Stable Fluids

Jos Stam
Alias|wavefront
Seattle, USA

In this talk I will present some research I have been doing over the last four years on modeling fluids at Alias|wavefront.

Note on the notes. I typed these notes in one session as if I was giving the talk. Usually the talk takes about one hour. I haven't corrected the grammar at all so it is very informal, but hey this is a talk not a paper.

I have given this talk at the following places: University of Washington (May 2000), SIGGRAPH course in New Orleans (July 2000), INRIA Rocquencourt, (January 2001), Gamer Seminar at SFO (January 2001), IMA Workshop on Graphics in Minneapolis (May 2001), invited talk at the Spanish Conference on Computer Graphics in Girona (July 2001), SIGGRAPH course in LA (August 2001), invited talk at the Eurographics Workshop on Simulation and Animation in Manchester (September 2001), INRIA Grenoble (September 2001).

Fluids in Computer Graphics

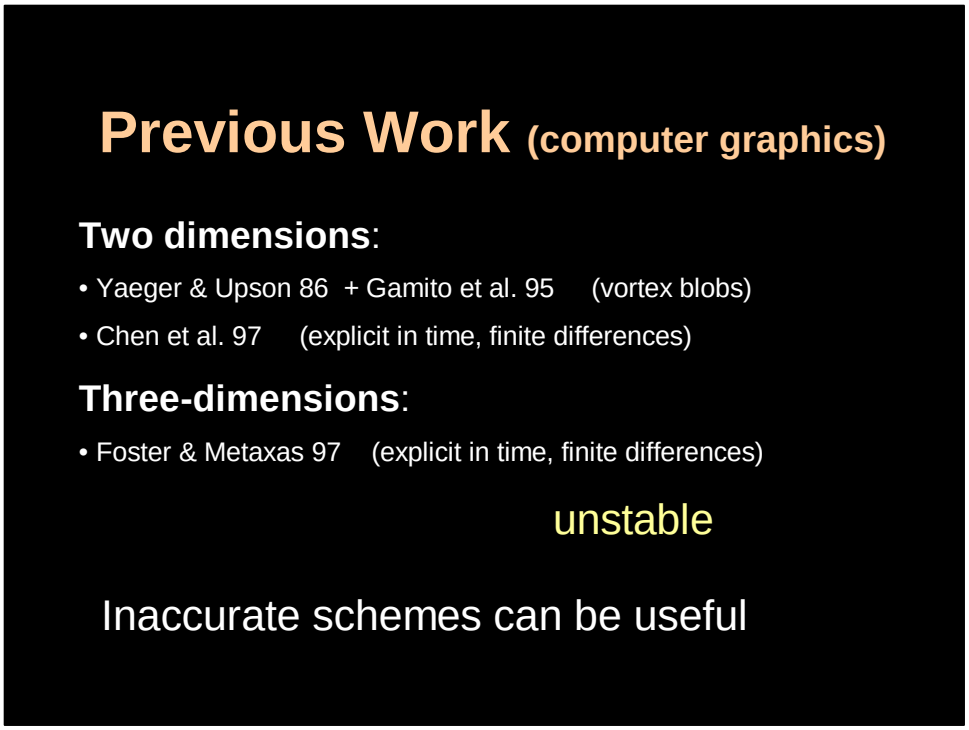
- Fast.
- Looks good.
- Easy to code.

The main goal in computer graphics is to have fluids that are both fast and look convincing. Ideally we want a user to be able to interact in real-time with a virtual fluid. In this manner effects can be orchestrated more rapidly. Real time performance is also important in games, for example. In addition I like solvers that aren't too hard to code. Later in this talk I will show you that a version of my solver can be coded in roughly 60 lines of C code.

Fluid Mechanics

- Natural framework for fluid modeling
  Full Navier-Stokes Equations
- Has a long history
  reuse code/algorithms
- Equations are hard to solve
  non-linear

To achieve these goals we can do whatever we want. Historically fluids in computer graphics have been modeled using a combination of simple primitives and the clever usage of texture maps. This is fine, but the dynamics of fluids are very hard to capture that way, and I am speaking from experience. A more natural way to model fluids is to use the physical equations that describe their motion. These equations are known as the Navier-Stokes equations and have been around for quite some time now. So potentially we can reuse the abundant literature in physics and engineering. Of course the reason that there are so many articles published in this area is that these equations are very hard to solve because they are non-linear.



It is therefore not surprising that so little work has focused on solving these equations directly in computer graphics. Why should we be able to solve these equations if the experts at NASA can't ? But there has been some work. The early techniques were mostly restricted to 2D and used techniques such as vortex blobs that work mainly in 2D. The most important previous work, at least to me, was the paper by Foster and Metaxas. They clearly showed that very convincing flows could be obtained even on coarse grids. Their technique is also quite easy to implement. Unfortunately it is not fast. The problem is that they use explicit scheme which become unstable for large time steps or large velocities.

Main Contribution

Stable Navier-Stokes solver

Any time step can be used

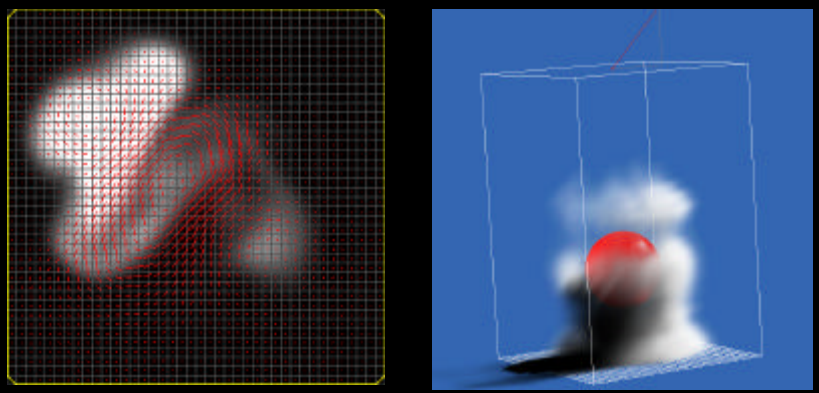
Bigger time steps = faster simulations

NOT accurate

So the motivation for this work was to extend Foster and Metaxas' work and make the solver stable for any time step. Large time steps means faster simulation. In fact you can make them as fast as you like. They might look strange but the simulation will not blow up. Of course the technique cannot be claimed to be accurate. But in graphics if it looks good it is good. And I will let you be the judge whether my simulations look good or not when I give the demos in a moment.

Application

Use velocity to move densities:



There are of course many applications for a fluid solver. In this talk I will use the fluid solver to move densities like smoke around in an environment. Here I show two snapshots from my interactive solver. The one on the left is from the 2D solver. The velocity field is shown in red and the density naturally follows the field. On the right is an example in 3D of a sphere interacting with the smoke density.

Application

Use velocity to move densities:

While (simulating)
- Get force from UI
- Get density source from UI
- Update velocity
- Update density
- Display density

The main structure of my solver is as follows. It is basically a single while loop. First I get some forces from the User Interface. In 2D these can be related to the movement of the mouse for example. Then I read in sources of densities from the UI. Then I update the velocity and the density using the solver and finally I display the density.

Equations

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

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

+ velocity should conserve mass

Equations very similar

To achieve this we need the physical equations for both the evolution of the density and the velocity. Here they are. What is immediately apparent is that these two equations look a lot the same. This should be obvious even to someone who has never seen these equations before. The top equation describes the evolution of the density denoted by rho. The velocity is denoted by the boldface vector u. The evolution of the velocity is given by the second equation which are the Navier-Stokes equations. The first equation is linear in rho and is much easier to solve than the second equation which is non-linear. The non-linear term is the second term on the right hand side where u appears twice. This term makes these equations hard to solve. Historically I first worked on a solver for the simpler equation and then applied to the harder equation. So in this talk I will first explain to you how to solve the first equation and then I will show you that the exact same techniques can be used to solve the second equation.

Equations

Evolution of density (assume velocity known)

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

Over a time step...

The first equation tells us how the density evolves over time, that's what the symbol on the left means. And the change is due to three causes which correspond to the three terms on the right hand side of the equation.

Equations

Evolution of density (assume velocity known)

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

Density changes in the direction of the flow

The first time says that the density should follow the velocity field. This makes sense since if I blow on smoke smoke it will naturally follow the direction of the wind field.

Equations

Evolution of density (assume velocity known)

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

Density diffuses over time

The second term says that the density may diffuse at a rate kappa.

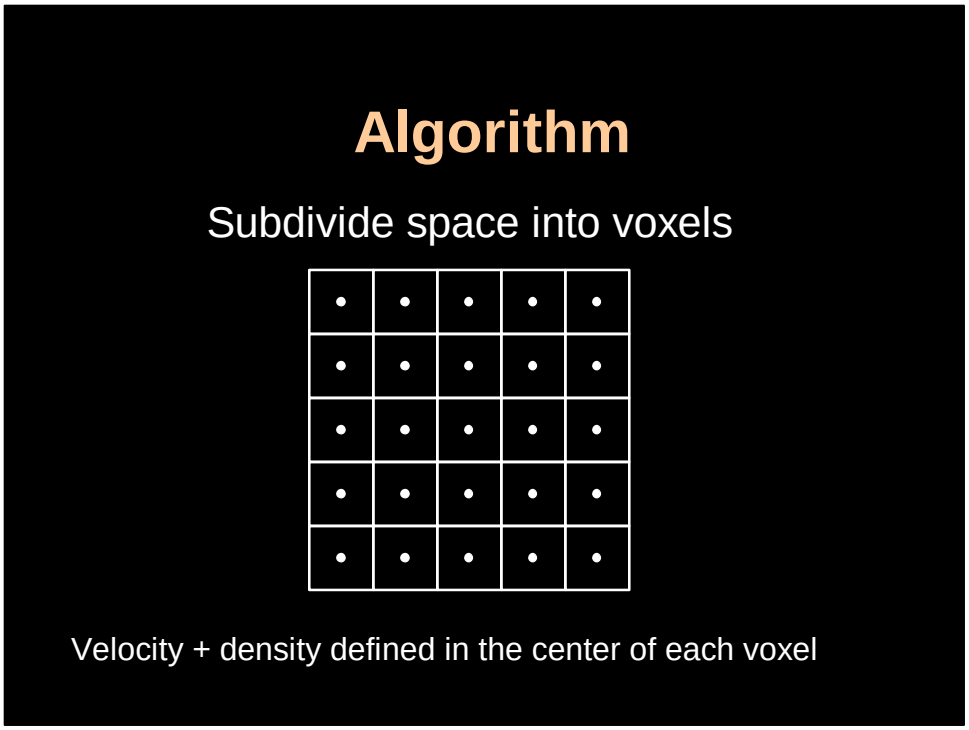
Equations

Evolution of density (assume velocity known)

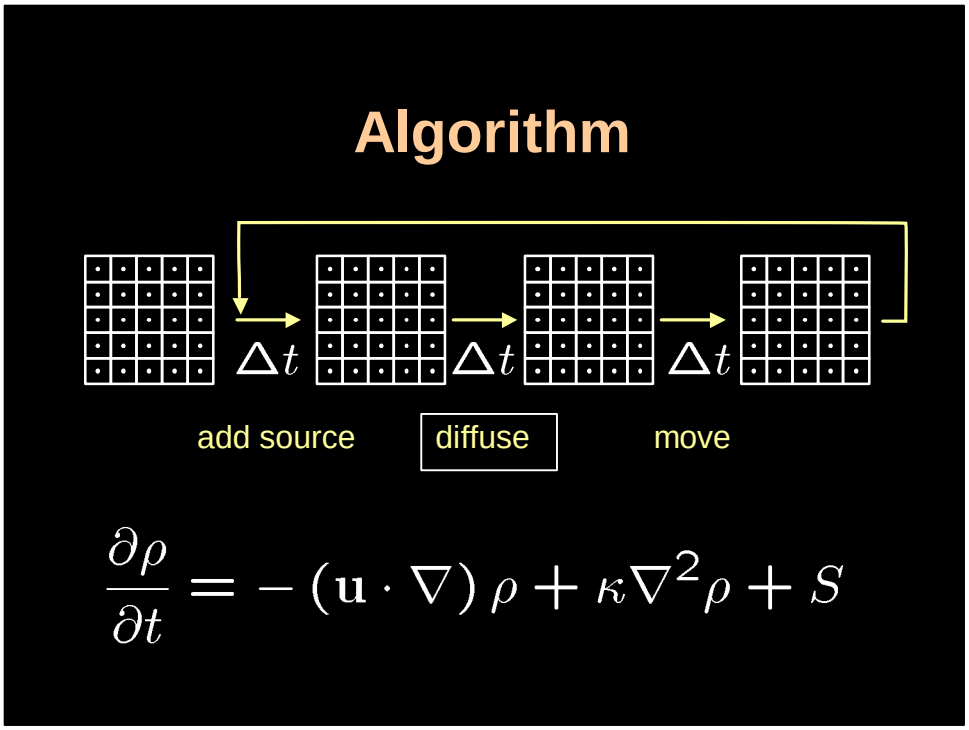
$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Increases due to sources from the UI

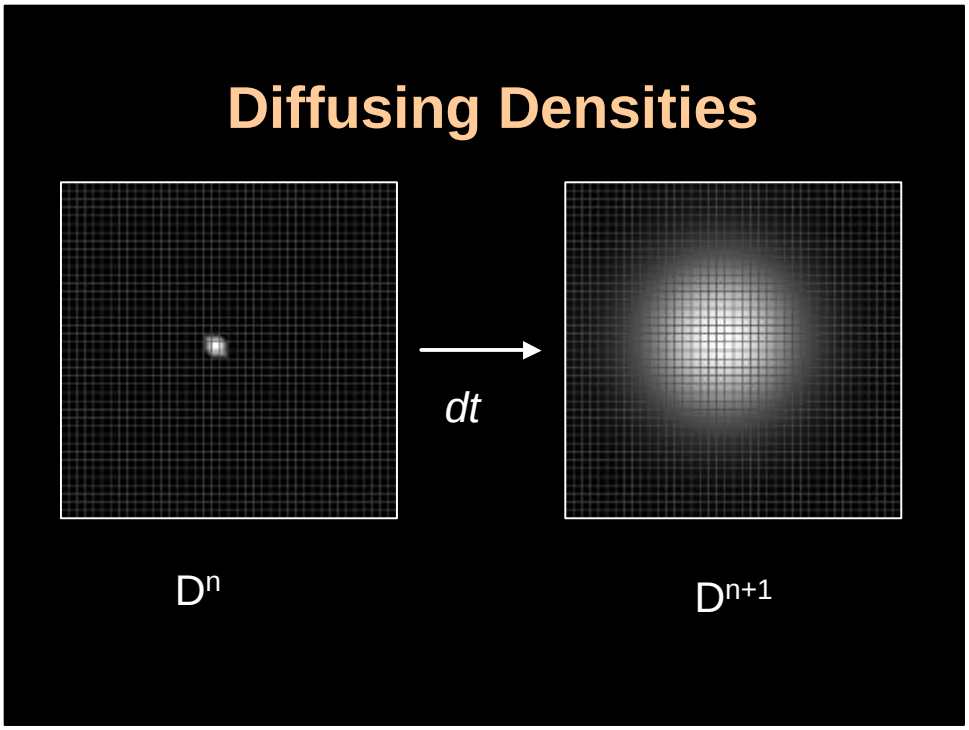
And finally the last term says that the density should increase due to external source. As I said before these sources are provided by the user through a suitable user interface.



Ok so how do we solve these equations. Well first we discretize the entire space into identical voxels with the density defined at the center of each voxel. In this talk I will present the 2D case just so everything is easier to visualize. However, nothing I say will be restricted to 2D. Everything I say can be easily extended to 3D by adding another index and or another for loop.

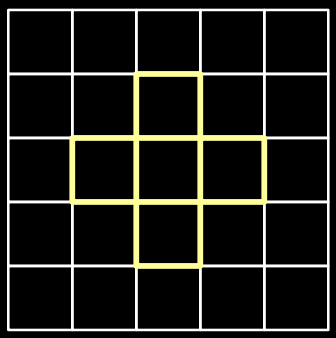


The main structure of the solver follows that of the equation. We first start with an initial grid of densities which is typically empty. Then we update this grid in three steps. Each step correspond to one of the terms of the equation. First we add the sources to the grid. This is really easy. In my implementation the sources are provided by a grid. So all I have to do is multiply this grid by the time step and add them to the density grid. Let me now explain how to solve for the diffusion step.



In this step we want to account for the effects of diffusion. This is shown in this slide. On the left is the density grid before diffusion and on the right after diffusion.

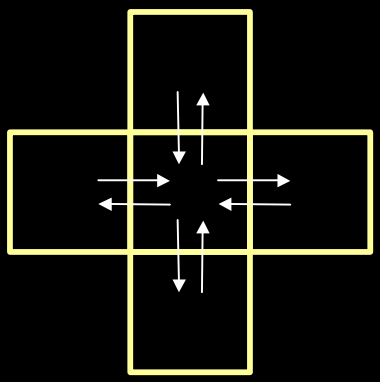
Diffusing Densities



Exchange of density between neighbors

The basic idea is to look at the exchange of density between immediate neighbors only. They are highlighted here in yellow for the cell in the center.

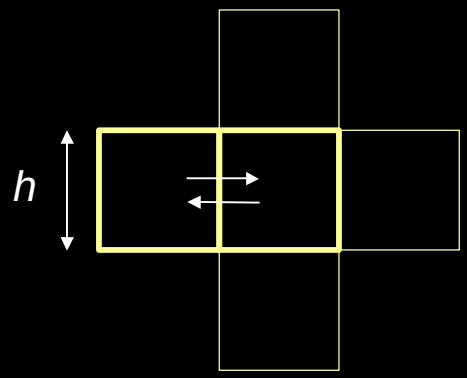
Diffusing Densities



Exchange of density between neighbors

We assume that density is exchange out and into the cell through the adjacent faces (edges in 2D).

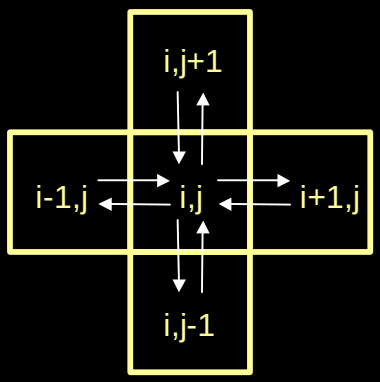
Diffusing Densities



Change = density flux in - density flux out
= $k\ dt$ (neighbor - center) / h^2

For a single face (edge) the exchange is equal to the density of flux coming in minus the density flux coming out. This is simply the difference in densities multiplied by the time step, the diffusion rate kappa and divided by the grid spacing squared. So the flux is higher for large time steps, large diffusion rate or small grid spacings.

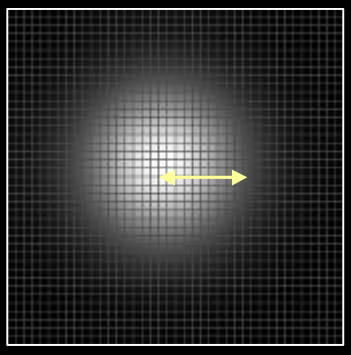
Diffusing Densities



$$D^{n+1}_{i,j} = D^{n}_{i,j} + k\,dt\,(D^{n}_{i-1,j} + D^{n}_{i+1,j} + D^{n}_{i,j-1} + D^{n}_{i,j+1} - 4D^{n}_{i,j})/h^2$$

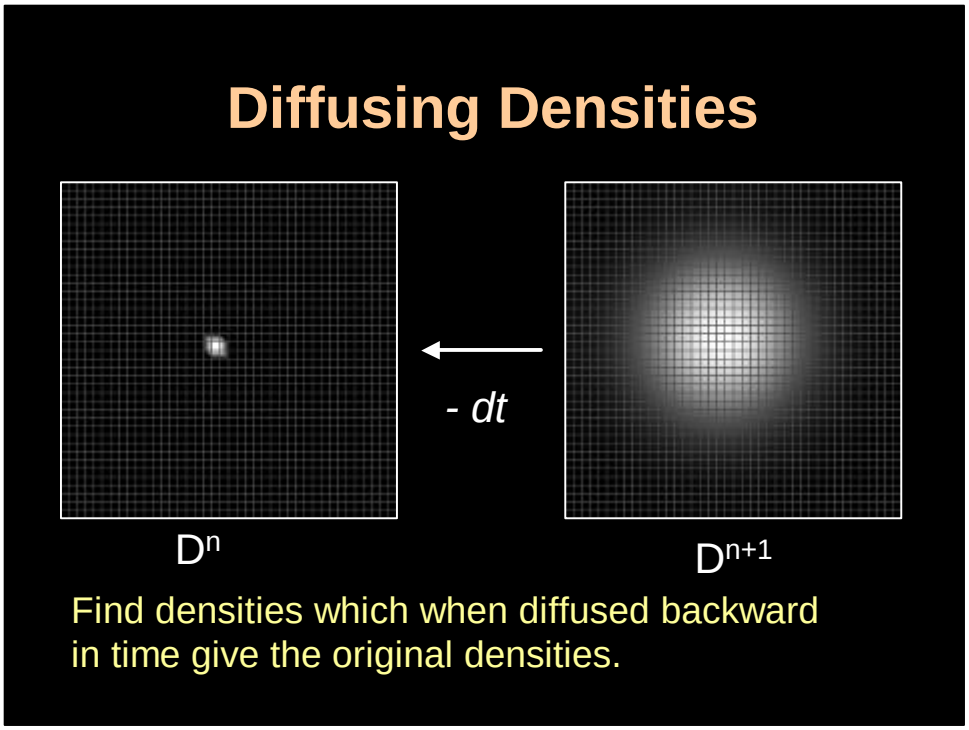
By summing up the fluxes for all the faces we end up with the following update rule. This is really easy to code simply add two for loops around this equation and you are done. Unfortunately it doesn't work.

Diffusing Densities



Unstable when $k\,dt / h^2 > 1/2$

The reason it doesn't work is that the it can become unstable when the diffusion rate is too high, time step too large or grid spacing too small. The problem is that this method breaks down when the density propagates further than just between neighbors.



So we have to consider an alternative. The alternative is to use implicit techniques. Intuitively we look for the densities which when diffuse backward in time give us the densities that we currently have. Implicit techniques always work this way by looking back in time.

Diffusing Densities

Linear system:

$$D^{n+1}_{i,j} - k\,dt\,(D^{n+1}_{i-1,j}+D^{n+1}_{i+1,j}+D^{n+1}_{i,j-1}+D^{n+1}_{i,j+1}-4D^{n+1}_{i,j})/h^2 = D^{n}_{i,j}$$

$$A\,x = b$$

A can be huge *but* is sparse

-> requires fast linear solver

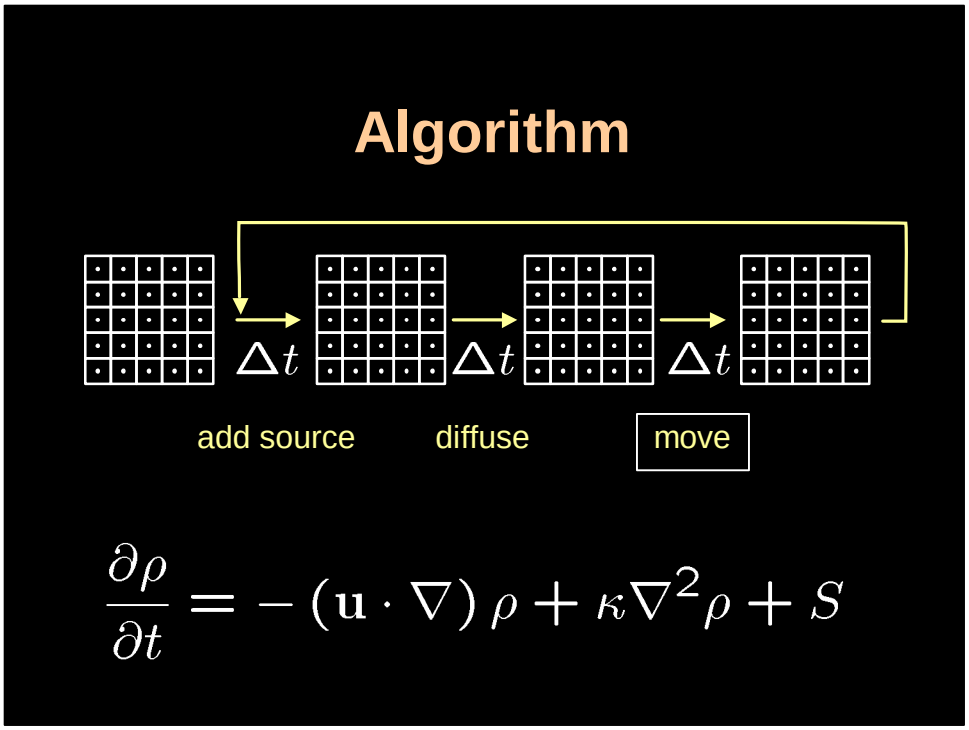
So now we can rewrite our update rule with the roles of Dn and Dn plus one exchanged and the time step reversed. The problem here is that all the terms on the left hand side are unknown. So we end up with a linear system that we have to solve. This seems like a crazy idea, the unstable solution is so simple and now we have to solve an entire system. Actually this isn't all that bad. First the system is sparse and many fast solvers exist. Also in practice the cost of solving the system over a large time step is much more effective than taking many tiny steps with a cheap unstable solver.

Diffusing Densities

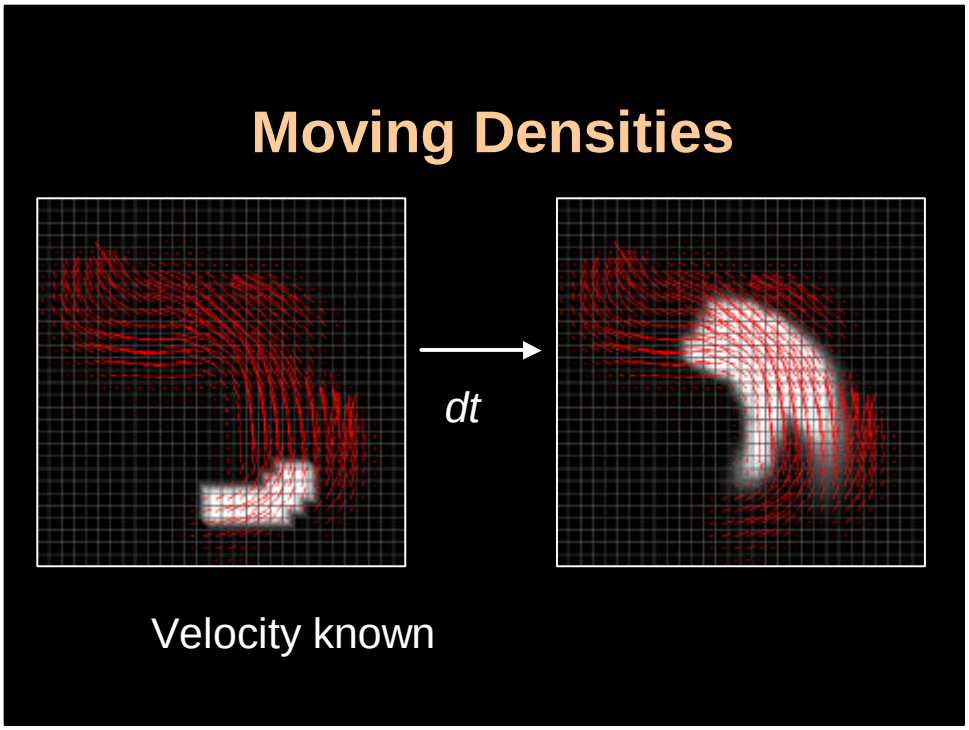
Linear solvers:

Name	Cost	Comments
Gaussian elimination	N^3	Use only for very small N (test code)
Jacobi/SOR relaxation	N^2	Easy to code but slow
FFT/cyclical reduction	N logN	Use when no internal boundaries
Conjugate gradient	$N^{1.5}$	Use when internal boundaries
Multi-grid	N	Slower than FFT in practice. Hard to code when internal boundaries present

So what kind of linear solvers are out there. The most naïve solver is to use Gaussian elimination. These solvers are only good for small or dense matrices. But they are available in many standard libraries and are useful to debug the solver code for small grids. The easiest solver to implement is simple relaxation such as Jacobi or Gauss-Seidel. The only problem is that they do not converge all that fast. I would start with this solver and later when the code is up and running switch to a more sophisticated technique such as the FFT based ones or the conjugate gradient. Both of these techniques converge much faster but are a little harder to implement. The FFT based techniques are really fast but only work when there are no internal boundaries, like an object in the fluid. These solvers are available in FISHPAK from netlib.org. It's in Fortran but you can convert it to c using the f2c tool also available at netlib.org. When objects are present in the flow I recommend using the conjugate gradient technique. It is fairly easy to code. In fact you can find some nice C++ templates (which I converted to C code) from the IML++ library at NIST. Finally, multigrid solvers are theoretically optimal. Although they are order N, the constant is quite high. And unless you consider huge grids both FISHPAK and the conjugate gradient will work much faster. Also multigrid methods are a total pain to code. Especially when objects are present or you consider grid sizes which are not powers of two.



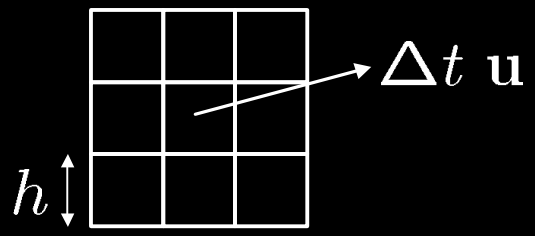
So this takes care of the diffusion step. Let's now turn to the last step in updating the density.



In this step we assume that the velocity is known. Over a time step we want to move the density along the velocity field.

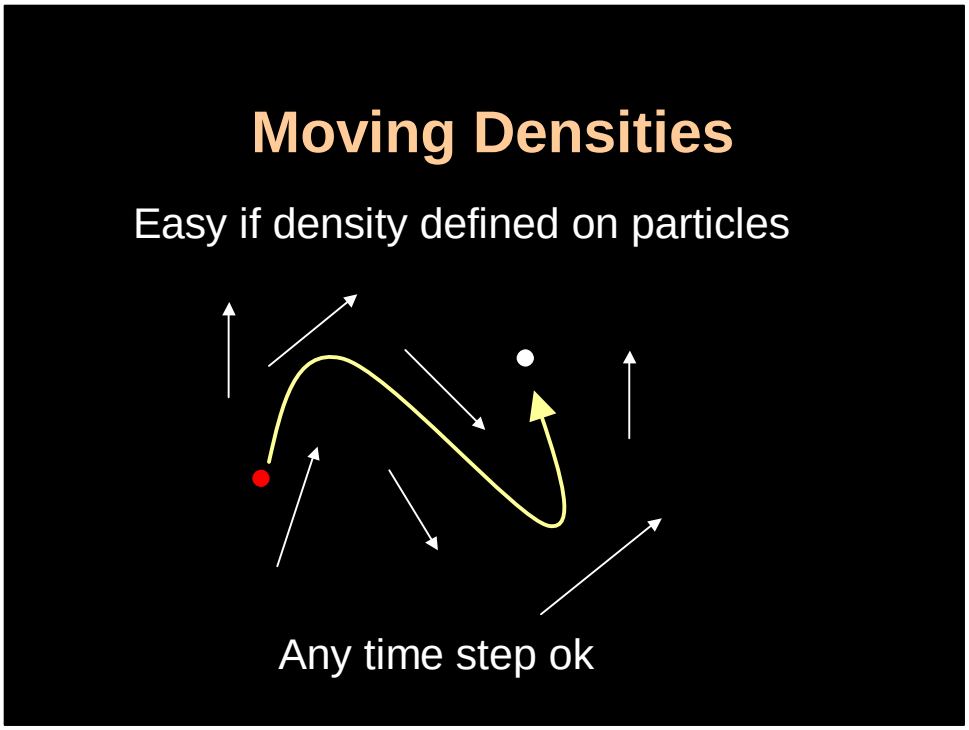
Moving Densities

Finite Differences: transfer only between neighbors

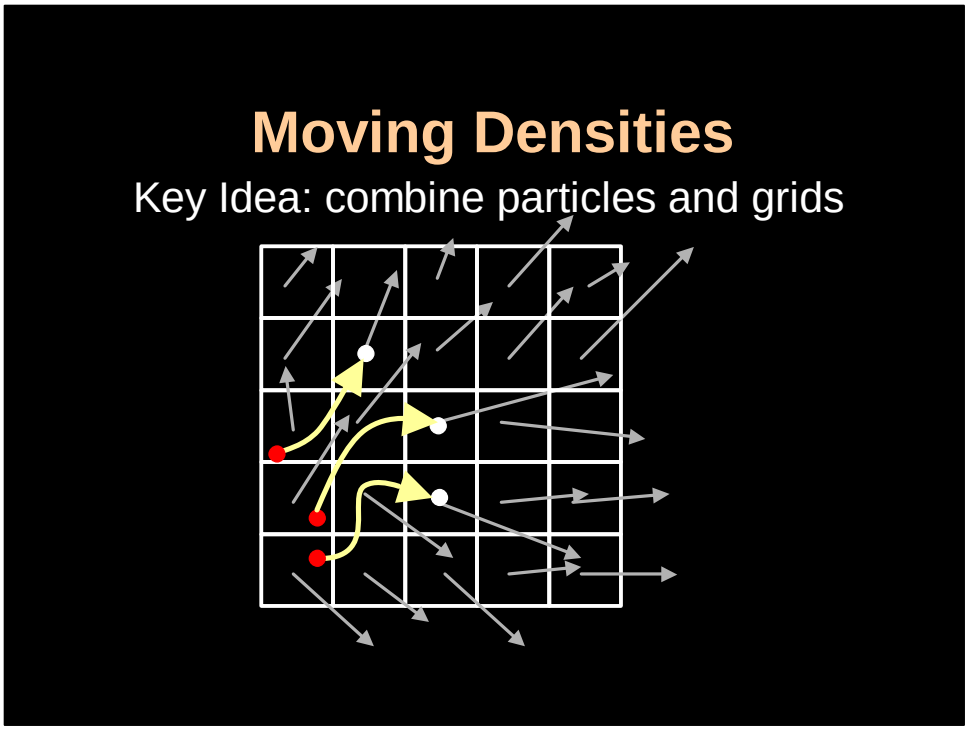


Unstable when $\Delta t\ |\mathbf{u}| > h$

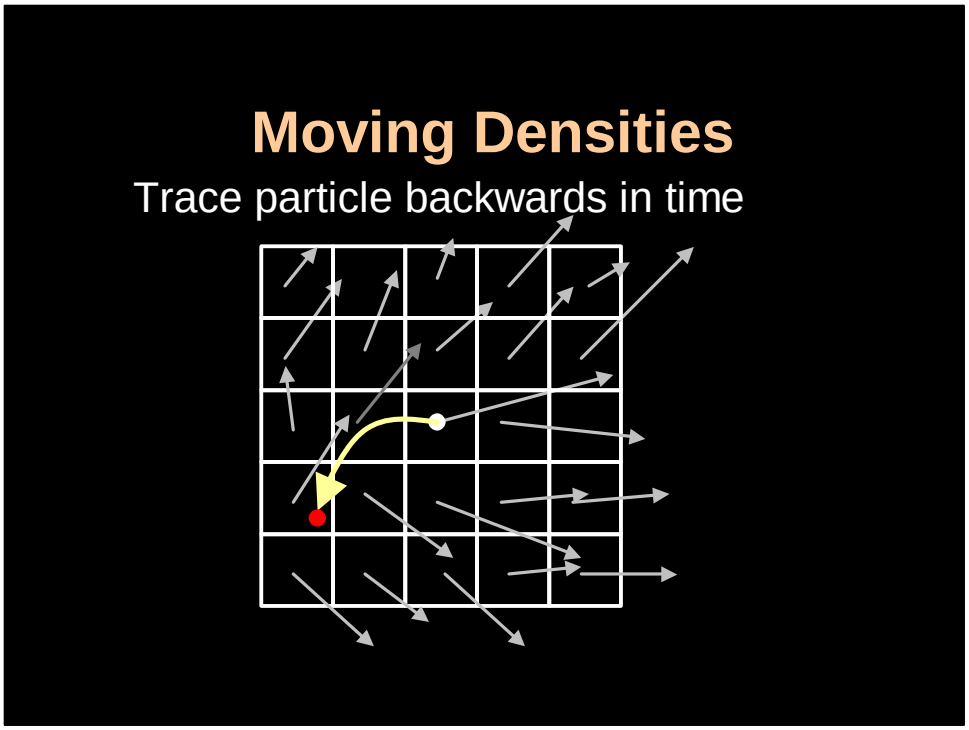
As for the diffusion step we can look at direct neighbors only and consider the fluxes. In this case the fluxes will be biased by the direction of the velocity field. The naïve implementation of this scheme again results in instabilities. The problem again is when the transfer is between neighbors that are more than one cell removed. This happened when either the velocity is too large, the time step too big or the grid spacing too small. We could use a stable implicit method, but the problem here is that the resulting linear system has a non-symmetrical matrix with varying constants. So the Conjugate Gradient solver cannot be used. Of course there are variants of the conjugate gradient such BiCGSTAB2 but they can become unstable when the system is ill-conditioned. However, it turns out there is a really simple technique that is stable and can solve for this step.



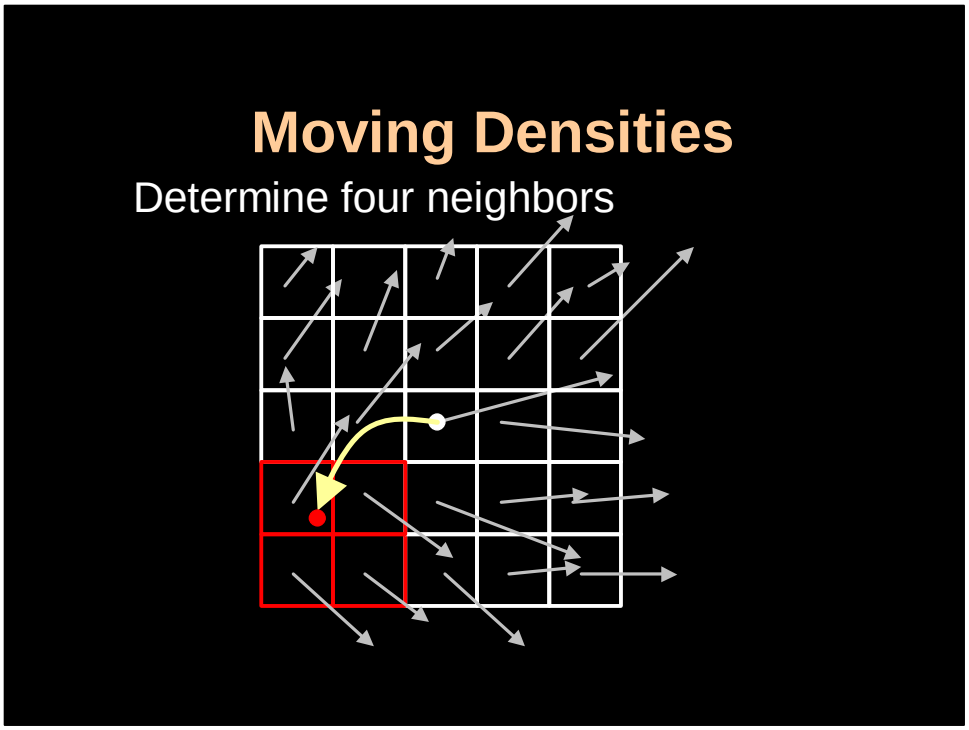
The basic idea is as follows. If the density was sampled on particles then this step would be trivial to compute: simply move the particles along the field using a particle tracer. Of course the problem is that we then have to use a particle solver for the other terms as well. And no good techniques are known to me for doing this. So we still want to keep the grids around.



Our idea is to find the positions of the particles that after one time step end up exactly at the grid centers.



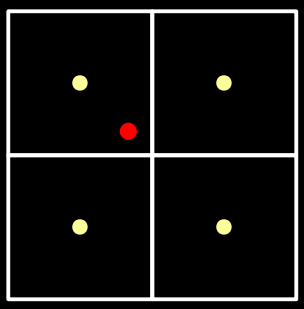
To find these particles we simply trace back each voxel center of the grid backwards through the field.



Doing this we will end up somewhere else in the grid. We first locate the four closest cells to the point.

Moving Densities

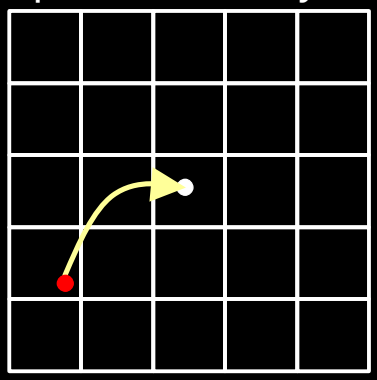
Interpolate the density at new location



And then we interpolate the density from these cells…

Moving Densities

Set interpolated density at grid location



Requires two grids

… and set the interpolated value as the new density of the departure cell. For this to work we require two grids. One that contains the density values of the previous time step and one that will contain the new interpolated values.

Moving Densities

This scheme is unconditionally stable:

$$\rho_{int} = (1 - s)\rho_0 + s\rho_1$$

$$\rho_0, \rho_1 \leq \rho_{max}$$

$$\rho_{int} \leq (1 - s + s)\rho_{max} \leq \rho_{max}$$

$\longrightarrow$ density is always bounded

The important property of this technique is that it is unconditionally stable: no matter how big the time step this technique will not blow up. This is why. Since we have a regular grid which is a tensor product of two one dimensional grids I only have to prove it for 1D data. Since the new data is a linear interpolation of previous data we have that the new maximum of the new densities is always bounded by the maximum density of the old values. So the density is always bounded no matter how big the time step and therefore will never blow up and become unstable.

Computing Velocities

Now let’s go back to the computation of the velocity.

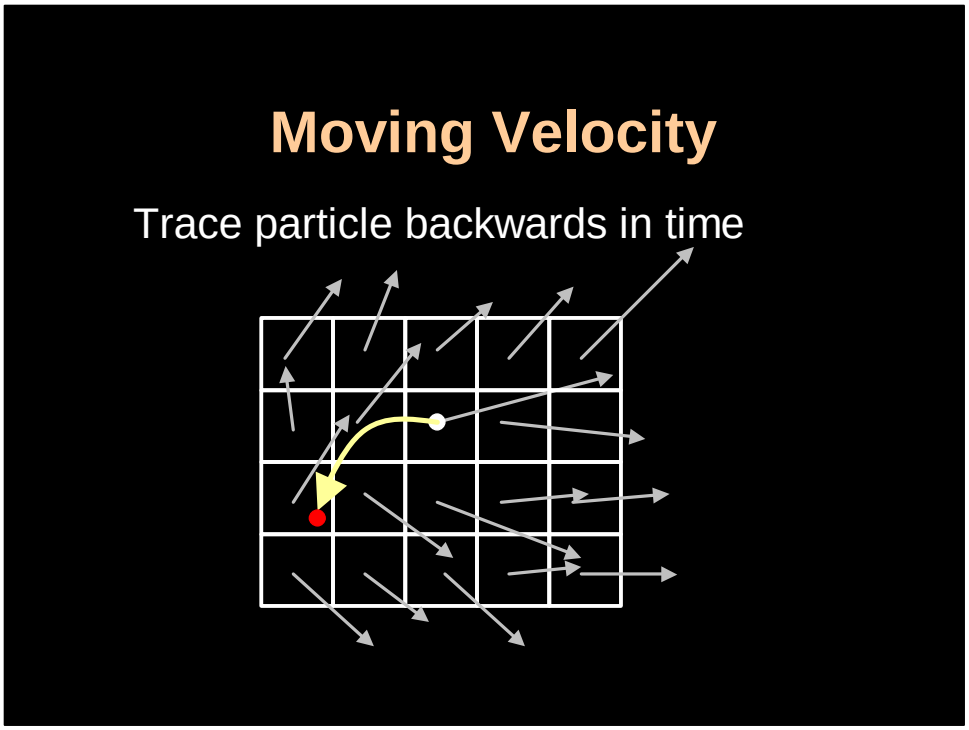
Computing Velocities

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

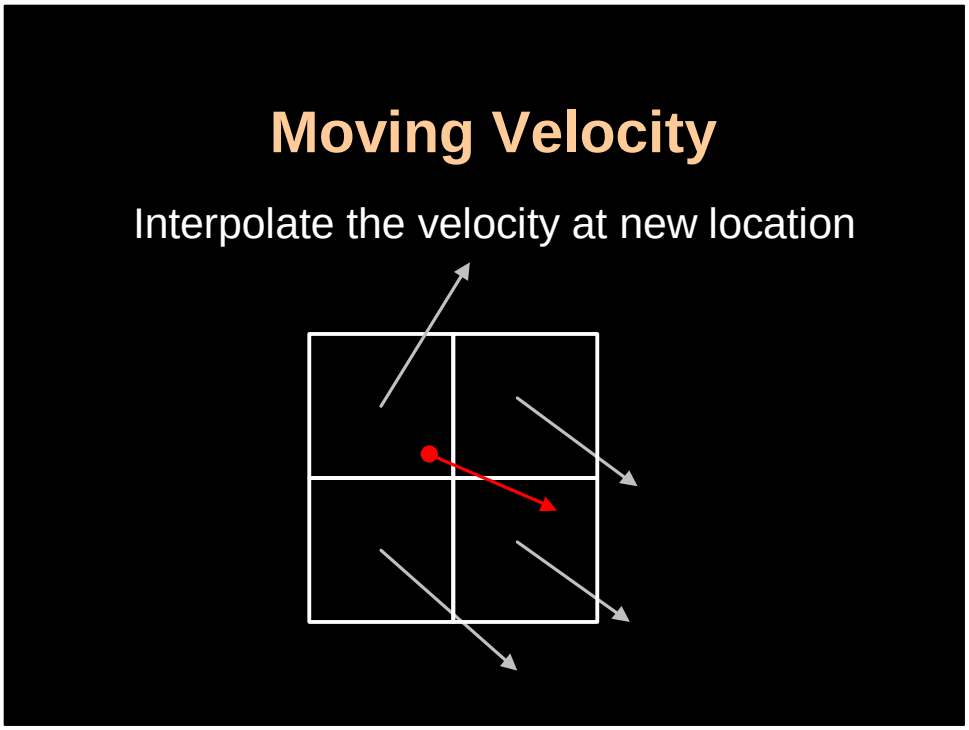
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

Velocity is moved by itself

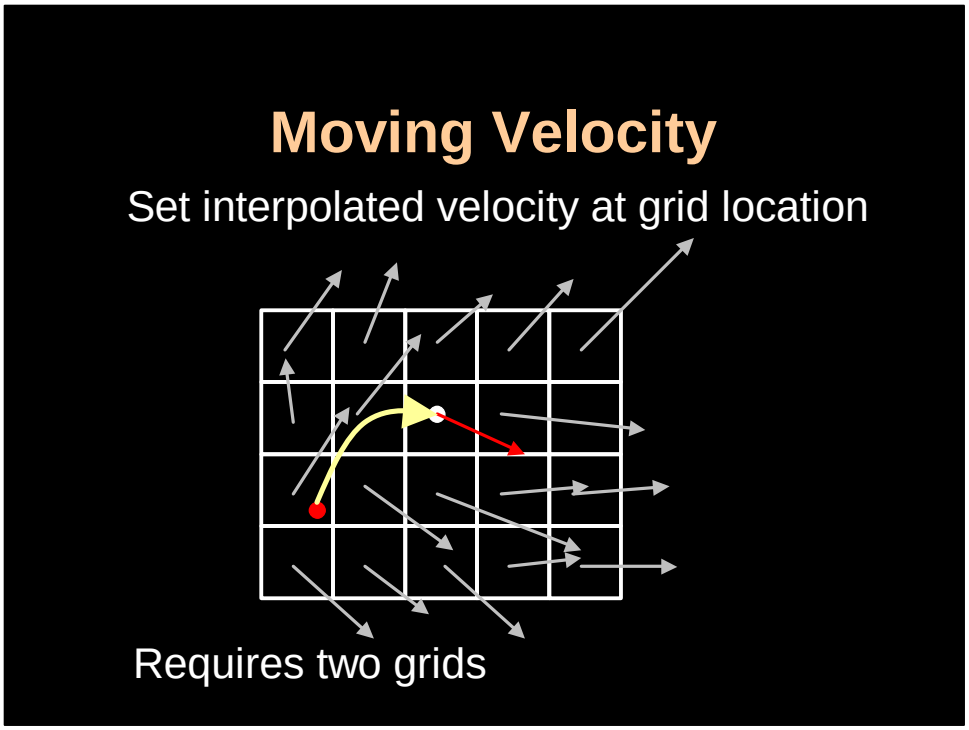
As I said earlier the equation I just showed you how to solve for the density looks a lot like the equation for the velocity. The velocity also has a "diffusion" term which accounts for the effects of viscosity. The higher the viscosity constant nu, the more the velocity field will be smooth, resulting in viscous-like fluids. In this case we have to solve two diffusion equations in 2D and three equations in 3D, one for each component of the velocity field. The last term is the force term and can also be accounted similarly as for the sources in the density solver. Finally the first term is the most interesting one. It looks just like the corresponding term for the density, except that the velocity appears twice making it non-linear. In the case of the density this term states that the density should follow the velocity field. So we can interpret this term as saying that the "velocity should move along itself". This might sound weird at first but this is how we can interpret it. And in fact we can blindly apply the same algorithm we used for the density. At first I never expected this to work and I was lucky that my first implementation didn't have bugs, if not I would have thought, Ah another crazy idea that doesn't work. Instead I was amazed that it worked so well.



Se here we go. Again we need two grids for the velocity. One that contains the old values and one that contains the new interpolated values. As before we trace each grid point backwards in time using the old velocities.



Doing this we end up somewhere else in the grid. And as for the density we interpolate a new velocity at that location from the neighboring grid cells.

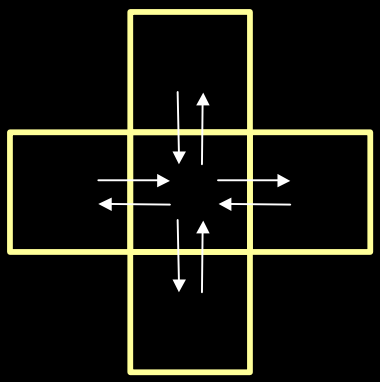


And then we set the new velocity to the interpolated one. Again this method is stable just like for the density solver. I learned from Ron Fedkiw that this technique was first invented in 1952 by Courant, Rees and Isaacson and has been rediscovered by many researchers in different fields. It is best known as a semi-Lagrangian technique.

Conservation of Mass

There is still one step we have to enforce before we are done and that is that the fluid should conserve mass.

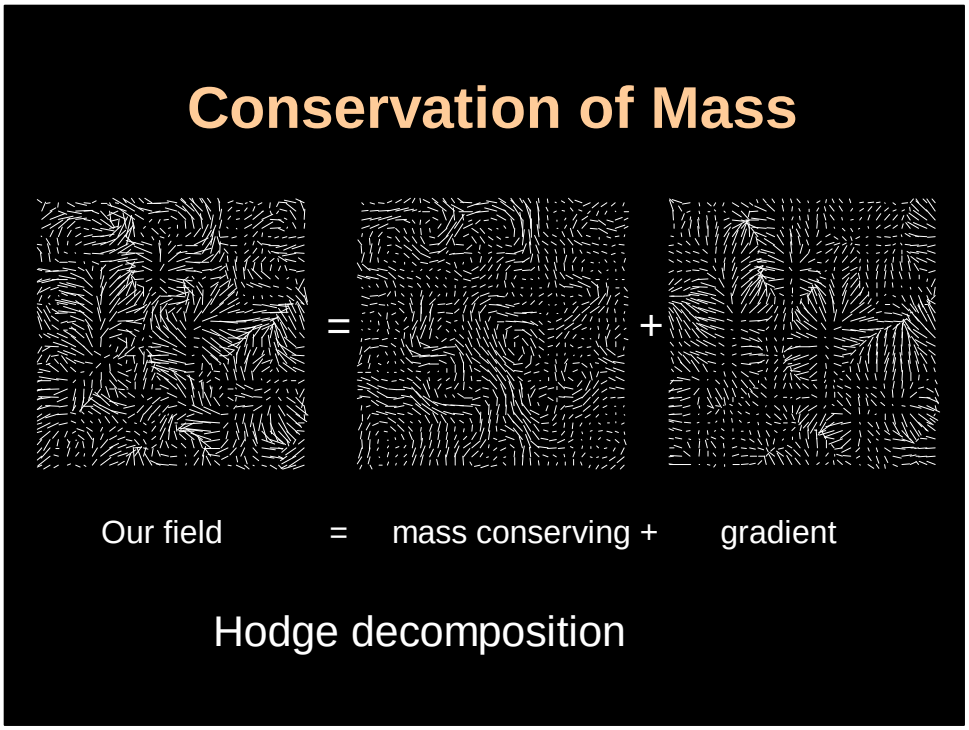
Conservation of Mass



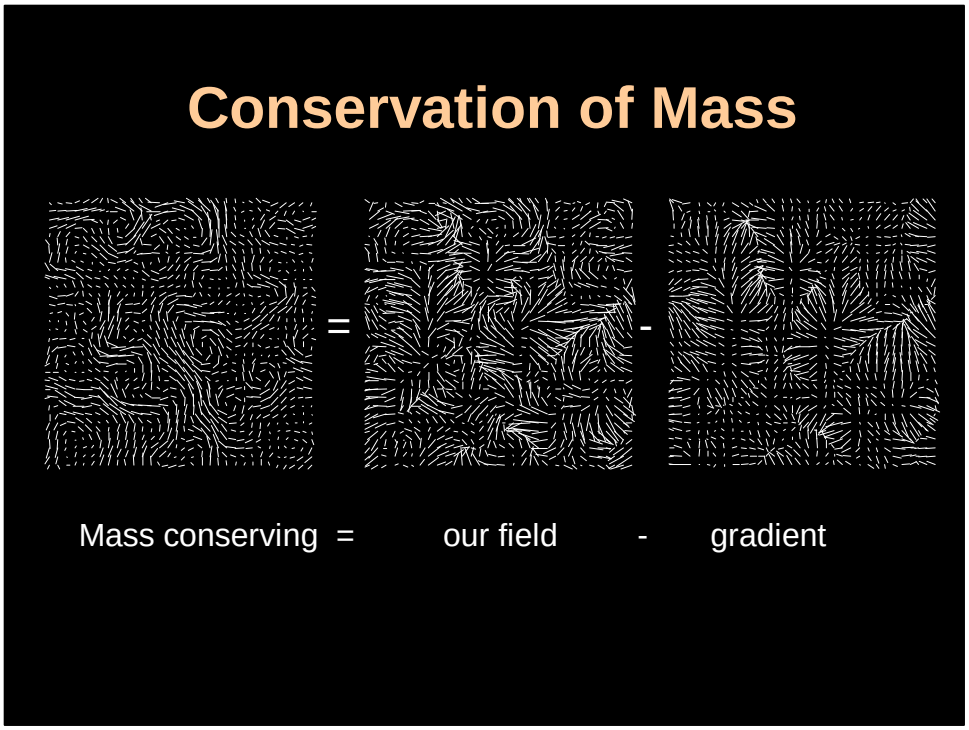
Flow into cell = Flow out of the cell

$U_{i+1,j} - U_{i-1,j} + V_{i,j+1} - V_{i,j-1} = 0$ not

What this means is that we want the flow into a cell to be equal to flow out of it. This results in the constraint shown on the slide. In practice after the three previous steps this is never the case. So the idea is to correct the situation in a final step.



To to this we use a mathematical result know as the Hodge decomposition of a vector field. This result states that every vector field such as the one shown on the left is the sum of a mass conserving field and a gradient field. The mass conserving field is exactly the sort of vector field we want as it has nice vortices which will result in swirling looking flows. The gradient field on the other hand is the worst possible case: at every point the flow either is all inward or outward. The gradient field can be visualized as being the slope function of some height field, it is defined entirely be a single scalar field.



To get a mass conserving field from an arbitrary vector field we simply subtract the gradient part from it. This requires us to find the scalar function that defines the gradient field.

Conservation of Mass

Scalar field satisfies a Poisson Equation:

$$P_{i+1,j}+P_{i-1,j}+P_{i,j+1}+P_{i,j-1} - 4\,P_{i,j} = (U_{i+1,j} - U_{i-1,j} + V_{i,j+1} - V_{i,j-1})\,h$$

Linear system

It turns out that this gradient field can be computed by solving the following Poisson equation. Again we have a sparse symmetrical linear system that we can solve using any of the solvers that I mentioned when talking about the diffusion step. Typically conjugate gradient is a good idea with a good preconditioner, Jacobi did the job for me.

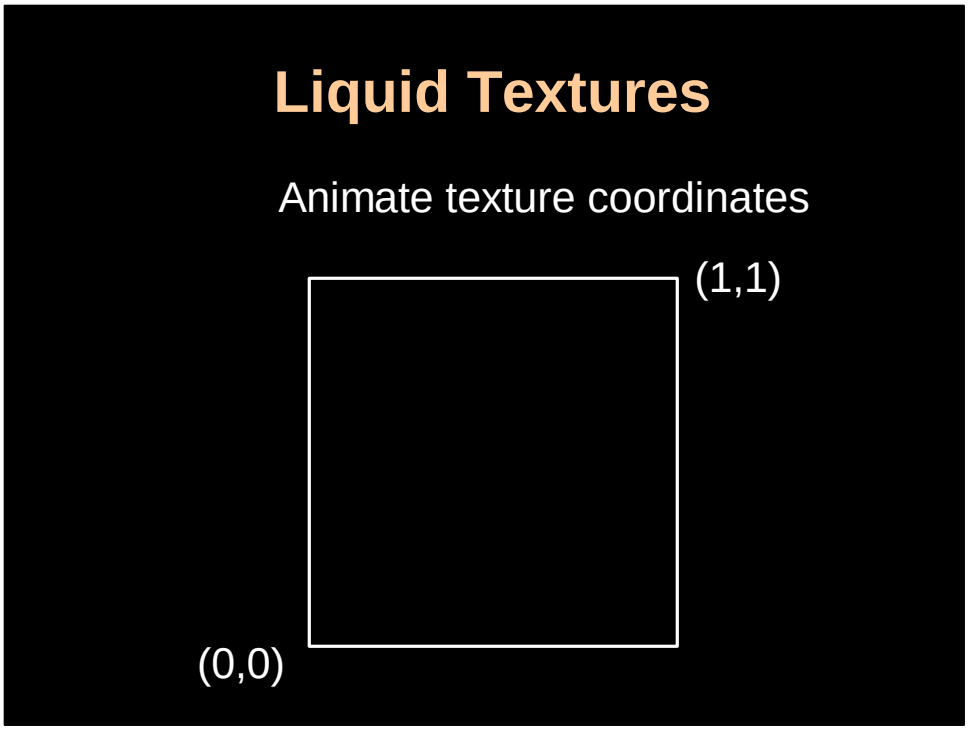
Summary

```
UpdateVelocity(U1,U0,F,visc,dt)
   AddForce(U1,U0,F,dt)
   Diffuse(U0,U1,visc,dt)
   Move(U1,U0,U0,dt)
   ConserveMass(U1,dt)
```

Very easy to code. Only need:
- Particle tracer + grid interpolator
- Linear solver (FISHPAK or CG)

So in summary here is all you need to write a fluid solver: a linear solver for the diffusion and the mass conservation step and a good particle tracer and grid interpolator for the self-advection step.

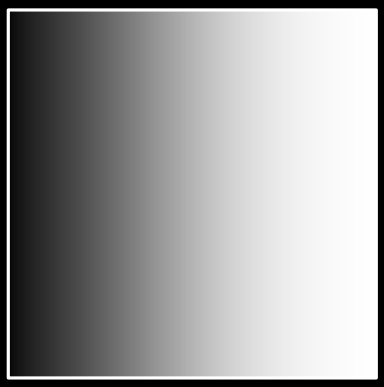
At this point I show a 2D demo that demonstrates each step of the algorithms just described.



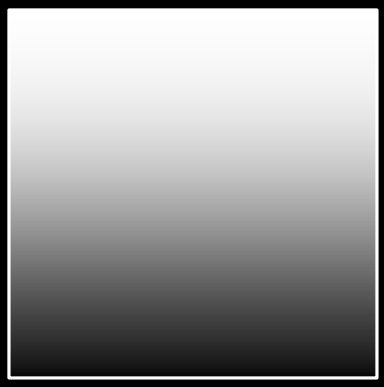
A nice way to add detail to the flow is through texture maps. This is a popular way to add visual detail in computer graphics. We assume that our fluid density has initially the texture coordinates shown on the slide. (0,0) in the lower left corner and (1,1) in the upper right corner.

Liquid Textures

Treat texture coordinates as densities

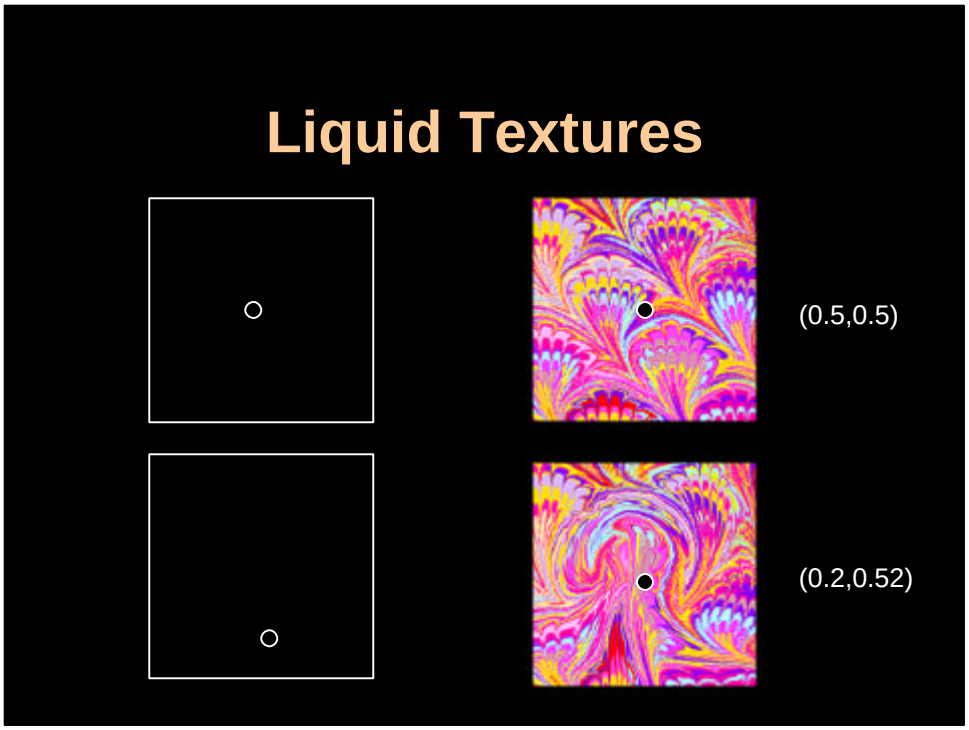


U-coordinate



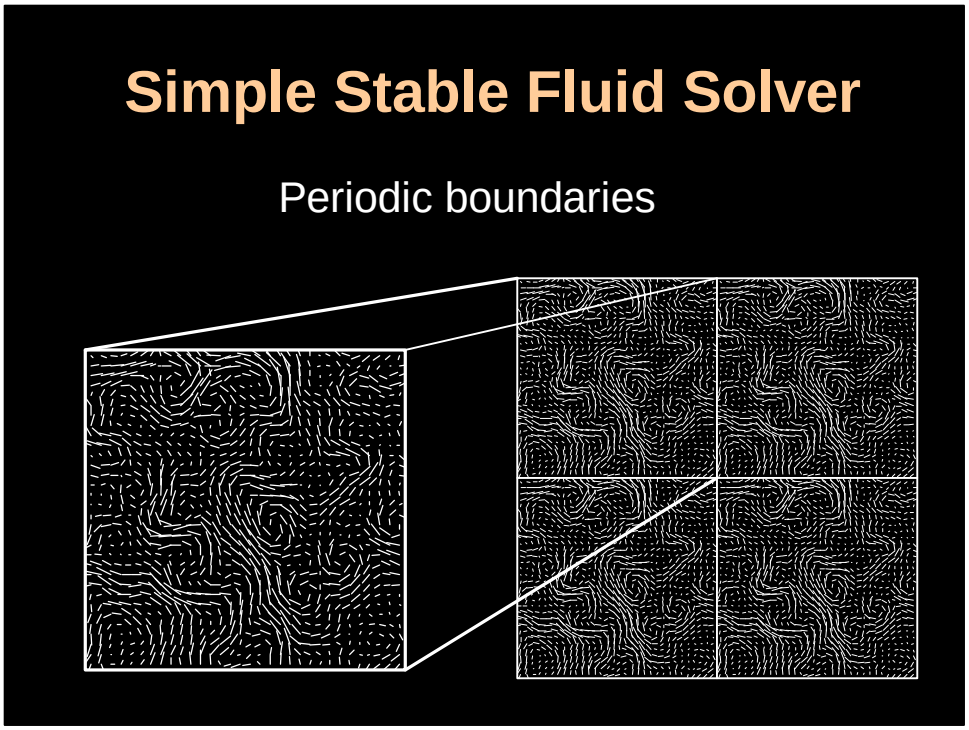
V-coordinate

Now we can interpret the texture coordinates as two densities which initially are just equal to the ramps shown here. Then the idea is to feed these densities to the solver and let them evolve according to the velocity field. This will give the impression that the texture moves with the flow. So in essence we are simulating three densities: the density, the u coordinate and the v coordinate.

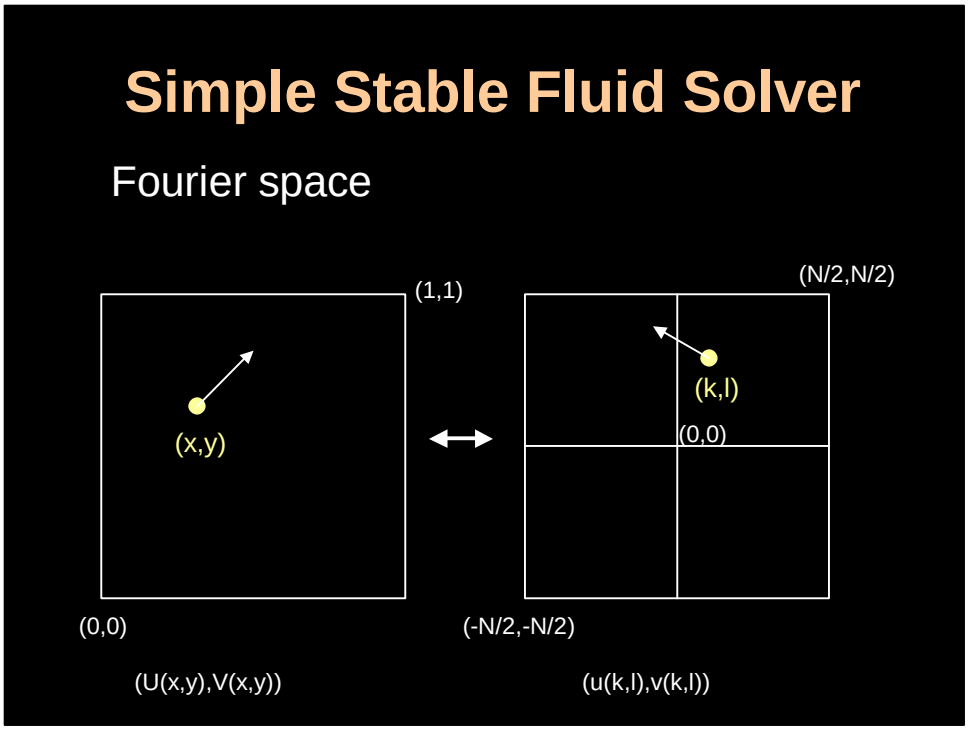


Here is an example. The point in the center of the texture initially corresponds to the center point in the fluid domain. However, when these texture coordinates are animated using the solver the coordinate in the center now point to a lower point in the texture map (when applying a force field upward). This will give the impression that the texture flows upward. Very simple trick that is pretty effective.

At this point I show some real time demos where I move these textures around. I also show a “wild paint” program where drops of different color fall down due to the effects of gravity.



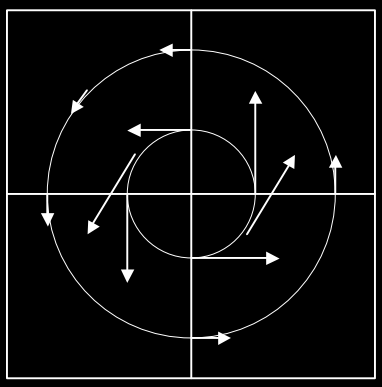
At the beginning of my talk I said that I could implement my solver in roughly 60 lines of code. You can do this in the case when the fluid is periodic. Which means that the flow is continuous across adjacent boundaries. So for example you can tile a single domain to the entire space. Like some funky wallpaper with arrows on them that also vary over time. These flows of course do not occur exactly in Nature, but they can be useful in computer graphics. For example it allows you to have field defined everywhere in space. So you can model some ambient turbulence that way. In fact if you throw some particles in the field, you will barely notice that the field is periodic. In fact I did something quite similar in my 1993 SIGGRAPH paper with Eugene Fiume.



Because the field is periodic we can use the Fourier transform. The Fourier transform of a vector field is also a vector field. Instead of assigning a vector to each point in space the Fourier transform assigns a vector for each wave number (k,l). The wave numbers have values between –N/2 and N/2, where N is the resolution of the grid. The reason that this is neat is that some operations are very simple to do in the Fourier space, while others are more easily done in the spatial domain.

Simple Stable Fluid Solver

diffusion = low pass filter: exp(-($k^2 + l^2$)??t)
mass conservation = velocity perpendicular to the fourier directions



More specifically both the diffusion step (viscosity remember) and the mass conserving step are very easy to compute in Fourier domain. The effect of viscosity is simply a low pass filter which dampens the higher frequencies. So the arrows get tinier as you move away from the center of the Fourier space as shown on the slide. In the Fourier domain a mass conserving field has the nice property that the velocity is always perpendicular to the wave number. So all the vectors are perpendicular to the circles centered at the origin. This is shown on the slide. The mathematical reason that this is the case is that incompressibility equation which reads Nabla dot u, becomes wave number dot u in the Frequency domain. A dot product is zero if an d only if the vector are perpendicular. So what all this says is that the velocity field has a very simple structure in the Fourier domain. A fact that is not widely know I think. The self-advection term is best solved in the spatial domain using the simple semi-Lagrangian style technique, where you trace each point back and do the interpolation. So is the addition of external forces which are usually localized in space. Although to model turbulence it might be a better idea to do it in frequency domain using a Kolmogoroff spectrum.

Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ )
   {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
   {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
      {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                        s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                        s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }
```

```
   FFT(1,n,u0); FFT(1,n,v0);

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

   FFT(-1,n,u0); FFT(-1,n,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j ]; }

   return;
}
```

60 lines of (readable) C code

So here is the code. All it assumes is that you have a good FFT solver. The one I use is called the Fastest Fourier Transform in the West (FFTW) which you can get from MIT for free. I will now go over the different parts of the code. The input is simply the velocity of the previous time step (u,v), the forces (u0,v0), the viscosity visc and the time step dt.

Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ )
   {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
   {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
      {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                        s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                        s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }
```

```
   FFT(1,n,u0); FFT(1,n,v0);

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

   FFT(-1,n,u0); FFT(-1,n,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j ]; }

   return;
}
```

Add forces

First we simply add the force grid multiplied by the time step to the velocity field.

Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ )
   {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
   {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
      {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                        s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                        s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }
```

```
   FFT(1,n,u0); FFT(1,n,v0);

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

   FFT(-1,n,u0); FFT(-1,n,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j ]; }

   return;
}
```

Move velocity

Then we do the self-advection step. Again we trace each voxel center back in time. The interpolation is quite straightforward since we do not have to worry about boundaries. A particle that exits one boundary reenters the grid from the opposite boundary. This can easily be implemented using couple mod operations in C. See the code.

Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ )
   {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
   {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
      {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                        s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                        s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }
```

```
   FFT(1,n,u0); FFT(1,n,v0);

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

   FFT(-1,n,u0); FFT(-1,n,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j ]; }

   return;
}
```

Diffuse + project

Next we transform our field to the Fourier domain where we do the diffusion and the mass conserving steps. The viscosity low pass filter depends on the wave number, the time step and the viscosity. The projection step is also very easy to implement. Notice that velocity values are complex numbers, however the storage is almost the same due to the symmetries of a Fourier transform of a real function. So in essence we only store the right half of the Fourier space. Another thing that is interesting is that the solver works in any dimension: simply add more arrays and for loops and the FFTW has a version for transforms in any dimensions. It is not clear to me what the applications would be of 4 or 5 dimensional fluid solver. Maybe the evolution of some set of parameters is governed by a Navier-Stokes Equation. The vector field in this case models the flow of change of these parameters. So I have a solution here that is looking for a problem.

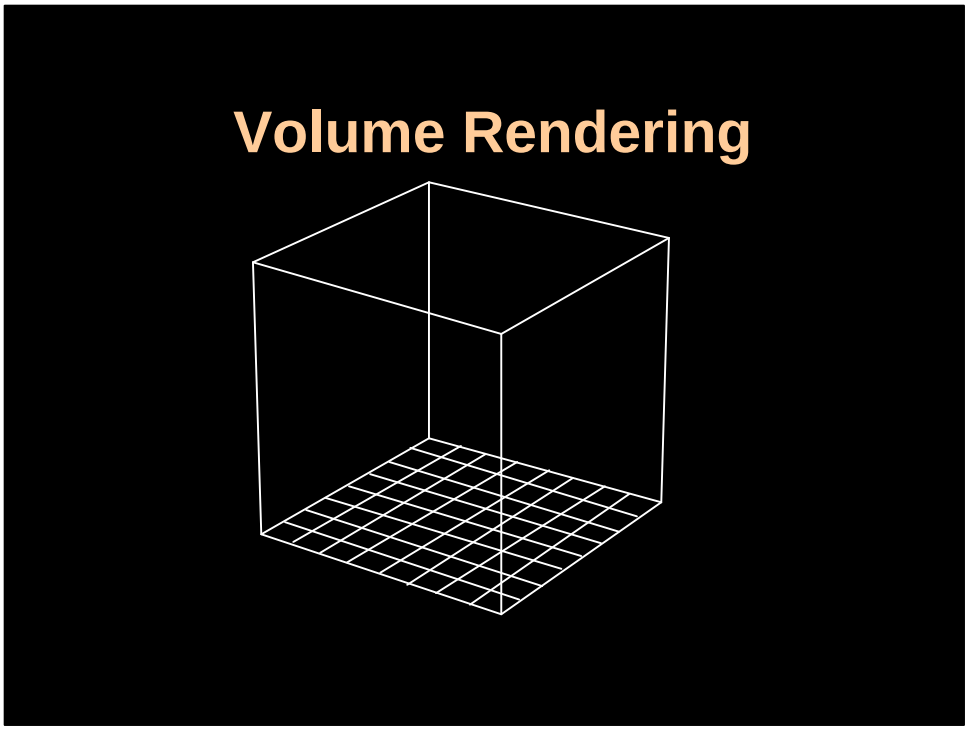
Here I show a demo of the simple solver where I interact in real time with a velocity field and switch back and force between the spatial and the Fourier domain.

3D Demos

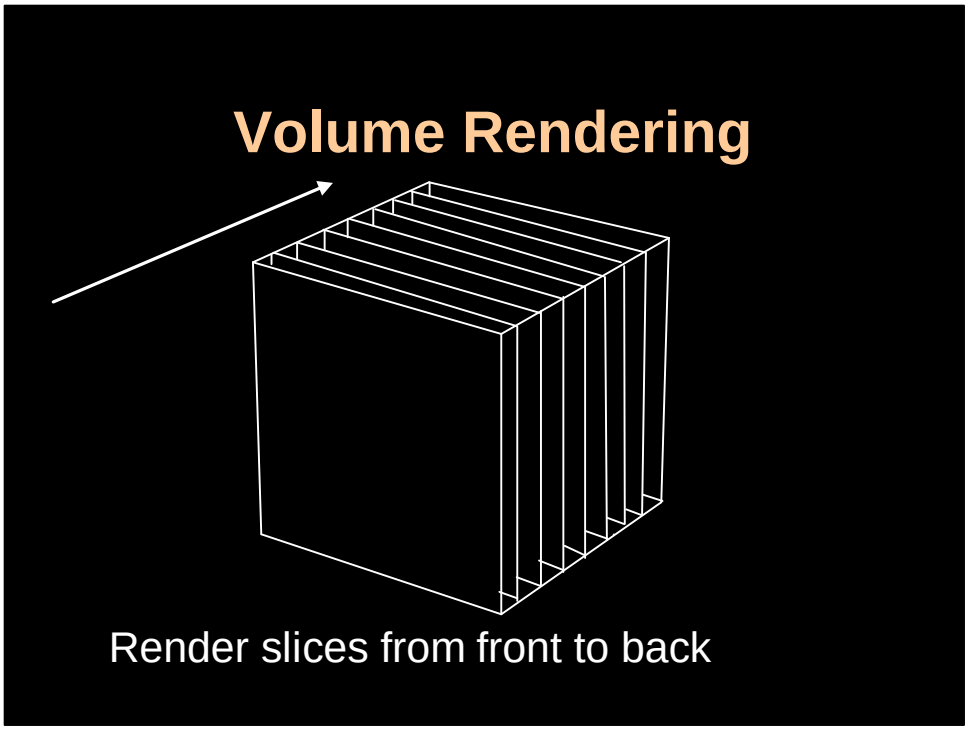
Move 3D solid texture coordinates

Interactive Volume Rendering

As I said at the beginning of the talk, although everything was explained in a 2D setting the algorithm easily extends to 3D by simply adding an extra index and extra for loops. One issue is however how we display the density since it is now a volume. For this we need a fast volume renderer.



Here is what a 3D grid looks like.



We do the rendering in hardware by sampling the volume along slices and then rendering them from front to back as semi-transparent quads. The slices are chosen to be aligned with the main axis most aligned with the view. This makes the coding much easier. It results in small popping artifacts in some cases. So don't use it in production but it allows someone to view the density in real-time. You can also self-shadow the volume by doing a shadow prepass from the light source and storing the accumulated transparencies. Essentially I use a 3D Bresenham with the origins at the center of the cells on the faces that are lit.

I now show various 3D demos which still run in real time. Showing the new hardware density rendering with self-shadowing. I also show densities which are textured using 3D texture maps. The Octane and the new nVidia cards have them in hardware, but there are ways to fake them with 2D texture maps, and in fact I have it working on my Dell 1GHz PIII laptop with a GeForce2 GO card in it. I also show some demos with boundaries where the smoke slides over them, etc. All this in real time of course.

Defeating Dissipation

Work with

Ronald Fedkiw & Henrik Wann Jensen

Stanford University

I will now talk about some recent work with Ron Fedkiw and Henrik Wenn Jensen both from Stanford University. As you could tell from the demos the motion is nice but tends to be dampened faster than actual flows. This is because to enforce stability additional damping is added to the flow. So the flows although they appear cool when stirred actually die out to fast. In this new work we address this problem and apply it specifically to smoke.

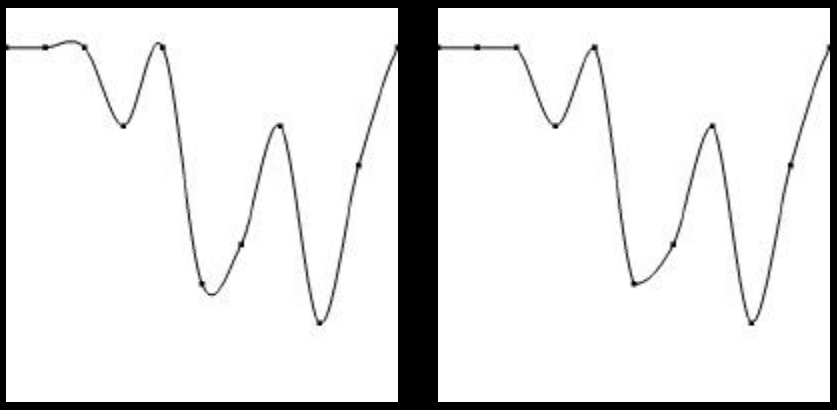
Higher Order Interpolation

Use Hermite interpolation instead of Linear interpolation.

The first technique is to improve the interpolation that is used during the self-advection step. We get higher accuracy with better interpolants such as the Hermite curve shown on the slide. The problem, however, is that these interpolants may under or overshoot the data. So we do not have the nice property anymore that the max of the new values is bounded by the max of the old values. And so potentially the simulation might blow up and we lose the nice property of stability.

Higher Order Interpolation

Force interpolant to be monotonic to avoid instabilities



To fix this problem we introduce some new splines which are guaranteed to not overshoot the data. The picture on left shows data interpolated with a standard Hermite spline. You can clearly see the overshoots and under shoots where the data isn't smooth, this is a basic behavior of any high order interpolant. The curve on the right shows our new interpolation scheme. Notice how there no over or undershoots and that the curve is discontinuous where it should be. Our solver with this interpolation scheme is stable for any time step.

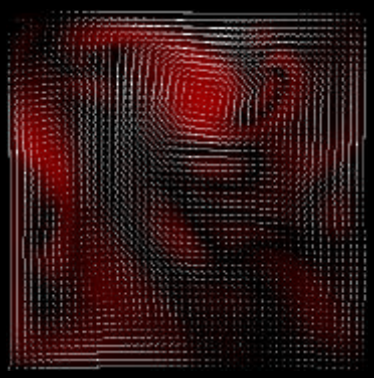
Higher Order Interpolation

linear

Hermite

Here are two animations where you can see the increase in precision using the new scheme. Of course the interpolation is more expensive, about 9 times more expensive than the linear interpolation. So the linear one is useful in pre-production phase, to get the motion basically right. While the more expensive solver can be useful at later stages to refine the motion.

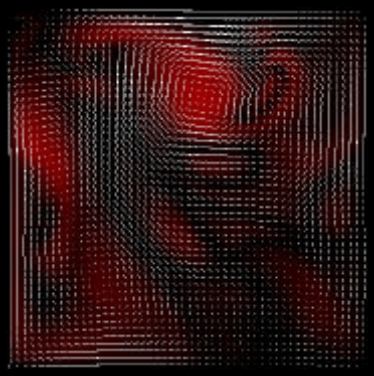
Vorticity Confinement



$$\omega = \nabla \times \mathbf{u}$$

So that's one way of reducing the numerical dissipation of the basic Stable Fluids algorithm. The next technique is very cool. The basic idea is to reinject the energy that is dissipated back into the flow through a clever force field. Instead of looking at the velocity we will consider the vorticity field which is defined as the curl of the velocity. Again this is a vector field. In 2D this vector always points out of the fluid plane and the vorticity can be treated as a scalar field. On the slide I show the velocity field and the corresponding magnitude of the vorticity in red. Notice how the field is strongest where the rotation is largest. Vorticity is thus a measure of how much the flow rotates. Clearly for turbulent smoke we want the vorticity to be strong and localized. This will result in nice looking swirling smoke fields.

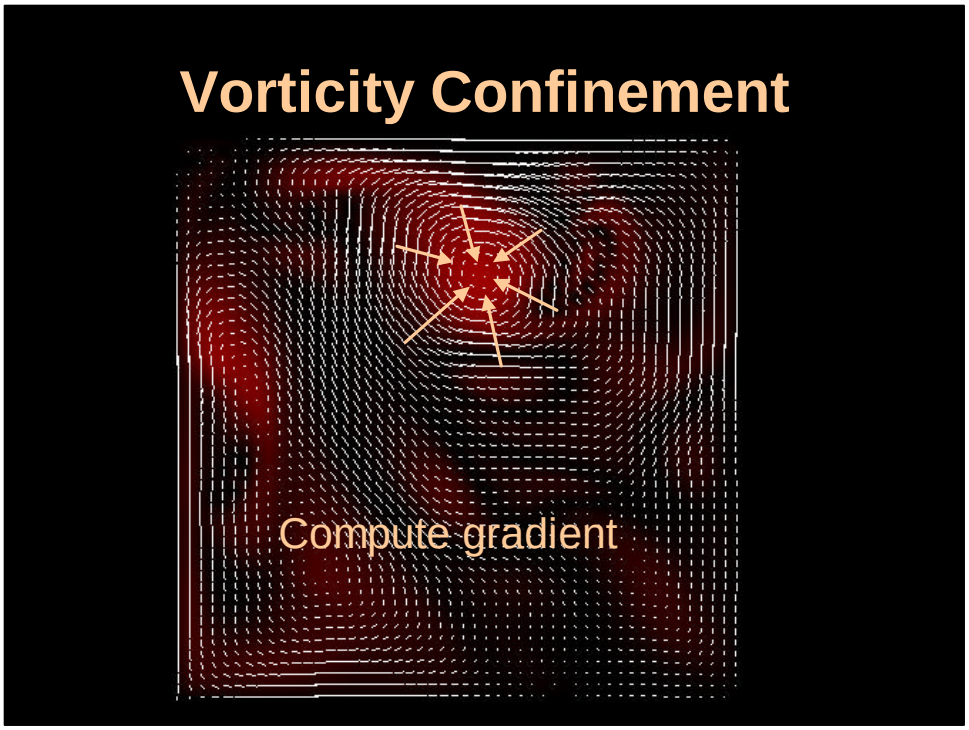
Vorticity Confinement



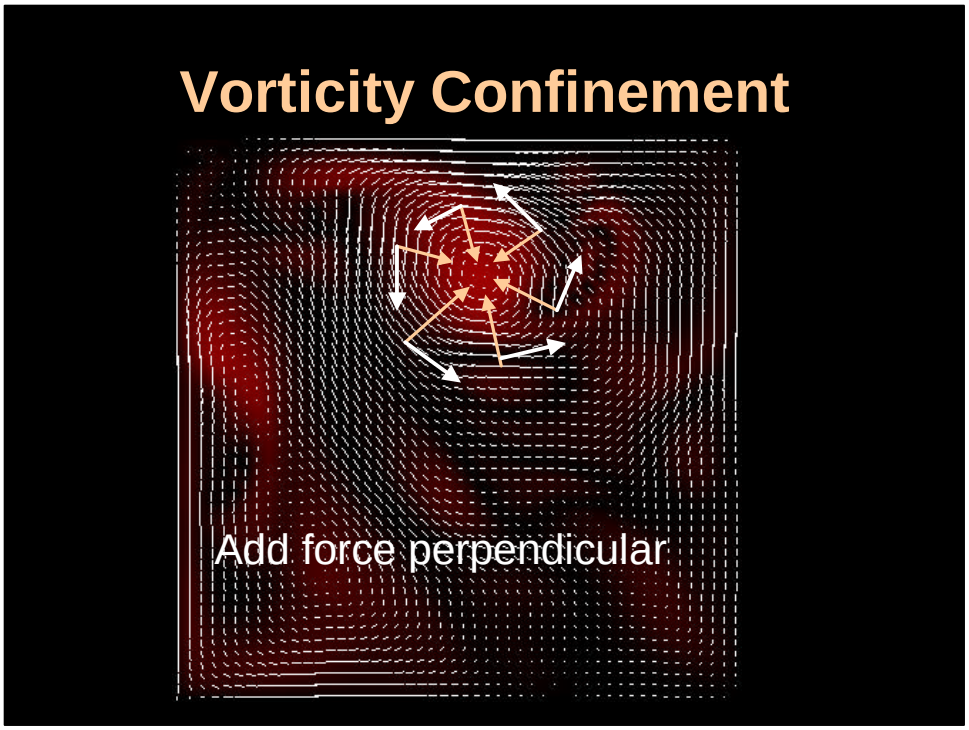
Basic idea :Increase vorticity

John Steinhoff 1987 (Flow Analysis)

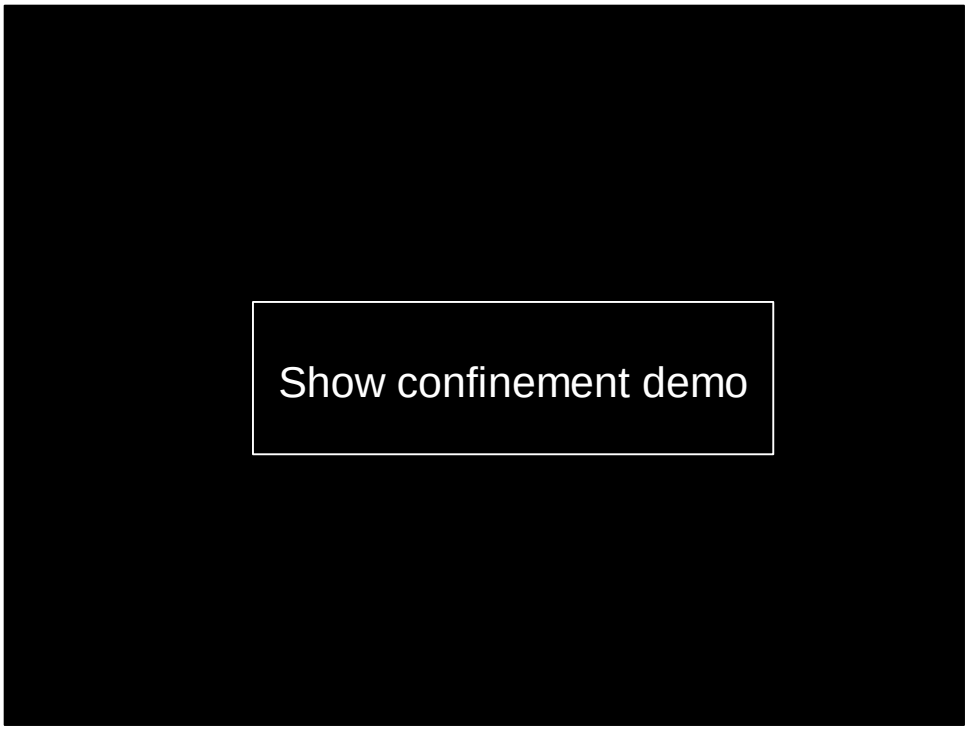
So the idea is to keep the vorticity alive using an external force. The basic idea is due to a brilliant insight from John Steinhoff that he had some ten years ago. John is a physicist who has his own company called Flow Analysis, he has many other cool ideas that are implemented in his commercial software.



The first step in computing the confinement force, is to compute the gradient of the vorticity. So this field points inwards to the vortex center.

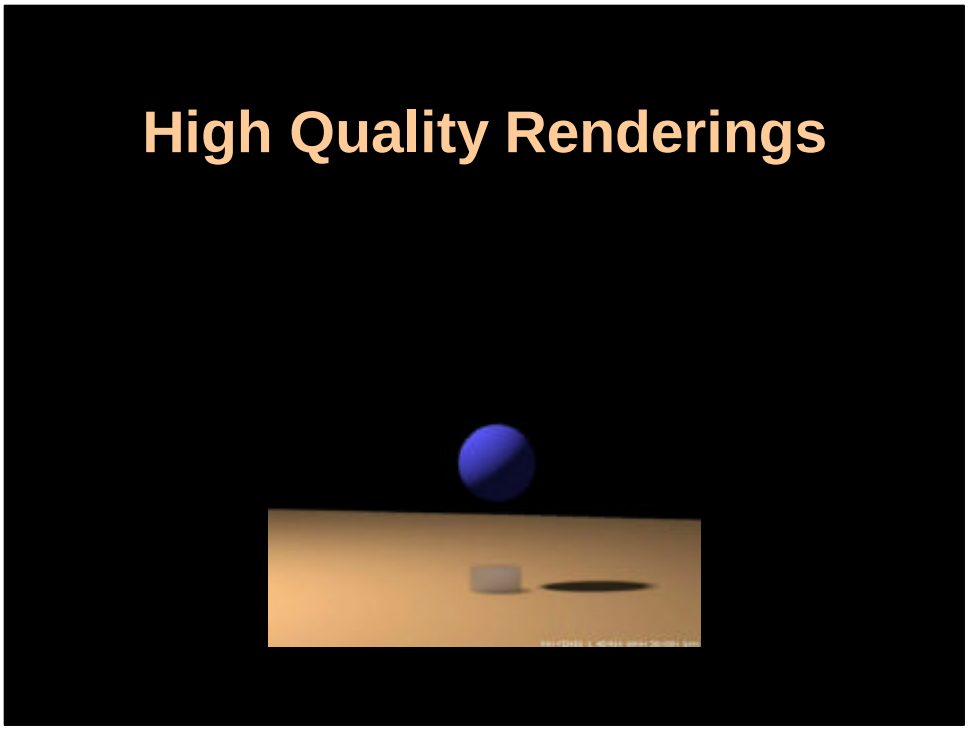


The next step is to add a force perpendicular to the gradient field. This will tend to keep the vortices alive and well localized.

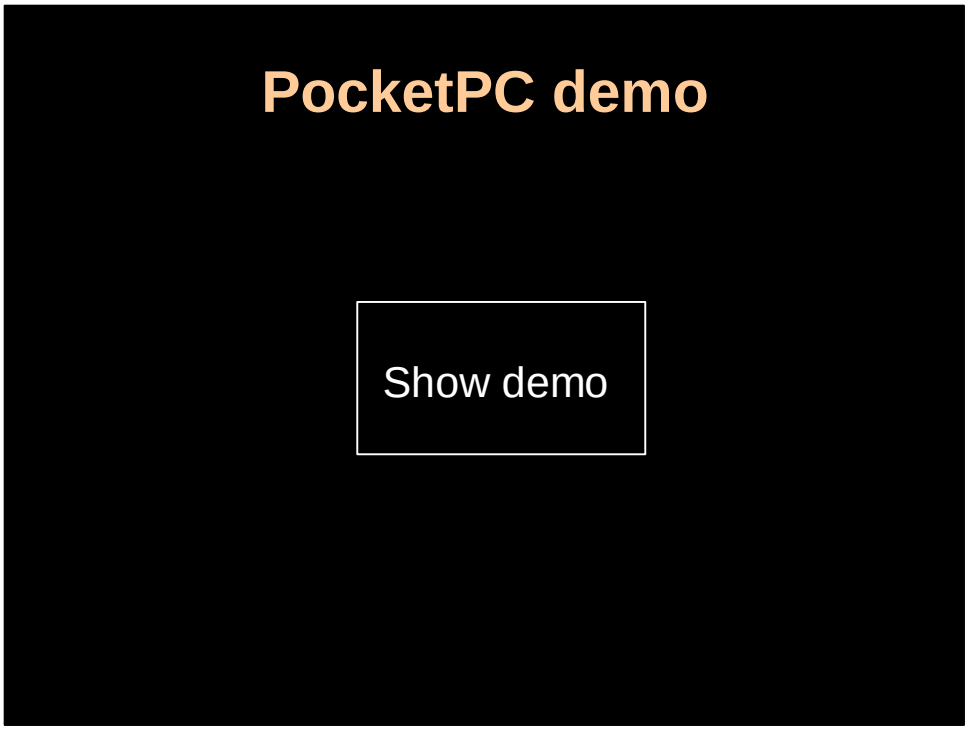


In this demo I do a side by side comparison of the solver with and without the confinement force. This is a real time demo and I display the vorticity field and also show how a density field has more detail with the confinement force added in. I also show some demos of a wind tunnel, in it you get the typical Karman vortex street pattern, this puts the Reynold's number of these flows at roughly 200. I also show 3D demos with the vorticity confinement, with object interacting with smoke, such as spheres and cloth.

This is a high quality animation rendered using Jensen’s photon map.



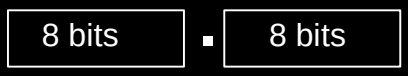
Here is another one with a sphere



Finally the last demo is where I show a 2D solver that runs on my Pocket PC (Compaq iPaq H3600). I did all the rendering myself directly to screen space using the GAPI gamer interface for the Pocket PC. And I ported my entire solver to fixed point arithmetic since the CPU in the PocketPC (Intel StrongARM) does not have hardware support for floats.

PocketPC demo

Fixed point math:




```
#define freal short     // 16 bits

#define X1 (1<<8)
#define I2X(i) ((i)<<8)
#define X2I(x) ((x)>>8)
#define F2X(f) ((f)*X1)
#define X2F(x) ((float)(x)/(float)X1)
#define XM(x,y) ((freal)(((long)(x)*(long)(y))>>8))
#define XD(x,y) ((freal)(((long)(x))<<8)/(long)(y)))

x = a*(b/c)       x = XM(a,XD(b,c))
```

In fixed point math all real are represented as 16 bit integers: 8 bit for the integer part and 8 for the fractional part. It is really easy to implement the equivalent operations for fixed point reals. All you have to do is define the following defines and use them instead of the usual operations. You have to be careful with certain algorithms that do not work well with fixed point due to underflows and overflows. For example the conjugate gradient fails miserably, so you are better off using a simple relaxation technique like Gauss-Seidel.

Future Work

- Handle free boundaries (water)
- Parallel implementation (in progress)
- Adaptive grids (in progress)
- “Smarter” texture maps

Here some of the stuff I am working on. I would like to have a real time simulation of water. This is harder as the boundary conditions change over time (at the water surface). I hope that I can reuse some of the ideas presented here and apply them to water. Also I would like to parallelize my code. Of course I do not have the hardware to test my ideas. That’s a problem. However, I was able to parallelize the code on my dual PIII at work. I simply feed the velcoity solver to one CPU and the density solver (+texture coord or temperature) to the second CPU, this gave some dramatic speedups. Also we are working on adaptive grids. The grid can be adapted to the region of interest. Imagine a thin trail of smoke, than it would be a waste to have a uniform grid for all spatial directions. Also I am looking into designing smarter texture maps to model things like pyroclastic flows. These flows are so highly detailed that they are beyond grid technology at this point, the same with clouds. So the idea is to add a smart texture map with billowing motion which is driven by a coarser simulation.

Thank you for your attention.

Any questions ?