through the volume, by walking along it in small steps.

With smoke and fire, there are two components to the light which reaches the eye along a given ray. The first is the light arriving directly from the incandescent gas in the flames. The second component is light from the flames and external sources which has been scattered in the direction of the eye by the smoke - this can happen after one or more scattering events.

The fire was rendered using a volume shader in PRMan. In the shader, we define the ray the camera is looking down parametrically, in the form $\underline{r}(t) = \underline{o} + \hat{\underline{d}}t$, where $\underline{o}$ is the origin of the ray (a point on the near clip plane), and $\underline{d}$ is a unit vector defining its direction. The parameter variable, $t$, initially ranges from 0 to a user defined maximum value, defining a line segment. The line segment is first clipped to the bounding box of the fire, and then marched along with a user defined step size, keeping track of two variables: the total light radiated towards the eye, $C$ (as a colour), and the visibility of the point we are visiting, $v$, having been occluded by the smoke. At each step, these variables are updated as follows:

$$v_n = v_{n-1}(1 - \rho\Delta t)$$

$$C_n = C_{n-1} + v_n(S + I)\Delta t$$

Where $\Delta t$ is the user defined step size, $\rho$ is the smoke density, $I$ is the light emitted by the incandescent gas, and $S$ is the light scattered from the smoke.

The most challenging part of the volume shader is computing the scattered light. This is a global illumination problem, depending on the geometry of the surrounding smoke, and can be very computationally intensive. However, good results can be achieved by making this proportional to the light received by the point, computed using a renderman illuminance loop. Shadowing from the smoke and other objects can be computed using deep shadow maps [25], and dealt with within the illuminance loop.

The incandescent colour of the gas can be defined arbitrarily, using the same peicewise linear lookup table class as used to define the buoyancy curve. A good colour curve can be calculated in a similar manner to the buoyancy curve, using the same, or similar temperature values, and calculating the red, green and blue values using planck's black body radiation formula, as in [7]. The volume data is transferred to RenderMan using brickmaps, which are RenderMan's volumetric textures. The saved grid data file is first converted to a RenderMan point cloud file using a standalone program, and this is then converted to a brickmap using the "brickmake" facility.

## 5.1 Computing Self Shadowing

An important visual cue in smoke is how it absorbs light and casts shadows on itself - this is computed efficiently using deep shadow maps[25]. A deep shadow map, rather than containing a single z depth value per pixel as in a normal shadow map, stores a peicewise linear function approximating the visibility of

the light source as a function of depth. This shadow format is supported in PRMan, and a convenient API exists for reading and writing deep shadow files.

PRMan does not support automatic generation of deep shadows from volume shaders, so a standalone program was written using the deep texture API. The program takes in a previously rendered deep shadow map, which contains a record of the camera and projection matrices of the light. It then reconstructs the view volume and ray marches through the smoke, generating additional deep shadow data and merging it with the original shadow as in [25], writing out a new deep shadow map.

The first method of reconstructing the view volume was to directly invert the light's projection matrix using the standard method (adjoint matrix divided by determinant). However, this proved numerically unstable. The view volume consists of two rectangles connected by a segment of a pyramid - using this method, the angle between the sides of the back rectangle typically deviated from a right angle by about two degrees. When the near clip plane was extremely close to the camera, this deviation became much worse, making the view volume unusable. This instability was improved by using QR decomposition to invert the projection matrix. However, the errors still became serious when the near clip plane was very close to the camera. A possible improvement would be to factor the projection matrix into the light's camera matrix and a pure projection matrix, and reconstruct the view volume using this information instead.

## 5.2 Rendering Light from the Flames

To render light from the flames, another standalone program was used to approximate the incandescence of the fire with a cloud of point lights. A special Renderman shader was written for these point lights, which used category strings so as not to communicate with the volume shader (having the lights affect the smoke would have made rendering very expensive). A problem when approximating volume illumination with point lights is that objects inside the volume can develop bright spots, due to geometry being too close to any point lights which get generated. To address this issue, the usual $\frac{1}{r^2}$ attenuation law, which can grow arbitrarily large for small values of r, is modified to include a light radius term, c, becoming $\frac{1}{(r+c)^2}$. This limits the brightness of the light, and reduces the appearance of bright spots.

Two methods of generating the point lights were implemented. In the first method, the points on the fire data grid were grouped using K-means clustering [30], and a point light was exported for each group. The K-means clustering algorithm identifies clusters of points in a point cloud, where each cluster is defined as the set of points closest to one of a number of centroid points. It achieves this by giving the centroids initial positions, then finding the set of points closest to each centroid. For each set, it then finds the "center of mass" (in this case weighted by the luminescence of grid each data point), and moves the centroid to the center of mass. It then repeats this process until the total distance moved by the centroids per iteration is below a specified tolerance. This algorithm effectively identifies isolated regions of incandescence, and can

approximate a flame efficiently with a small number of light sources. Unfortunately, a given incandescence data set has multiple equilibria, and this can lead to popping artifacts in an animation.

The second method is based on the median cut algorithm proposed by Paul Debevec in [31]. This method builds a KD tree from the volume data, where the volume is recursively divided into two halves containing equal light energy along axis aligned planes. For each volume, the "center of mass" is found, as in the K means algorithm, and a light source is placed at this point, with a colour equal to the total brightness in the volume.

The aim of implementing this algorithm was to avoid popping, so the split axes are chosen prior to running the algorithm, rather than being decided based on the longest axis of the cell as in [31] - ie the algorithm splits the volume first on the x axis, the y axis then the z axis (or some user specified permutation of this) and then starts again. This way, all the positions of the light sources are functions of integrals over the domain, and therefore adding a small perturbation to the incandescence data will change the positions of the lights by a correspondingly small amount, meaning there are no discontinuities, which can lead to popping. In the implementation of the median cut algorithm, a summed volume table is first generated (a dataset in which each cell contains the sum of the values in all the cells with all their grid coordinates less than those of the cell), which is used to quickly find the median points of the cells.

A disadvantage of the median cut strategy is that, as it divides the volume into cells of equal energy, it may incorrectly place a light source if there is a large region of light emission, and a small but visually important region far away from it. If the ratio of the energy in the larger region to that in the smaller region is greater than the number of lights being used, it may group the smaller region with part of the larger region, and place a light source half way between the two. Despite this, the median cut strategy appears to generate better output than k-means clustering, particularly for animations, although there are problems with the k means clustering code which require work.

# 6 Results

This section shows stills from various tests animated and rendered using the work done on this project. The scenes were set up and rendered using the Houdini pipeline system described in Appendix A.

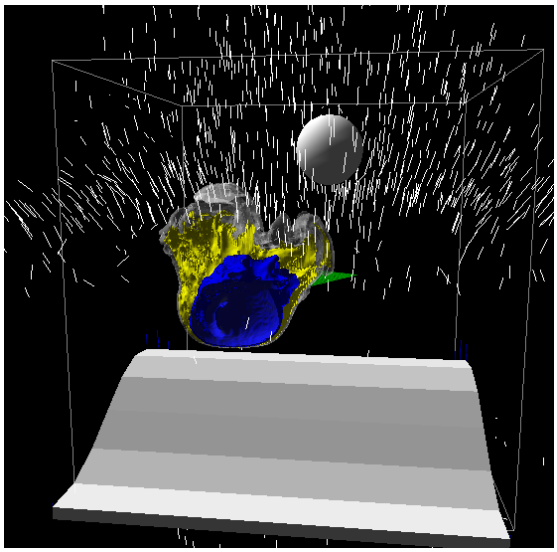

Figure 3: A screenshot of an early test , showing a burning teapot flying over an undulating plane and a sphere. The reaction front is shown in blue, yellow shows all fluid which crossed the reaction front less than half a second ago, and an isosurface of the smoke density field is shown in grey.

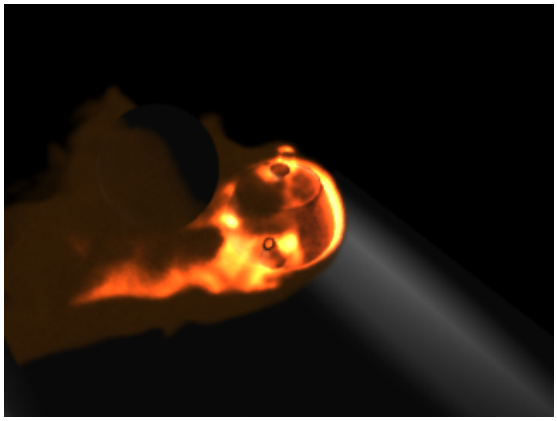

Figure 4: A still from the same scene as in figure 3, rendered using the smoke and fire volume shader in RenderMan. Deep shadows have not been implemented as yet.

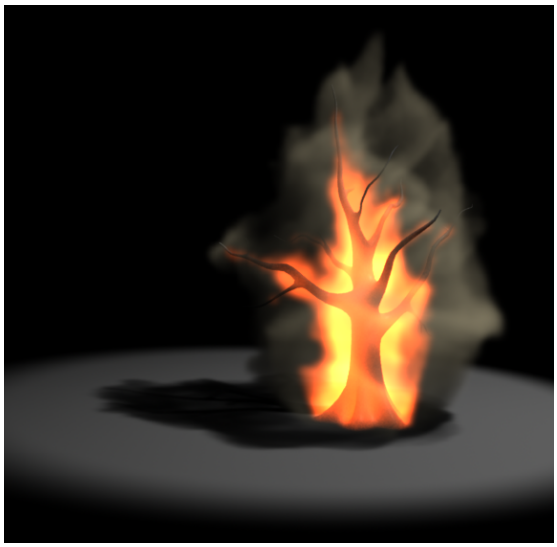

Figure 5: This image shows one of the first renders using deep shadows on the smoke, although a bug in the body force section of the simulator meant that the behavior of the smoke and fire was incorrect.

Figure 6: A still from an animation of a Bunsen burner, in which fuel is injected through a tube into the domain at high speed.

Figure 7: Another object emitting smoke and flames with deep shadows. Note the realistic rolling of the smoke due to a bug fix in the body force code.

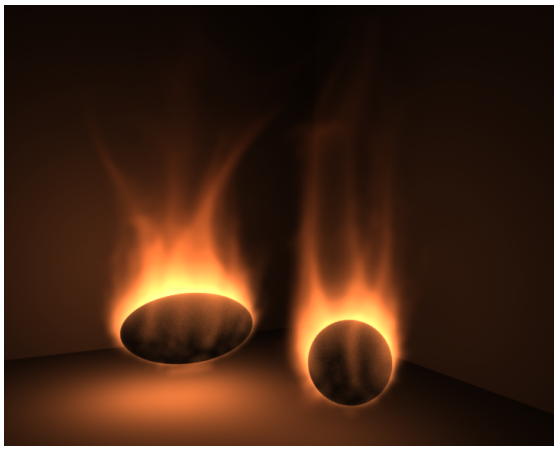

Figure 8: A still from the first test of the K means clustering algorithm for generating lights to illuminate the surrounding geometry.

Figure 9: This screenshot shows part of an animation of smoke interacting with an object. The colour curve of the flame has also been edited to give the flame a blue core, and the flame can be seen illuminating the surrounding geometry, using point lights generated using the K means clustering algorithm.

Figure 10: A screenshot of an animation with a high resolution fire simulation applied to it. Model by Ritchie Moore, animation by Malcolm Childs.

# 7 Current Problems

The simulator has been used to successfully create several animations, albeit for demonstrative rather than artistic purposes, and having been refined over the course of creating these animations, it allows for a good work flow. The user can preview the animation in OpenGL before rendering it properly, and can also preview the rendering to some extent while editing the colours. There are several problems which need fixing however.

The first problem is the boundary conditions for the edges of the domain. It has been observed that these boundaries emit unnatural winds if fluid motion occurs too close to them, presumably due to the fact that surfaces where the pressure is constant are rarely found in nature. These winds can cause the flames to take on strange shapes, and can be quite strong, so it could be worth researching different kinds of boundary conditions which reduce this effect.

Another problem, which sometimes surfaces when the geometry is bad, is the occasional appearance of cavities with narrow channels communicating with the rest of the fluid, which the solver handles badly. These can cause the solver to take a large number of iterations to converge, not converge at all, or in some cases diverge, giving the fluid large, arbitrary velocities and rendering a simulation useless. This problem can be addressed in part by adding code which detects and plugs channels which are a single cell wide prior to the flood fill - this only removes part of the problem though. It could be that situations such as this require more sophisticated solver algorithms, and it would also be worth improving the mesh voxelization code so it can handle bad geometry more robustly. For now though, the artist should try and avoid these situations in the geometry which is sent to the simulator, and make sure the polygon meshes are closed, and avoid self intersections in the surface as much as possible.

# 8 Conclusion and Future Work

The aim of this project was to develop usable tools for the rendering and physically based simulation of fire, and to generate visually pleasing output. I believe this objective has been fulfilled, and a potentially useful tool has been created. Over the course of this project, the author has become acquainted with a wide range of work in several fields of computer graphics and has acquired large body of knowledge about numerical simulation, computational linear algebra, and various algorithms which are of general use and applicability. A large amount of code has been written which will potentially be useful in future, and numerous problems have been encountered and overcome.

Creating a fluid simulator for computer animation is an enormous task, and like most software projects it is never finished. There are a large number of things which could be implemented in future to improve the system, some of which are:

- Attempting to extend the work of Elcott et al in [32] to work on cubic grids. This paper elegantly eliminates the need for vorticity confinement

by explicitly conserving the circulation of fluid round all loops moving with the fluid, a conservation law known as Kelvin's law. At the moment the algorithm only works on tetrahedral meshes, but higher resolutions would be achievable using cubic grids, because of their more efficient memory usage.

- Implement some kind of adaptivity in the simulation grid. This could include using an adaptively refined octree, as in [33], and various other additions like resizing the simulation grid when detailed motion occurs near the edges, implementing grids which can move, and allowing different simulation grids to be coupled together.

- Memory optimizations for the solver, for example using run length encoding to store the level set, which need not be defined everywhere, as in [22].

- Using adaptive ray marching to render the fire. Currently, fixed steps are taken along the rays, which can lead to problems when very thin, bright layers of fire are encountered - something which often happens when an object emits fire. A program could analyze the smoke and fire data and output an axillary data grid which informs the ray marcher of the maximum step it should take at any given point. This could greatly increase the quality and speed of rendering, at the cost of some extra storage space.

- Time restrictions have cut short investigations into the performance and behavior of the point light generation algorithms, and it would be good to continue with this. Also, it may be possible to write a more advanced light shader using a RenderMan shade-op plugin, using a hierarchical approach to integrate the illuminance over the fire domain.

- Advected textures to add detail to the smoke. With this technique, texture coordinates are stored and tracked on grids in the same way as with other quantities like smoke, and are used to reference volume textures or procedurally generated patterns when rendering. This is essential for depicting larger scale phenomena.

- Control particles and particle based fuel emitters for increased artistic control

## Thanks:

Thanks to Joao Montenegro, Johannes Saam, Gerard Keating, Ian Stephenson and John Macey for their help and suggestions, and to Malcolm Childs and the Cave Troll team for letting me use their animation.

# Appendix A

## Houdini Pipeline

Because of its elegant interface, excellent RenderMan exporter and configurability, a pipeline was built in Houdini, to test the simulator and generate output. It takes the form of a Houdini OTL, a package containing a number of custom nodes and scripts, which can be installed in a Houdini scene file.

The first thing that must be done before the pipeline can be used is to place all the necessary programs and renderman shaders in a single directory. This directory must contain the following executable:

```
DMFire
kmeanslight
medcutlight
pointconvert
smokeshadow
```

and the following RenderMan shaders:

```
volfire.slo
nosmokelight.slo
```

Houdini must now be informed of this directory by defining a variable called "DMFIREPATH". To do this, open up the Houdini text port, by pressing Alt+T on the keyboard, and type the following:

```
set DMFIREPATH = <path>
```

Where <path> is the path where the files have been placed. Now save a .hip file, and install OPfiredomain.otl by choosing "Install Operator Type Library" from the File menu.

You can now create a domain for the fluid simulation by going to obj level and creating a "DM fire domain" node. Clicking on the Controls tab in the parameter pane will reveal the interface shown in figure 11.

This allows you to set various simulation parameters, like the frame range to export, the dimensions of the domain and the size of the grid cells. You can also use the transform tab to position and rotate the domain[5]. Once these parameters have been set up, create some nodes, either DMFire Ellipsoid, DMFire Mesh or DMFire Dynamic Mesh, and put them in the simulation domain. The mesh objects have a "SOP" field in their "Controls" tab, which must be point to the polygon geometry you wish to define the object, and all nodes have a "burn" parameter, which indicates how much fuel the object emits. This uses the geometric normals of the object to define which direction to emit fuel in, so if a burning object appears to be sucking in air in the simulation rather than

[5]Do not adjust the scale of the transform, as this may cause the simulation to behave strangely.

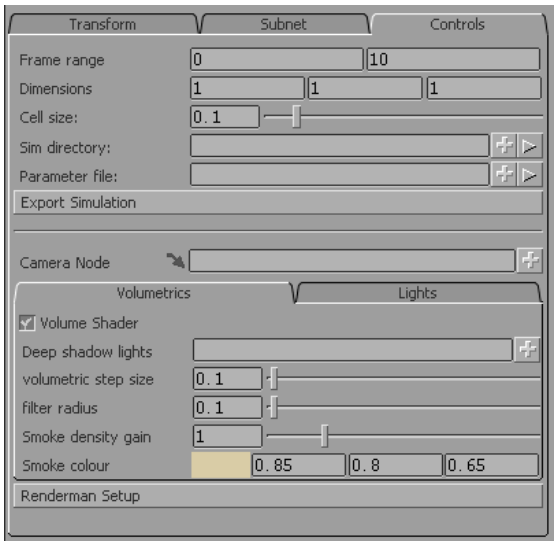


Figure 11: The Houdini OTL user interface

blowing fuel out, modify its geometry with a reverse SOP. Next create a new directory where all the files used by the simulator can go, and enter it into the "Sim directory" box. Finally select a parameter file, which is an XML file with the extension .lut, and defines properties of the flame like colour curves and the flame speed, and press the Export Simulation button.

This will export several files into the sim directory and launch the sim, a screen shot of which is shown in figure 12. The controls on the right can be used to interactively adjust the parameters, and the "Rendering" button brings up a render preview window, where the user can edit the colour curves and see them rendered using ray marching. The "Curves" button allows editing of the smoke and buoyancy curves. If you wish to save or load sets of parameters as .lut files, use the save and load options in the File menu.

Once the simulation has finished, wind the timeline back at the bottom of the 3D viewport and press the Play button to view the animation. The sim has saved the smoke and fire data grids for all the frames to the sim directory, and you can now go back into Houdini to render this data.

Scripts are defined in the OTL to set up render nodes to do this. First, you must specify a camera using the "Camera Node" box. The OTL can automatically set up deep shadows for you, and if you wish any light sources to cast deep shadows, enter them in the "Deep shadow lights" box.

The user can also use the "Lights" tab (figure 13) to generate lights to be included in the render (these lights are not visible in the Houdini viewport). To enable this, tick the "Rib Lights" box, and select the strategy and the parameters. The Light Gain parameter can be used to brighten or dim the lights, and the Downsampling parameter reduces the resolution of the grids before the

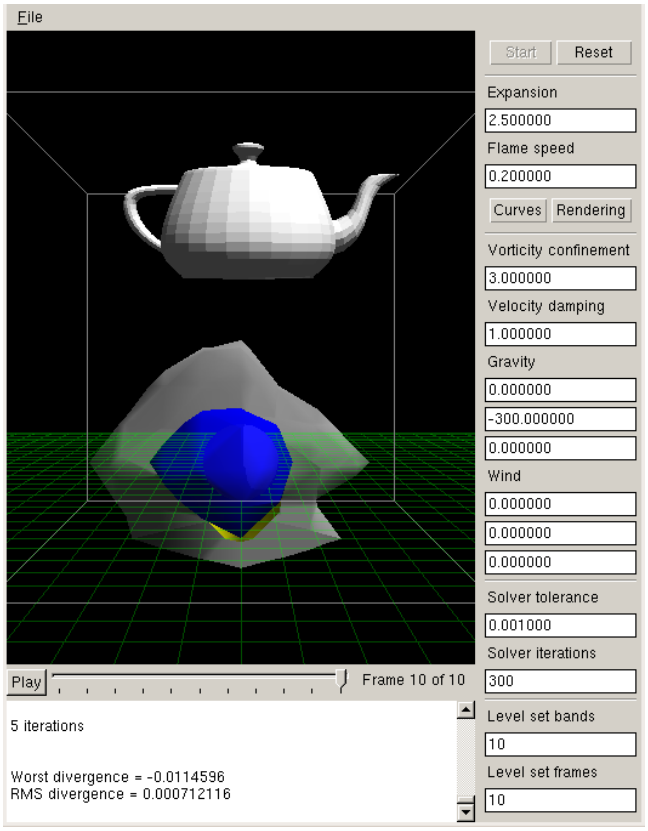


Figure 12: The simulator's user interface

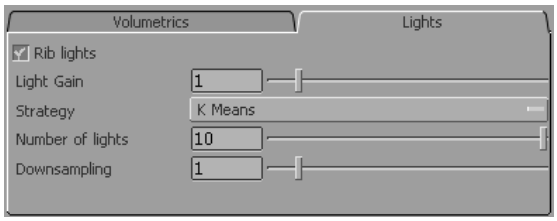


Figure 13: Lights tab in the Houdini user interface

algorithms are run, and can reduce processing time and memory consumption for large simulations. Finally, go to rop level in Houdini and press L with the mouse over on the network editor, to lay the render nodes out nicely. You can then go to the DMFire node which all other nodes are feeding into, and render your images.

# References

[1] Bridson et al, Fluid Simulation Siggraph course 2006

[2] Fedkiw, R., Aslam, T., Shaojie, X. 1999, The Ghost Fluid Method for Deflagration and Detonation Discontinuities

[3] Lamorlette, A., Foster, N. 2002, Structural Modeling of Flames for a Production Environment

[4] Sethian, J. 1995, A Fast Marching Level Set Method for Monotonically Advancing Fronts

[5] Enright, D., Fedkiw, R. 2003, Robust Treatment of Interfaces for Fluid Flows and Computer Graphics

[6] Foster, N., Fedkiw, R. 2001, Practical Animation of Liquids

[7] Nguyen, D., Fedkiw, R., Jensen, H. 2002, Physically Based Modeling and Animation of Fire

[8] Shewchuk, J. 1994, An Introduction to the Conjugate Gradient Method Without the Agonizing Pain

[9] Fedkiw et al 2006, Multiple Interacting Liquids

[10] http://en.wikipeda.org/wiki/Binary_heap, Binary Heap article

[11] Nguyen, D., Fedkiw, R., Kang, M. A. 2001, Boundary Condition Capturing Method for Incompressible Flame Discontinuities

[12] Liu, X., Fedkiw, R., Kang, M. 2000, A Boundary Condition Capturing Method for Poisson's Equation on Irregular Domains

[13] Hong, J., Shinar, T., Fedkiw, R. 2007, Wrinkled Flames an Cellular Patterns

[14] Feldman, B., O'Brien, J., Okan, A. 2003, Animating Suspended Particle Explosions

[15] Jensen, H., Buhler, J. 2005, A Rapid Hierarchical Rendering Technique for Translucent Materials

[16] Rasmussen, N., Nguyen, D., Weiger, W., Fedkiw, R. 2003, Smoke Simulation for Large Scale Phenomena

[17] Heinbockel, J., Introduction to Tensor Calculus and Continuum Mechanics

[18] Hirt, C. W. 1970, An Arbitrary Lagrangian-Eulerian Computing Technique

[19] Huang, J. et al. 1998, An Accurate Method for Voxelizing Polygon Meshes

[20] Houston, B., Bond, C., Wiebe, M. 2003, A Unified Approach for Modeling Complex Occlusions in Fluid Simulations

[21] Nooruddin, F. S., Turk, G. 2003, Simplification and Repair of Polygonal Models using Volumetric Techniques

[22] Houston, B. et al 2006, Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation

[23] Stam, J. 1999, Stable Fluids

[24] Marschner, S., Cornell University, 2004, Volume Rendering Equation (lecture notes)

[25] Lokovic, T., Veach, E. 2000, Deep Shadow Maps

[26] Foster, N., Metaxas, D. 1996, Realistic Animation of Liquids

[27] http://mathworld.wolfram.com/DivergenceTheorem.html

[28] http://mathworld.wolfram.com/StokesTheorem.html

[29] Harlow, F., Welch, G. 1965, Numerical Calculation of Time Dependent Viscous Incompressible Flow of Fluid with Free Surface

[30] http://people.revoledu.com/kardi/tutorial/kMean/Algorithm.htm

[31] Debevec, P. 2005, A Median Cut Algorithm for Light Probe Sampling

[32] Elcott, S., Tong, Y., Kanso, E. Schroder, P, Desbrun, M. 2007, Stable,circulation preserving, simplicial fluids

[33] Losasso et al., 2004, Simulating Water and Smoke with an Octree Data Structure