

Cowboy Programming Game Development and General Hacking by the Old West

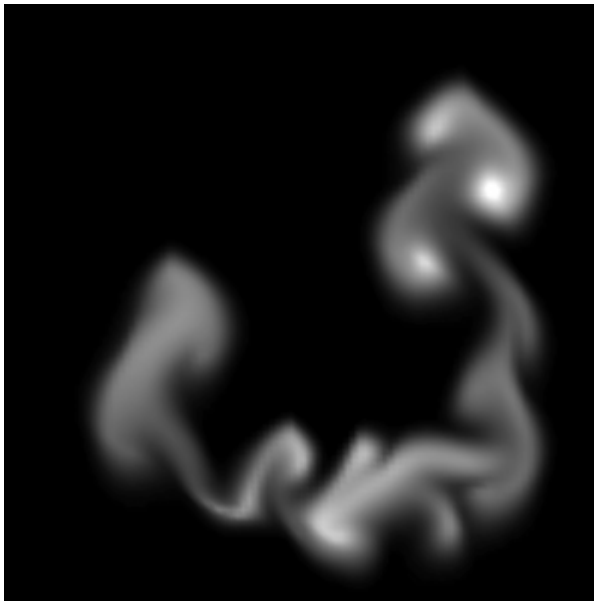
April 1, 2008

Practical Fluid Mechanics

Filed under: [Game Development](#), [Inner Product](#) — Mick West @ 1:57 pm

(This article originally appeared in two parts in Game Developer Magazine, March and April, 2007) Fluid effects such as rising smoke and turbulent water flow are everywhere in nature, but are seldom implemented convincingly in computer games. The simulation of fluids (which covers both liquids and gasses) is computationally very expensive. It is also mentally very expensive, with even introductory papers on the subject relying on the reader having math skills at least at the undergraduate calculus level.

In this article I will attempt to address both these problems from the perspective of a game programmer not necessarily conversant with vector calculus. I'll explain how certain fluid effects work without using advanced equations and without too much new terminology. I shall also describe one way of implementing the simulation of fluids in an efficient manner without the expensive iterative diffusion and projection steps found in other implementations. A working demonstration with source accompanies this article and can be [downloaded from here](#), example output from this can be seen in figure 1.



GRIDS OR PARTICLES?

There are several ways of simulating the motion of fluids. These generally divide into two common types of methods: *grids methods* and *particle methods*. In grid methods, the fluid is represented by dividing up the space a fluid might occupy into individual cells, and storing how much of the fluid is in each cell. In particle methods the fluid is modeled as a large number of particles that each move around and react to collision with the environment and interacting with nearby particles. Here I'm going to concentrate on simulating fluids with grids.

It is simplest to discuss the grid methods with respect to a regular two-dimensional grid, although the techniques apply equally well to three dimensions. At the simplest level, to simulate fluid in the space covered by a grid you need two grids, one to store the density of liquid or gas at each point, and another to store the velocity of the fluid. Figure 2 shows a representation of this, with each point having a velocity vector, and also containing a density value (not shown). The actual implementation of these grids in C/C++ is most efficiently done as one dimensional arrays. The amount of fluid in each cell is represented as a float. The velocity grid (also referred to as a velocity field, or vector field) could be represented as an array of 2D vectors, but for coding simplicity it is best represented as two separate arrays of floats, one for x and one for y.

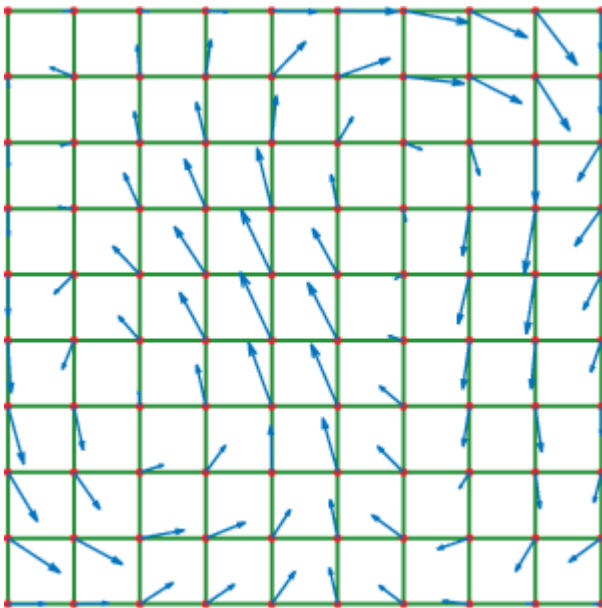


Figure 2 - A velocity field. Each point stores a velocity.

In addition to these two grids we can have any number of other matching grids that store various attributes. Again each will be stored as matching array of floats, and can store things such as the temperature of the fluid at each point, or the color of the fluid (whereby you can mix multiple fluids together). You can also store more esoteric quantities such as humidity, for if you were simulating steam or cloud formation.

ADVECTION

The fundamental operation in grid based fluid dynamics is **advection**. Advection is basically moving things around on the grid, but more specifically it's moving the quantities stored in one array by the movement vectors stored in the velocity arrays. It's quite simple to understand what is going on here if you think of each point on the grid as being an individual particle, with some attribute (the density) and a velocity.

You are probably familiar with the process of moving a particle by adding the velocity vector to the position vector. On the grid, however, the possible positions are fixed, so all we can do is move (advect) the quantity (the density) from one grid point to another. In addition to advecting the density value, we also need to advect all the other quantities associated with the point. This would obviously include additional attributes such as temperature and color, but also includes the velocity of the point itself. The process of moving a velocity field over itself is referred to as **self-advection**.

The grid does *not* represent a series of discrete quantities, density or otherwise, it actually represents (inaccurately) a smooth surface, with the grid points just being sampled points on that surface. Think of the points as being X,Y vertices of a 3D surface, with the density field being the Z height. Thus you can pick any X and Y position on the mesh, and find the Z value at that point by interpolating between the closest four points. Similarly while advecting a value across the grid the destination point will not fall directly on a grid point, and you will have to interpolate your value into the four grid points closest to the target position.

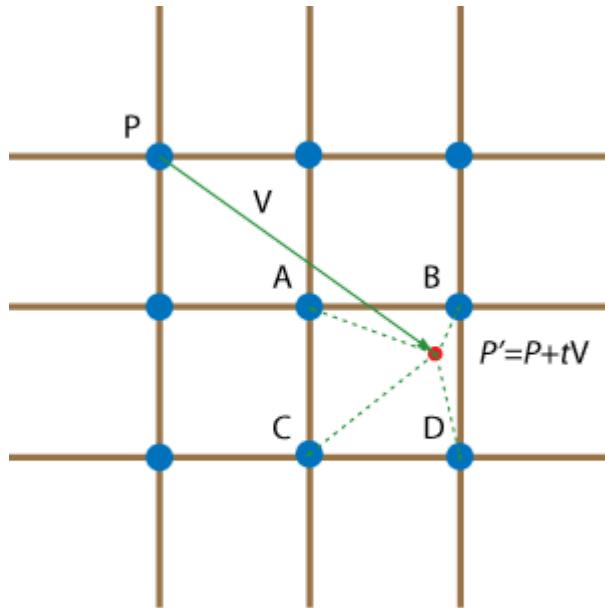


Figure 3 - Advecting Forwards, the value at P moves by vector V, and affects A,B,C & D

In figure 3, the point at **P** has a velocity **V**, which, after a time step of **dt**, will put it in position **P' = P + dt*V**. This point falls between the points A, B, C and D, and so a bit of **P** has to go into each of them. Generally **dt*V** will be significantly smaller than the width of a cell, so one of the points A,B,C or D will be P itself. Advecting the entire grid like this suffers from various inaccuracies, particularly that quantities dissipate when moving in a non-axis-aligned direction. This inaccuracy can actually be turned to our advantage.

STAM'S ADVECTION

Programmers looking into grid based fluid dynamics for the first time will most often come across the work of Jos Stam and Ron Fedkiw, particularly Stam's paper "[Real-Time Fluid Dynamics for Games](#)", presented at the 2003 Game Developer Conference. In this paper Stam presents a very short implementation of a grid based fluid simulator. In particular he describes implementing the advection step using what he terms a "linear backtrace", which simply means instead of moving the point forward in space, we invert the velocity and find the source point in the opposite direction, essentially back in time. We then take the interpolated density value from that source (which, again, will lay between four actual grid points), and then move this value into the point P. See figure 4.

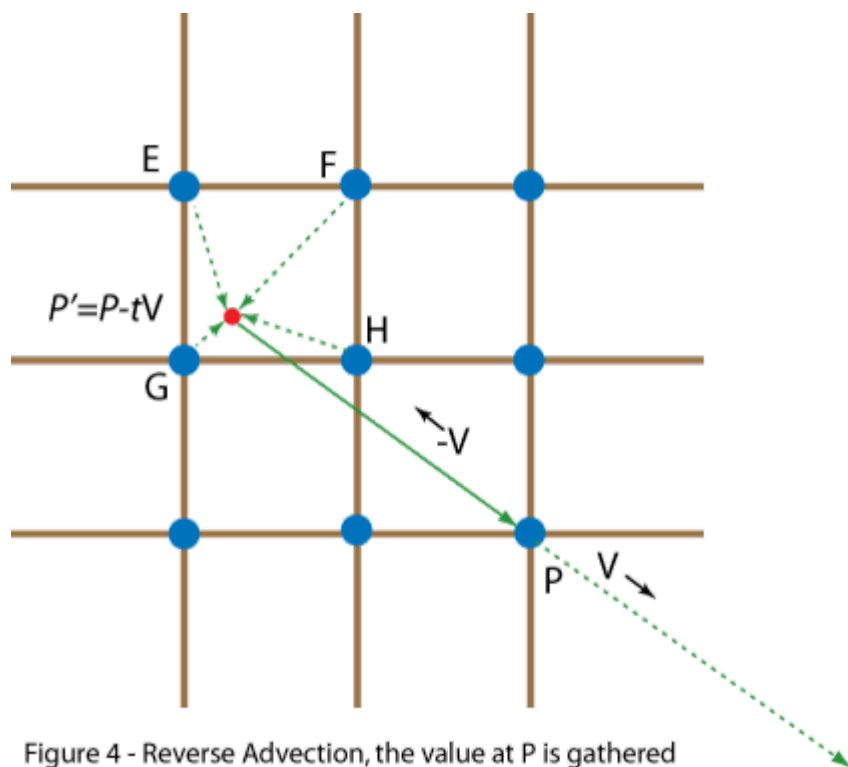


Figure 4 - Reverse Advection, the value at P is gathered from the points E,F,G&H, indicated by -V

Stam's approach produces visually pleasing results, yet suffers from a number of problems. Firstly the specific collection of techniques discussed may be covered by U.S. patent #6,266,071, although as Stam notes, the approach of backtracing dates back to 1952. Check with your lawyer if this is a concern to you. On a more practical note **the advection alone as described by Stam simply does not work accurately unless the velocity field is smooth in a way termed mass conserving, or incompressible.**

Consider the case of a vector field where all the velocities are zero except for one. In this situation the velocity cannot move (advection) forward through the field, since there is nothing ahead of it to "pull" it forward, instead the velocity simply bleeds backwards. **The resultant velocity field will terminate at the original point, and any quantities moving through this field will end up there. This problem is solved by adding a step to the algorithm termed projection, which is basically smoothes out the velocity by making it incompressible, thus allowing the backtracing advection to work perfectly, and making the paths formed by the velocity be "swirly", as would be the case in real water.**

The problem with this approach is that projection is quite expensive, requiring 20 iterations over the velocity field in order to “relax” it to a usable state. Another performance problem with Stam’s approach is that there is a **diffusion** step, which also involves 20 iterations over a field. This is needed to allow the gas to spread out from areas of high density to areas of low density. If the diffusion step were missing, solid blocks of the fluid would remain solid as they moved over the velocity field. The diffusion is an important cosmetic step in the process.

ACCOUNTING ADVECTION

If a velocity field is not mass conserving, then this means that some points will have multiple velocity vectors from other points pointing towards them. This means that if we simply move our scalar quantities (like density) along these vectors, then there will be multiple quantities going to (or coming from) the same point, and the result will be a net loss or gain of the scalar quantity. So if the total amount of something such as the density would either fade to zero or gradually (or perhaps explosively) increase.

The usual solution to this problem is to make sure the vector field is incompressible and mass conserving. But as mentioned before, this is computationally expensive. One partial solution is to make the advection step mass conserving, regardless of if the velocity field actually is mass conserving. The basis of this solution is to **always account for any movement of a quantity by subtracting in one place what is added in another.** **Advection uses a source and destination buffer to keep it independent of update order.** In Stam’s implementation, the destination buffer is simply filled one cell at a time by combining a value from four cells in the source buffer, and placing this value into the destination buffer.

To properly account for compressible motion, we need to change this **copying** to **accumulating**, and initially make the destination buffer a copy of the source buffer, and as we move quantities from one place to another we can subtract them in the source and add them in the destination. With the **forward advection** in figure 3, we are moving a quantity from point P to points A,B,C and D. To account for this we simply subtract the original source value in P from the destination value in P, and then add it (interpolated appropriately), to A,B,C,D. The net change on the destination buffer is zero. With the **reverse advection** in figure 4, as used by Stam, the solution would initially seem to be symmetrically the same: just subtract the interpolated source values in E,F,G and H from the destination buffer, and add them to P.

While this works fine for signed quantities such as velocity, the problem here is that quantities such as density are positive values. They cannot go below zero as you cannot have a negative quantity of a liquid. Suppose that point E was one source point for two destinations P1 and P2, both of which wanted 0.8 of E. Now, if we follow our initial plan and subtract $0.8 * E$ from E and add $0.8 * E$ to both P1 and P2, the net effect is zero, but now the value at E is negative. If we clamp E to zero then there is a net gain of $0.6 * E$. If we subtract $0.8 * E$ from the source value of E after updating P1, then when we update P2 it will only get $0.8 * 0.2 * E$, when clearly both P1 and P2 should both get equal amounts, and intuitively here it seems they should both get $0.5 * E$, and the resulting value in E should be zero, leading to a net zero change.

To achieve this result I first create a list that for each point records the four points that are sources for that point, and the fraction of each point they want. Simultaneously I accumulate the fractions asked of each source point. In an ideal world, this would add up to one, as the entire value is being moved somewhere (including partially back where it started). But with our compressible field the amount of the value in each point that is being moved can be greater than or less than one. If the total fraction required is greater than one, then we can simply scale all the requested fraction by this value, which means the total will be one. If less than one, then the requesting points can have the full amount requested. We should not scale in this case as it will lead to significant errors.

With the mass conservation of advection fully accounted for in both directions, it turns out that neither forward or backward linear advection alone will produce smooth results. After some experimentation I determined that applying forward advection followed by backward advection worked very

well, and give a very smooth and artifact free flow of fluid over a compressible velocity field.

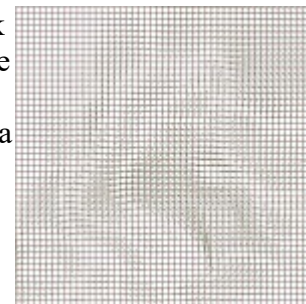
NOW WHAT?

So, we can now perform both forward and reverse advection in a mass-conserving manner, meaning we can move fluid around its own velocity field. But even though our velocity field does not *need* to be mass-conserving, we actually still want it to be, since the velocity fields of real world fluids generally are incompressible. Stam solves this problem by **expensively forcing the field to be fully mass conserving after every change**. This is necessary, since the **reverse advection requires it**. The key difference now is that since our advection step does not require the field to be mass-conserving, we are really only doing it for cosmetic purposes. To that end, any method that rapidly approaches that state over several time-steps will suit our purpose. That method, and the method of diffusion, can be found in the accompanying code, and are discussed below.

PRACTICAL FLUID DYNAMICS: PART 2

In last month's article (above) I gave an overview of the nuts and bolts behind simple two dimensional fluid dynamics using a grid system. This month I'll expand upon this, explaining how we can achieve a reasonable level of realism without too many expensive iterations. I'll also continue with my goal of explaining how everything works by using no math beyond basic algebra.

To recap so far: we have a **velocity field** which is an array of cells, each of which stores the velocity at a particular point (click example on the right). Remember this a continuous field, and we can get the velocity at any point on the field surface (or in the field volume for 3D), by interpolating between the nearest points on the field. We also have a matching field of **density**. The density field represents how much of the fluid or gas is in a particular grid cell. Again this a continuous field, and you can get a density value for any point in the simulated space by interpolating. I then described the process of **advection**, which is the moving of the values in one field (say the density field), over the velocity field.



I described both **forward advection** and **reverse advection**, where the quantities in the field are respectively pushed out of a cell, or pulled into a cell by the velocity at that cell. I noted that the advection process worked well if you perform forward advection and then follow it with reverse advection.

INCOMPRESSIBLE FIELDS

I noted that reverse advection in particular would only work if the velocity field was in a state termed **incompressible**. But what does this mean? Well, you might have heard that “water is incompressible”, meaning you can't squeeze water into a smaller volume than it already occupies. Compare this with gasses such as air, where you can clearly be compressed. Picture, for example, a diver's air tank. The tank contains a lot more air than the volume occupied by the tank. But if you were to take that tank, and fill it with water, and then somehow push in another pint of water, then the tank would explode.

Water, in fact, is actually compressible, very slightly, since it's physically impossible to have a truly incompressible form of matter. The incompressibility of a material is measured by a metric called a “bulk modulus”, For air this is about 142,000 whereas for water, it's 2,200,000,000 or

approximately 15,000 times as much. By comparison, the least compressible substance known to humankind, aggregated diamond nanorods, are just 500 times more incompressible than water.

So for most practical purposes, you can imagine water as being incompressible. So, with water being considered incompressible, then when considering a solid volume of water, there can not be more water in one cell than in another. So, if we start out with an equal amount of water in each cell, then after moving the water along the velocity field (advecting), we can't increase or decrease the amount of water in each cell. If this happens, then the velocity field is incompressible or **mass conserving**.

PRESSURE

You can think of the **pressure** at a particular node as being the difference in density between a cell and its neighbors. Now with water being incompressible, the pressure is going to be the same throughout the density field. If we think of a node as having a series of inputs and outputs during the advection process, then in an incompressible field, the sum of input is equal to the sum of outputs (Figure 5a). When we move the water along its incompressible velocity field, then the density at each node will remain constant, and hence the pressure will remain constant.

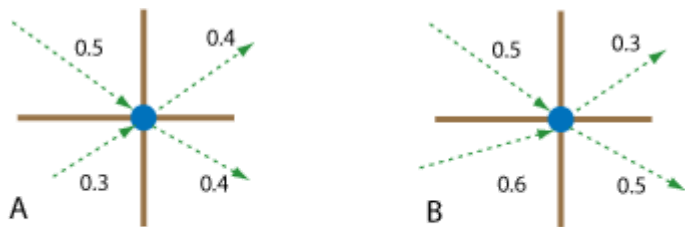


Figure 5 – A simplified view of the inputs and output of a node in the velocity field. A is balanced, but in B the density will continually accumulate in this node

On the other hand, if the velocity field happens to be structured in such a way that for some cells more is going into them than is coming out, then the velocity field is compressible (Figure 5b). When the density of the fluid is advected across a compressible velocity field, then the density in individual cells will increase or decrease. If we simply keep advecting the density, then the density will eventually all be compressed into the cells of the velocity field that have a net gain of input over output.

If we were not performing accounting in our advection step (as explained last month), then there would be an overall net loss in density (the field is not mass conserving). Stepping back from our abstraction for a second, what prevents this from happening in real-life? Well, obviously if more of a fluid flows into a cell than is flowing out, then the density of that cell increases relative to its neighbors, and hence the pressure in that cell increases. High pressure in a cell creates an acceleration force on the neighboring cells, increasing their velocity away from that cell, hence increasing the outflow rate from the cell, and evening out the imbalance. As with the atmosphere, fluid flows from an area of high pressure to an area of low pressure.

APPLYING PRESSURE

Listing 1 shows the code for applying pressure. Here **mp_p0** is the array that stores the density (which is equivalent to the pressure, so I actually refer to it as pressure in the code). The arrays **mp_xv1** and **mp_yv1** store the x and y components of the velocity field. The function **Cell(x,y)** returns a cell index for a given set of x and y coordinates. The loop simply iterates over all horizontal and vertical pairs of cells, finds the difference in pressure, scales it by a constant (also scaled by time) and adds it to both cells.

The logic here is slightly unintuitive, since physics programmers are used to the Newtonian principle that every action has an equal and opposite reaction, yet here when we add a force, there is no opposing force, and we don't subtract a force from anywhere else. The reason is clear if you consider what is actually going on. We are not dealing with Newtonian mechanics. The force actually comes from the kinetic energy of the molecules of the fluid which are actually randomly traveling in all directions (assuming the fluid is above absolute zero), and the change to the velocity field actually happens evenly across the gradient between the two points, so in effect we are applying the resultant force from a pressure gradient to the area it covers, which here is two cells, and we divide it between them.

Here's an example, just looking in the x direction, we have a flat pressure field, with one cell denser than the rest. The cell densities are 4,4,5,4,4. The gradients between the four pairs of cells is 0,-1,+1,0. Adding this to each cell (ignoring scaling), we get: 0,-1,0,+1,0. See Figure 6.

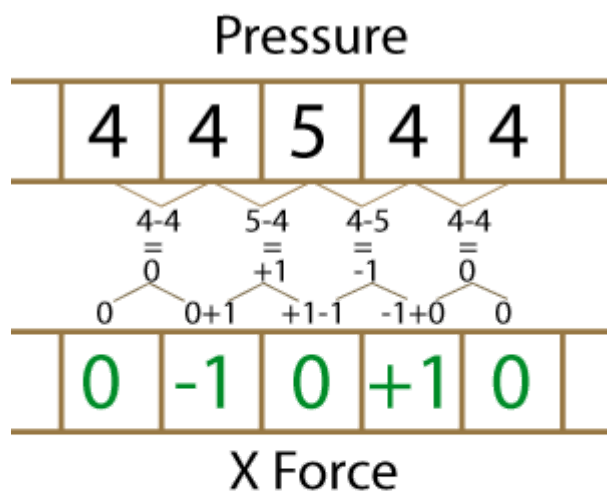


Figure 6 – Pressure generates a directional force in the X direction, which is then scaled and applied to the X velocity field. When done to the Y velocity field as well this gives a reasonable representation of reality.

Here the cells on either side of the high pressure cell end up with a velocity pointing away from that cell. Consider now what will happen with the advection step, the reverse advection combined with forward advection will move the high pressure cell outwards, reducing the pressure, and reducing the force. The fluid moves from an area of high pressure to an area of low pressure.

Effectively this makes the velocity field tend towards being incompressible and mass conserving. If there is a region that is increasing in density, then the resultant increase in pressure will turn the velocity field away from that area, and hence decrease the density in that area. Eventually the velocity field will either become mass conserving (mass just circulating without density change), or it will stop (become zero).

Listing 1 – The pressure differential between two cells creates an identical force on both cells

```
for (int x = 0; x < m_w-1; x++) {  
  for (int y = 0; y < m_h-1; y++) {  
    int cell = Cell(x,y);  
    float force_x = mp_p0[cell] - mp_p0[cell+1];  
    float force_y = mp_p0[cell] - mp_p0[cell+m_w];  
    mp_xv1[cell] += a * force_x;  
    mp_xv1[cell+1] += a * force_x;  
    mp_yv1[cell] += a * force_y;  
    mp_yv1[cell+m_w] += a * force_y;  
  }  
}
```

INK AND SMOKE

What we are modeling here is motion *within* a fluid (such as air swirling around inside a room), and not the overall motion of a volume of water, (such as water sloshing around a cup. This method, as it stands, does not simulate the surface of the fluid. As such, visualizing the fluid itself is not very interesting, since a room full of air looks pretty much the same regardless of how the air is moving. Where it become interesting is when we introduce some substance into the fluid that is suspended by that fluid, and carried around by the fluid.

In water this could be silt, sand ink, or bubbles. In air, it could be dust, steam, or smoke. You can even use the velocity field techniques outlined here to move larger object such as leaves or paper in a highly realistic manner. Note it's important that what we are talking about is a suspension of one substance in another. We are generally not so interested in simulating two fluids that do not mix (like oil and water).

Games have things that burn and explode, so smoke is a very common graphical effect. Smoke is not a gas, but a suspension of tiny particles in the air. These tiny particles are carried around by the air, and they comprise a very small percentage of the volume occupied by the air. So we do not need to be concerned about smoke displacing air.

In order to simulate smoke, we simply add another advected density field matching, where the value at each cell represents the density of smoke in that region. In the code this is referred to as “ink”. This is similar to the density of air, except the density of smoke or ink is more of a purely visual thing, and does not affect the velocity field.

HEATING THINGS UP

One final ingredient that often goes along with a fluid system like this is the *heat* of the fluid/gas at each location. Sources of smoke are usually hot, which heats up the air the smoke initially occupies. This causes the smoke to rise. It rises because higher temperatures mean more energy, which

means the fluid molecules are moving faster, which means higher pressure, which means lower density (remember density is only proportional to pressure at constant temperature), which makes the hot air rise.

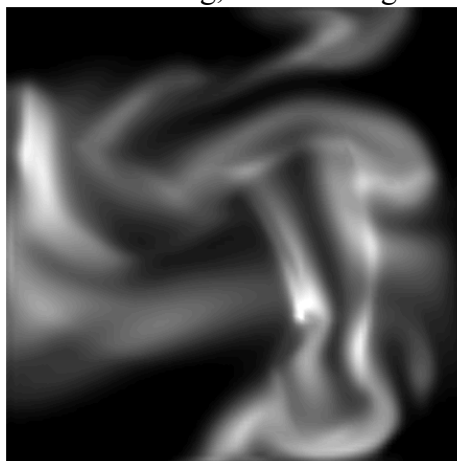
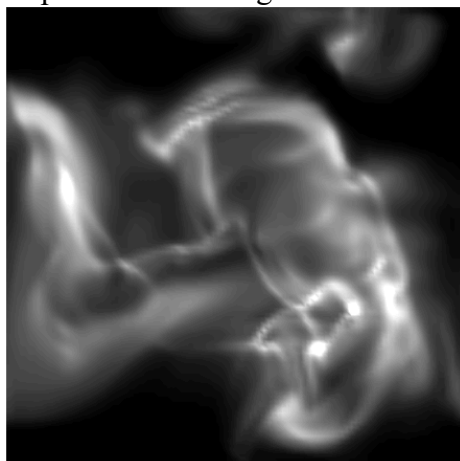
Now, that's a complex sequence of events, but it's initially simpler to just model the result, "hot air rises", and have the relative temperature of a cell create a proportionate upwards force on the velocity field at that cell. We can do this trivially by adding a scaled copy of the heat field to the Y velocity field. Similarly, rather than attempt to model the effects of heat in the initial phases of a simulation, I found it easier to simply model the expected results.

So, although a flame creates heat which makes the smoke rise, more pleasing results were found by "manually" giving the volume of air around the flame an initial upwards velocity, and then letting the heat take it from there. With more complex systems such as an explosion, the fiddly physics happens in the first tenth of a second, so you can just skip over that and set up something that looks visually pleasing with our simplified physics.

FILTERING THINGS OUT

The simplistic application of forces we perform for acceleration due to pressure (Figure 2) has the tendency to introduce artifacts into the system. These typically present as unnatural looking ripples. The way these are dealt with is to smooth out the velocity and pressure fields by applying a simple diffusion filter. If you use the Stam style reverse advection with projection, then you have to use a computationally intensive filter iterating several time. But with the inherent diffusion of forward advection, combined with the accuracy of the combined forward and backwards accounted advection, we can get away with a single iteration.

It's often difficult to see exactly what effect a change can have on a fluid system such as this. The fluid is very complex looking, and small changes to parameters often have an effect that is not immediately obvious. The ideal way to solve this problem is to set up your system so you can run two copies of the same system in parallel, with one having the modified parameters. The difference can then become obvious. Figure 7 (below) shows such an A/B comparison. The image on the left has no diffusion filtering, and the image on the right has a single pass of diffusion filtering applied



every update.

FLUID IDEAS

I've glossed over a few other important aspects here, but details of these aspects can be found in the accompanying code. You need to pay particular attention to how you handle the cells that are at the edge of the system, as the differing number of neighbors has a significant effect. At the edges of a system you have the option of either reflecting, wrapping or zeroing values, depending on what you want. By wrapping in one direction you essentially get a tiling animated texture in that direction, which could be used as a diffusion or displacement map for the surface of a moving stream of fluid.

There is also the issue of friction. Motion in a fluid is generally quite viscous. This can be implemented as a simple friction force that applies to the velocity field. If there is no friction in the fluid it can slosh around forever, which is generally not what you want. Different viscosity settings give very different visual results. There are a very large number of variables that can be tweaked to give radically different visual effects, even in my rather simple implementation. It's worthwhile spending some time just playing with these values just to see what happens.

Additional resources

- <http://www.upl.cs.wisc.edu/~aselle/fluid/>
- <http://graphics.stanford.edu/~fedkiw/>
- <http://www.dgp.toronto.edu/people/stam/stam.html>
- <http://www.cis.upenn.edu/~fostern/>

3D Version

This approach has also been implemented in 3D by Quentin Froemke, et al, at Intel, as part of their research into Multi Threaded programming.

http://www.gamasutra.com/view/feature/4022/sponsored_feature_multithreaded_.php
[Comments \(11\)](#)

11 Comments »

1. [...] Mick West has been kind enough to upload some code and accompanying Game Developer articles for a Stam solver he wrote. Very [...]

Pingback by [Deadrock » Fluid dynamics](#) — April 1, 2008 @ 2:41 pm



2. awesome!

Comment by [raigan](#) — April 2, 2008 @ 5:58 am

3. Tutorial de simulaci3n de mec3nica de fluidos...