Dan's Morris's Notes on Stable Fluids (Jos Stam, SIGGRAPH 1999)

This is intended to be a detailed by fairly-low-math explanation of Stam's "Stable Fluids", one of the key papers in a recent series of advances in simulating fluids for computer graphics.

Navier-Stokes Overview

The goal of this whole field is to simulate realistic fluid motion; this paper is represents one step forward in that direction. So let's first talk about the basic Navier-Stokes equations, which are an accepted physical description of the things that affect fluid motion. These equations represent "the truth", which 10 years of research have attempted to translate into algorithms. These formulations are taken from Stam's paper, although there are about one zillion different formulations of the Navier-Stokes equations.

$$\nabla \cdot \mathbf{u} = 0 \qquad (1)$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \qquad (2)$$

These equations describe the behavior of fluid at one point in a fluid volume. **u** is the vector-valued velocity at that point, which is what we're trying to solve for over and over. t is time. ρ is density, p is pressure, υ is viscosity, and **f** is external force; these four values are basically assumed to be handed down from space, i.e. whoever is *using* this algorithm sets these values to model a particular fluid or a particular environment. Also, for right now, we assume a constant density, which is reasonable for an incompressible fluid like water.

- Equation (1) says that at any point, the velocity in and out of the point must sum to zero. For a constant density, this is the same as saying that *mass* is conserved, which seems important, since we're not simulating nuclear explosions or magic, so mass can't appear or disappear.

- Equation (2) lists the factors that govern the movement (partial derivative) of velocity:

    - The first term says that velocity moves along itself, and it moves faster at regions of high velocity. This is a known property of fluids; it's the reason that swirling regions of smoke or water continue to swirl. Because **u** appears twice in this term, this whole equation is non-linear. This is the *advection* term.

    - The second term says that velocity moves along a pressure gradient. This term will actually disappear later on because it creates divergence (velocity would always flow *out* of high-pressure cells), which is illegal in our constant-density universe, so Stam in fact omits this term entirely in his presentations.

    - The third term says that velocity tends to *diffuse* along the velocity gradient. The more velocity diffuses, the smoother the velocity field gets. How would you describe a fluid with a very smooth velocity field? It would be hella viscous. Hence *v* is viscosity. This term is the *diffusion* term.

- The last term allows *external forces*. Since we assume a constant density, we call this term "force". The left side of this equation is basically acceleration (derivative of velocity), and we know that all equations in physics have to look basically like f=ma, which in this case works because m (density in our case) is assumed to be constant.

Now for completeness, I'm going to bring in an equation that's not shown explicitly in the Stam paper. Very often density is not constant (e.g. in any smoke simulation), and we want density to move around too. There is a new equation that we add to our system to simulate density movement, and it looks so much like equation (2) that I won't really even explain it. Stam adds this equation in his presentations, and basically any implementation of this system uses this equation.

(3) $$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Using what we've already discussed, this equation says that (a) density moves along the velocity field, (b) density tends to diffuse according to some constant κ, and (c) the user can add or remove density anywhere he wants (S).

Solving Navier-Stokes

Okay, now we have useful equations that describe how fluids behave. Now it's time to write my fluid simulation code…

```
void main {
    float time=0.0;
    while(1) {
        // oh shit… I can't paste the equations into my code…
        time += delta;
    }
}
```

Now we need a way to actually *solve* these equations, i.e. if the velocity and density look like some particular set of values at time *t*, what will they look like at the next timestep? This is the one-line summary of the entire field of computational fluid dynamics.

- In 1996, Foster and Metaxas started the modern wave of research in fluid simulation for graphics. They showed everyone the Navier-Stokes equations, and presented their approach to solving them:
    - They discretized the fluid volume into cubes, i.e. they use an Eulerian (fixed) grid for their simulation. Particles don't move around… there is a fixed grid, and the velocity

of the fluid at each grid cell changes over time. This seems standard now, but if you asked me to write a fluid simulation before reading any of this literature, I would have headed straight for the particles and written a Lagrangian simulation. So this is a big deal.

- They use explicit timestepping; i.e. at each timestep, they grab density and velocity from each cell and move it along the velocity field to neighboring cells.

- Sometimes this violates incompressibility, so to enforce incompressibility, they iteratively move tiny bits of mass around neighboring cells until everyone has the same mass that he started with. This is a "relaxation" technique.

The Foster and Metaxas system works nicely, but the explicit timestepping scheme leads to instability (long timesteps make the velocity field go crazy) and prevents interactivity, and the relaxation technique is slow and somewhat inaccurate.

Stable Fluids

Enter Jos Stam, 1999. Stam takes the F/M approach and moves it forward tremendously, presenting a semi-Lagrangian/implicit method for solving the Navier-Stokes equations that is unconditionally stable for any timestep. In this paper he focuses on moving *velocity* around (not density), but I'll discuss density at the end of the summary, because it's much simpler than moving velocity around, and he covers it in other places online.

- He starts with a very clean summary of the N-S equations, which I've paraphrased above

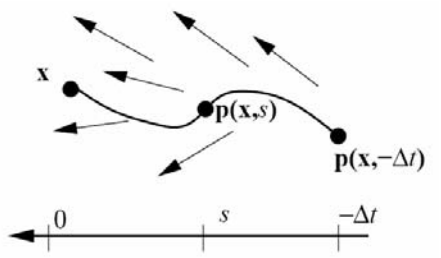
- Now he presents a magic tidbit of math that he'll use later: the Helmholtz-Hodge decomposition. This says that every vector field can be decomposed into a gradient field (of a scalar function) and a *divergence-free* vector field. I'll treat this math as a black box. As you might expect, he'll use this property to make illegal divergence disappear, instead of the relaxation scheme used by Foster and Metaxas.

- Next he presents a very clear set of steps that one needs to do to solve the N-S equations (which, as a reminder, means "update the velocity field"). The steps here should remind you of the individual terms in the Navier-Stokes equations. Note that we'll re-use most of these steps when we talk about moving density around, but for now we're talking about velocity.

    - *Add external force*: just add $\Delta t * \mathbf{f}(x,t)$ to each cell x where **f** is an external force

    - *Advect*: move the velocity field along itself
        - Foster and Metaxas did this using explicit simulation… if I'm a grid cell and my neighbor has some velocity that points toward me, take some of his velocity. Do this for all of my neighbors.

- As I mentioned earlier, this makes the velocity field freak out for large timesteps. So Stam's biggest contribution is an unconditionally-stable, semi-implicit advection scheme. He uses the "method of characteristics", which basically says that if I want to know the velocity at point x, I can trace a line backwards along the current velocity field at point x to see what my new velocity should be.
- More specifically, we imagine we have a particle sitting at point x (the center of a grid cell), and whatever the velocity **u** is at that cell, we move our imaginary particle by -**u**Δt. I.e., it walks backwards along the velocity field. It won't necessarily land in the middle of a grid cell, so wherever it lands, we grab the nearest velocity values and interpolate. This is the new velocity at cell x. Very stable.
- Because we sort of used a particle, we call this a semi-Lagrangian scheme. Stam's figure 2 illustrates this process… note that the particle is created at grid cell x, and moved *backwards* along the velocity field to get the corresponding value at "Δt seconds ago".



- *Diffuse*: let the velocity field spread out to represent viscosity
  - You could just diffuse any scalar field by taking each pair of neighbors and moving "stuff" (in this case velocity) from the higher-valued neighbor to the lower-valued neighbor. This would be a simple explicit scheme. Unfortunately it's really sensitive to grid size and timestep, so it's really unstable.
  - What you really want is to look at the whole grid at once and find the new velocity field that best represents diffusion according to our diffusion constant. So Stam uses an implicit technique, i.e. he asks the question: what velocity field would give us the current velocity field if we ran the diffusion process *backwards*?
  - This is a classic implicit system, and it leads to a linear equation, which they plug into a black-box linear system solver (choose your favorite from the web), which gives them a new, nicely-diffused velocity field.

- *Project*: get rid of all the divergence
  - Remember we want to enforce conservation of mass, but the previous steps didn't necessarily do this; we could have created or lost a little mass at individual cells. This is where Stam uses the magic Helmholtz decomposition, which basically finds the best divergence-free field to match a vector field, and lets you throw out the leftover junk.

        - This decomposition yields a Poisson equation (a second-order PDE), which they also feed to their favorite black-box solver (FISHPAK) to get a divergence-free field

- These steps let you update the velocity field given the previous velocity field. Now what about density? The density equation (3) is virtually identical to its velocity-oriented cousin (2). The first term depends on velocity, which we have now computed. So we use nearly the same steps for moving density around:

        - *Add source:* if the user wants more density at some places, do it

        - *Convect*: move the density field along the velocity field
            - The same semi-Lagrangian method works great here

        - *Diffuse*: spread the density out according to a user-defined diffusion constant
            - The same implicit method works great here

        - No need to project, since our grid of density is a scalar field, not a vector field

Now, we can put all this together to fill in our pseudocode:

```
void main {

    // velocity field (vectors)
    float v[SIZE][SIZE][3];
    // density field (scalars)
    Float d[SIZE][SIZE];
    float time=0.0;

    while(1) {

        // solve for velocity
        addExternalForces(v);
        advect(v);              // uses particle-tracing technique
        diffuse(v);             // uses linear system solver
        project(v);             // uses linear system solver

        // solve for density
        addExternalSources(d);
        convect(d);             // uses particle-tracing technique
        diffuse(d);             // uses linear system solver

        // do whatever I want to do with my fluids, e.g. rendering them...

        time += delta;
    }
}
```

- These are the main contributions of this paper, although he also notes that if you assume a "wrap-around" boundary condition (practical only for texture synthesis), you can transform the whole problem into frequency space and use a pretty – albeit slower – FFT-based solver

A really clear 2D java implementation of this paper is available at:
http://www.multires.caltech.edu/teaching/demos/