

Stable Fluids

Jos Stam

Alias|wavefront

Seattle, USA

Fluids in Computer Graphics

- Fast.
- Looks good.
- Easy to 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

Previous Work (computer graphics)

Two dimensions:

- Yaeger & Upson 86 + Gamito et al. 95 (vortex blobs)
- Chen et al. 97 (explicit in time, finite differences)

Three-dimensions:

- Kajiya & Von Herzen 84 (very coarse grids)
- Foster & Metaxas 97 (explicit in time, finite differences)

unstable

Inaccurate schemes can be useful

Main Contribution

Stable Navier-Stokes solver

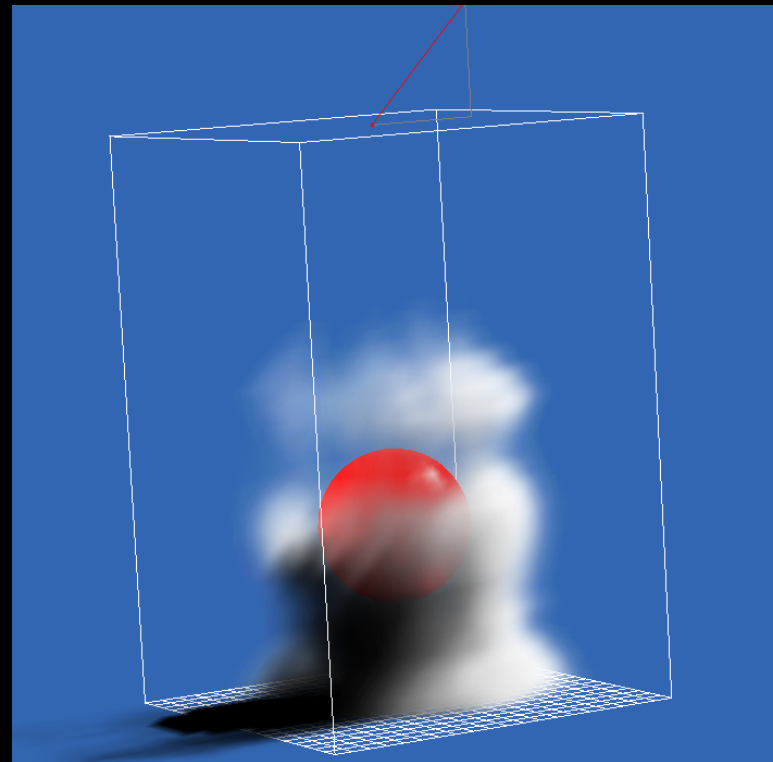
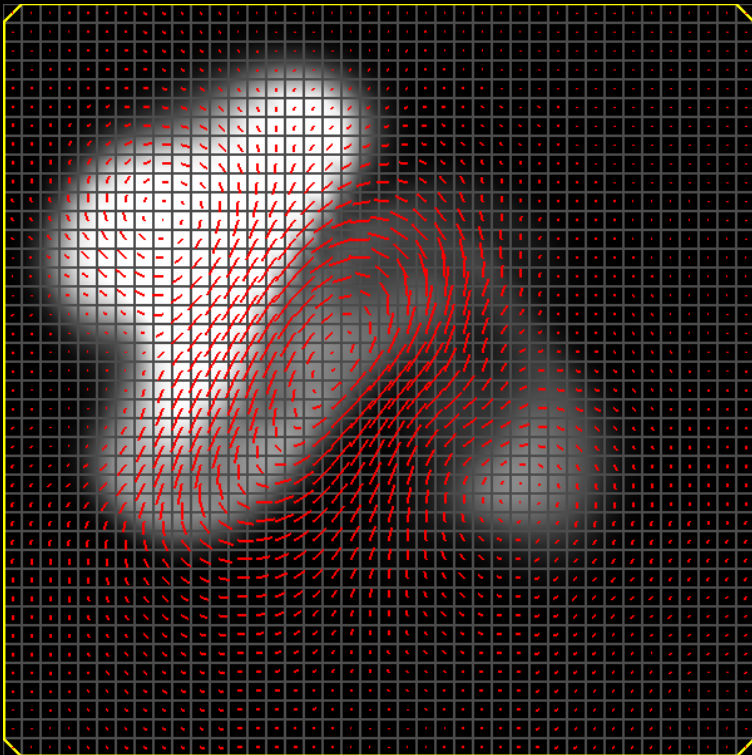
Any time step can be used

Bigger time steps = faster simulations

NOT accurate

Application

Use velocity to move densities:



Application

Use velocity to move densities:

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

Equations

Evolution of density (assume velocity known)

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

Over a time step...

Equations

Evolution of density (assume velocity known)

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

Density changes in the direction of the flow

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

Equations

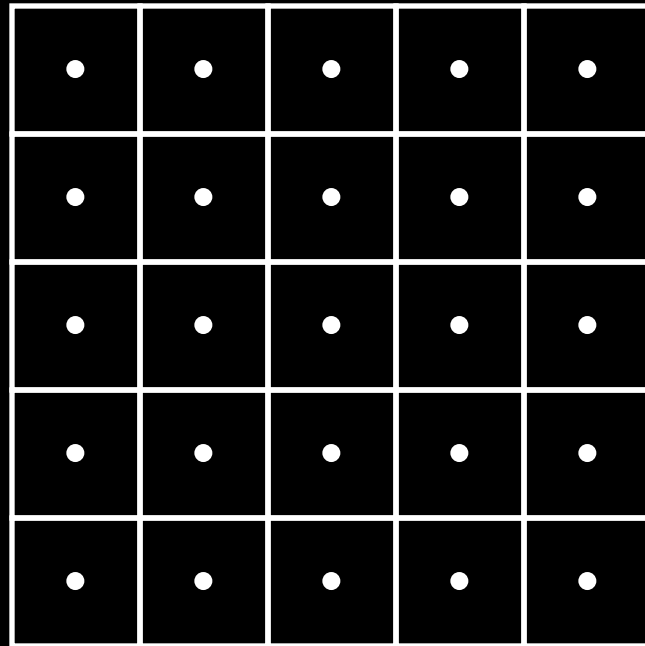
Evolution of density (assume velocity known)

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

Increases due to sources from the UI

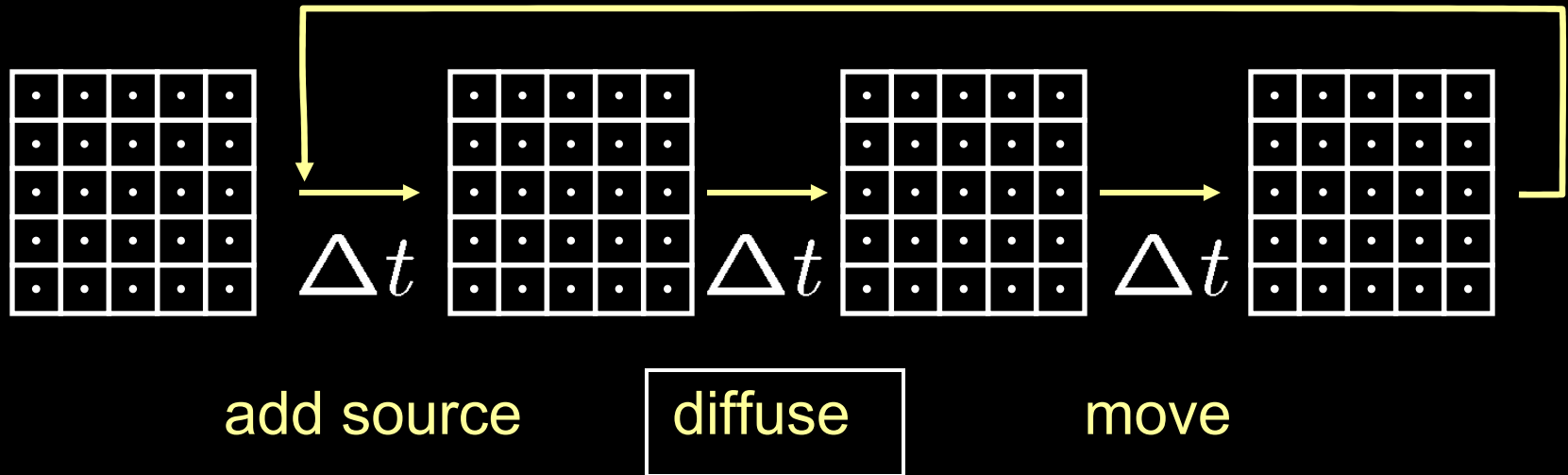
Algorithm

Subdivide space into voxels



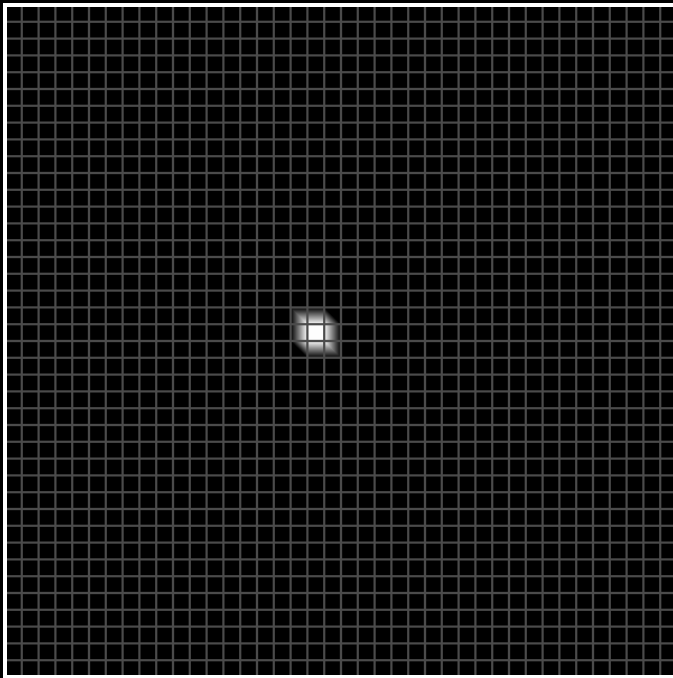
Velocity + density defined in the center of each voxel

Algorithm



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

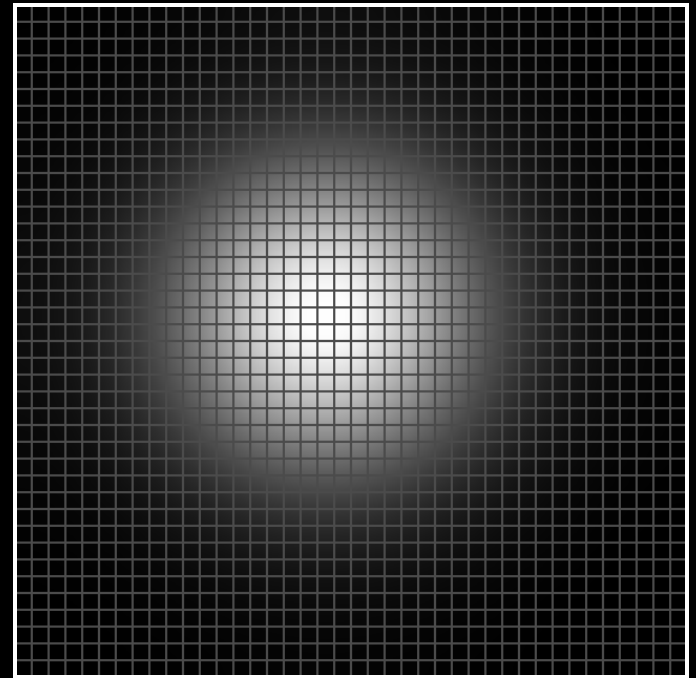
Diffusing Densities



D^n

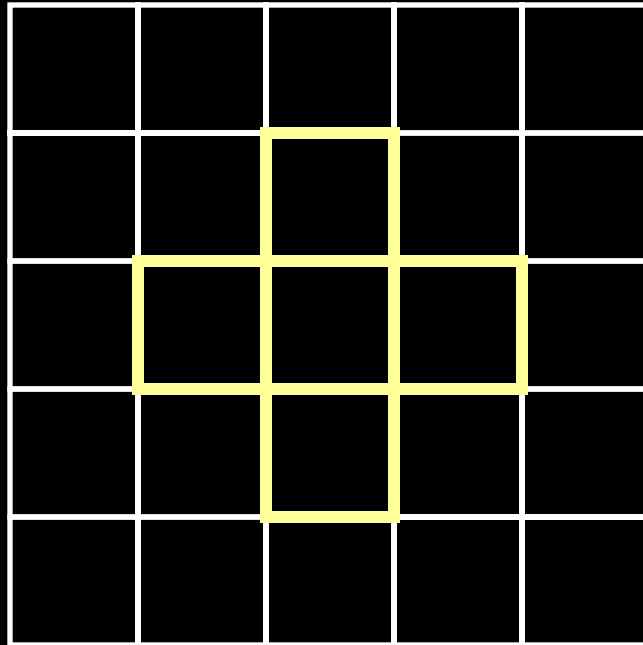


dt



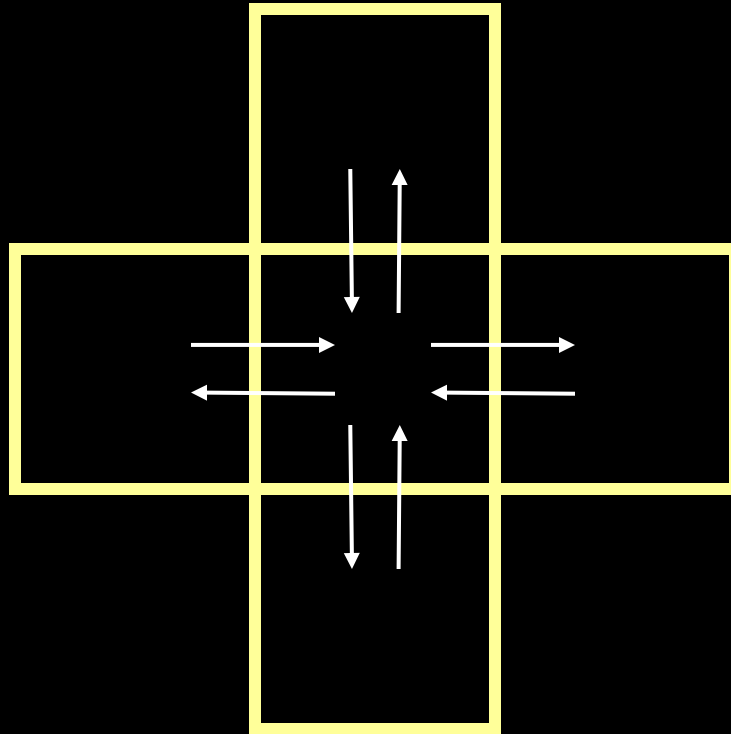
D^{n+1}

Diffusing Densities



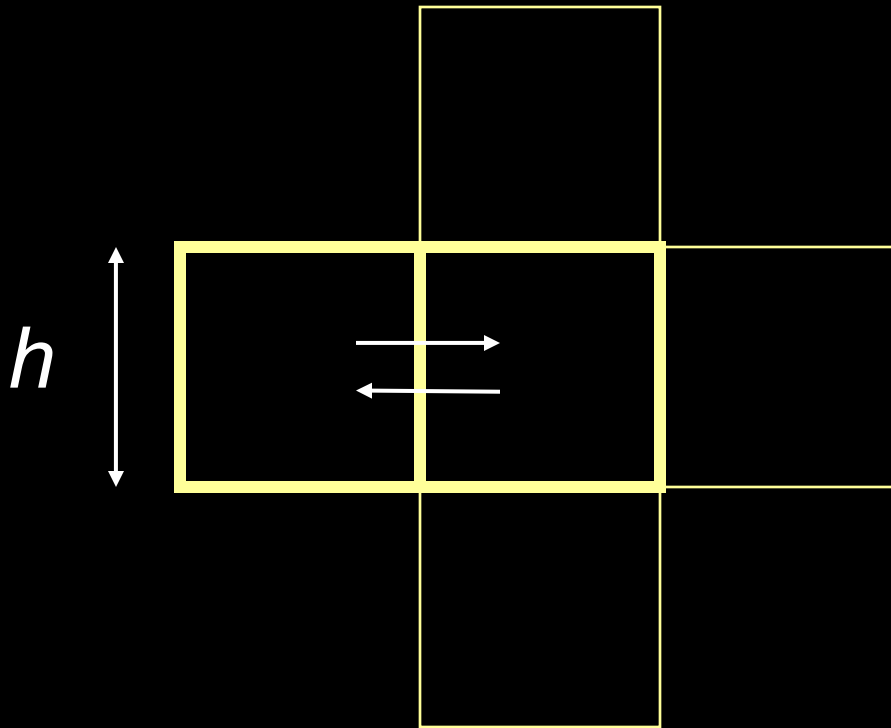
Exchange of density between neighbors

Diffusing Densities



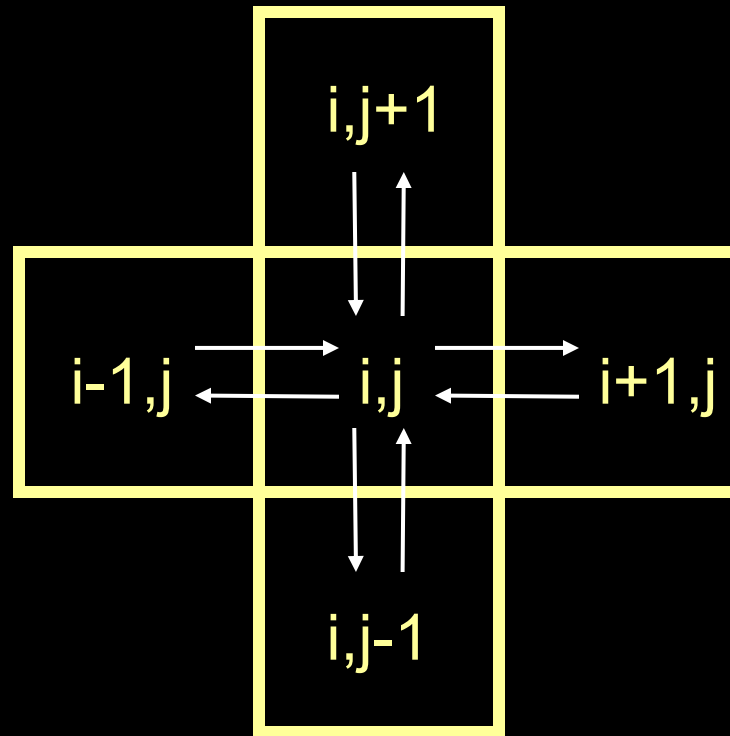
Exchange of density between neighbors

Diffusing Densities



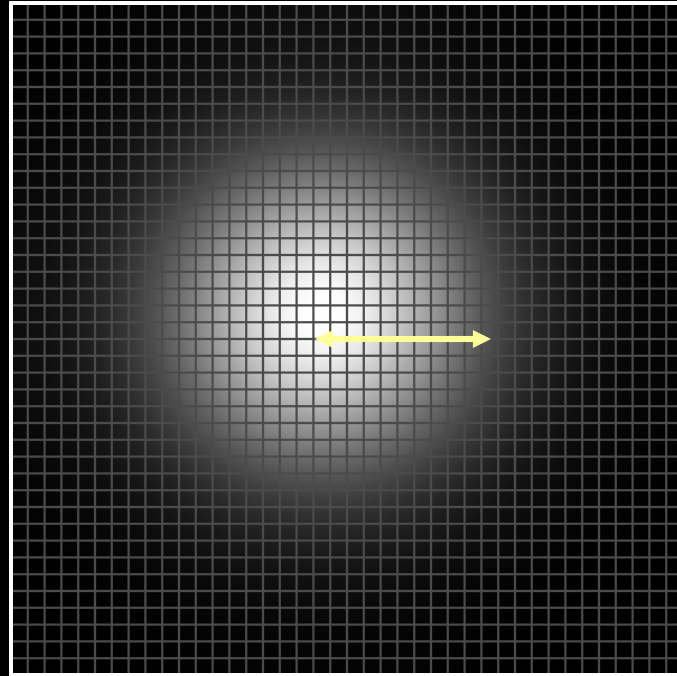
$$\begin{aligned} \text{Change} &= \text{density flux in} - \text{density flux out} \\ &= k dt (\text{neighbor} - \text{center}) / h^2 \end{aligned}$$

Diffusing Densities



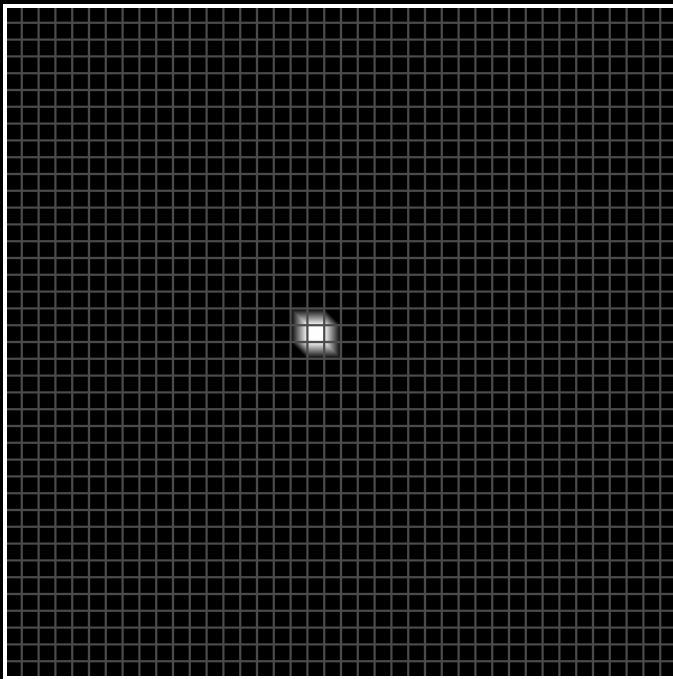
$$D^{n+1}_{i,j} = D^n_{i,j} + k dt (D^n_{i-1,j} + D^n_{i+1,j} + D^n_{i,j-1} + D^n_{i,j+1} - 4D^n_{i,j})/h^2$$

Diffusing Densities



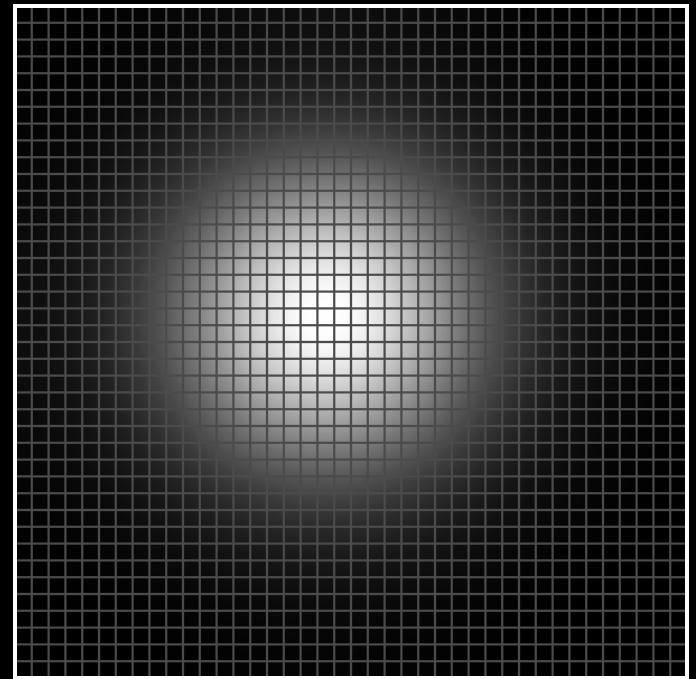
Unstable when $k dt / h^2 > 1/2$

Diffusing Densities



D^n

←
 $- dt$



D^{n+1}

Find densities which when diffused backward in time give the original densities.

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

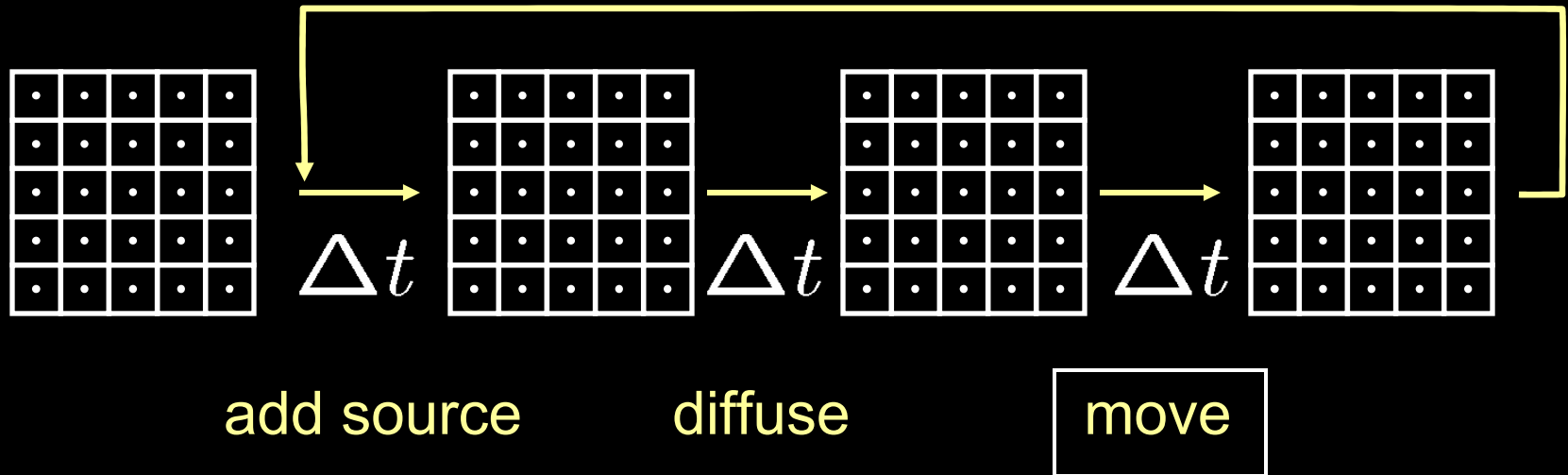
-> requires fast linear solver

Diffusing Densities

Linear solvers:

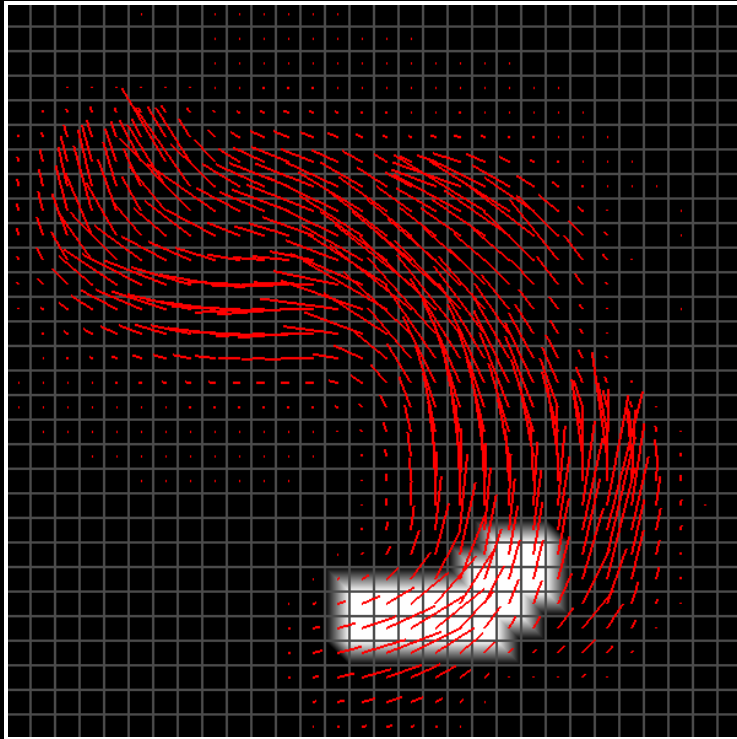
Name	Cost	Comments
Gaussian elimination	N^3	Use only for very small N (test code)
Jacobi/SOR relaxation	N^2	Easy to code but slow
FFT/cyclical reduction	$N \log N$	Use when no internal boundaries
Conjugate gradient	$N^{1.5}$	Use when internal boundaries
Multi-grid	N	Slower than FFT in practice. Hard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

Algorithm

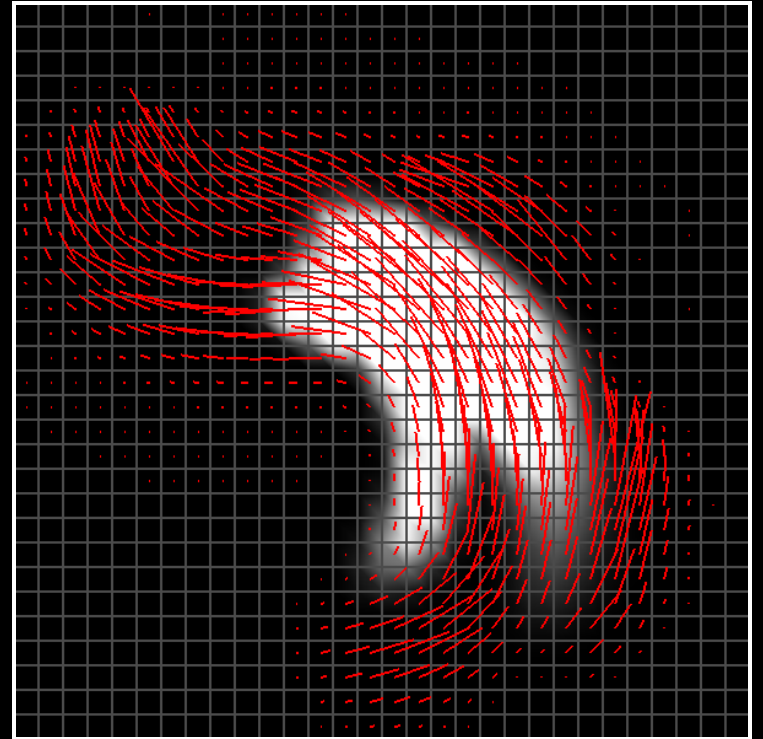


$$\frac{\partial \rho}{\partial t} = - (\mathbf{u} \cdot \nabla) \rho + \kappa \nabla^2 \rho + S$$

Moving Densities



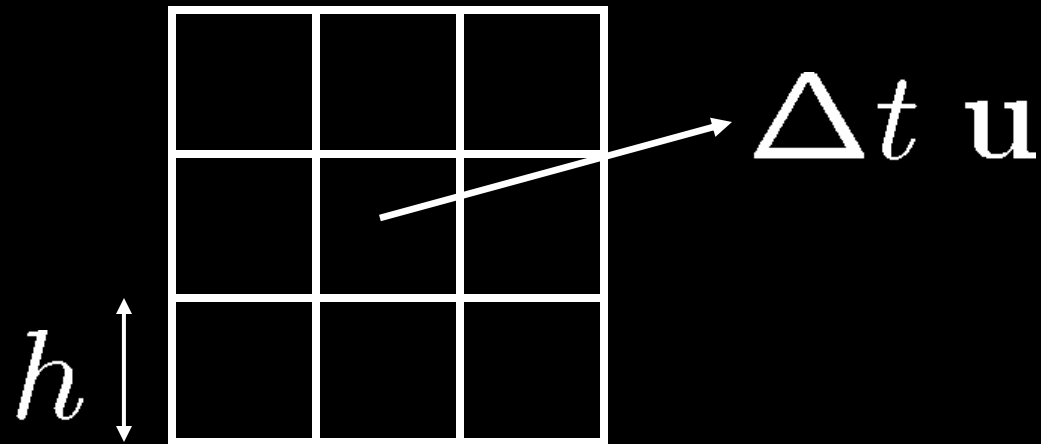
dt



Velocity known

Moving Densities

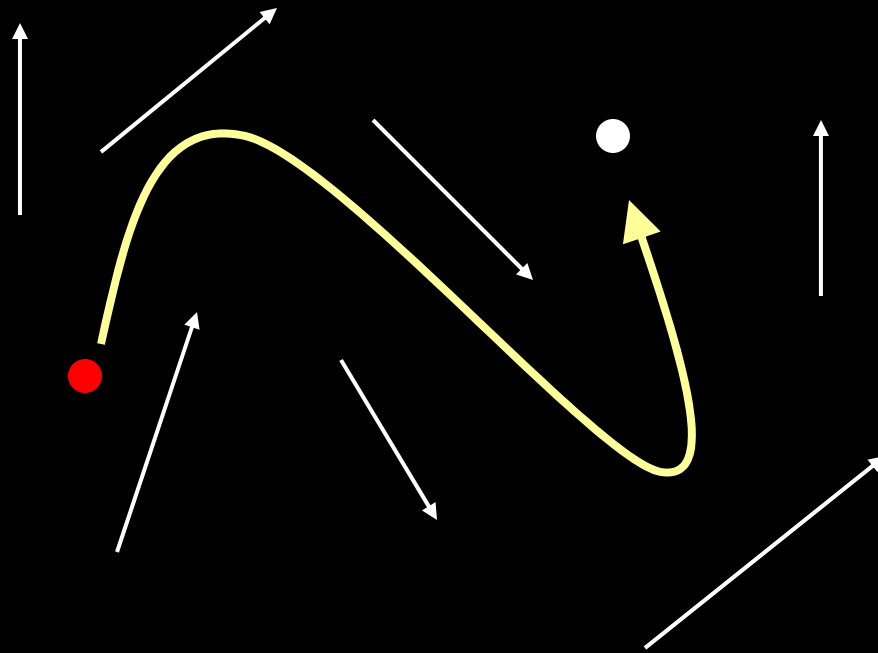
Finite Differences: transfer only between neighbors



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

Moving Densities

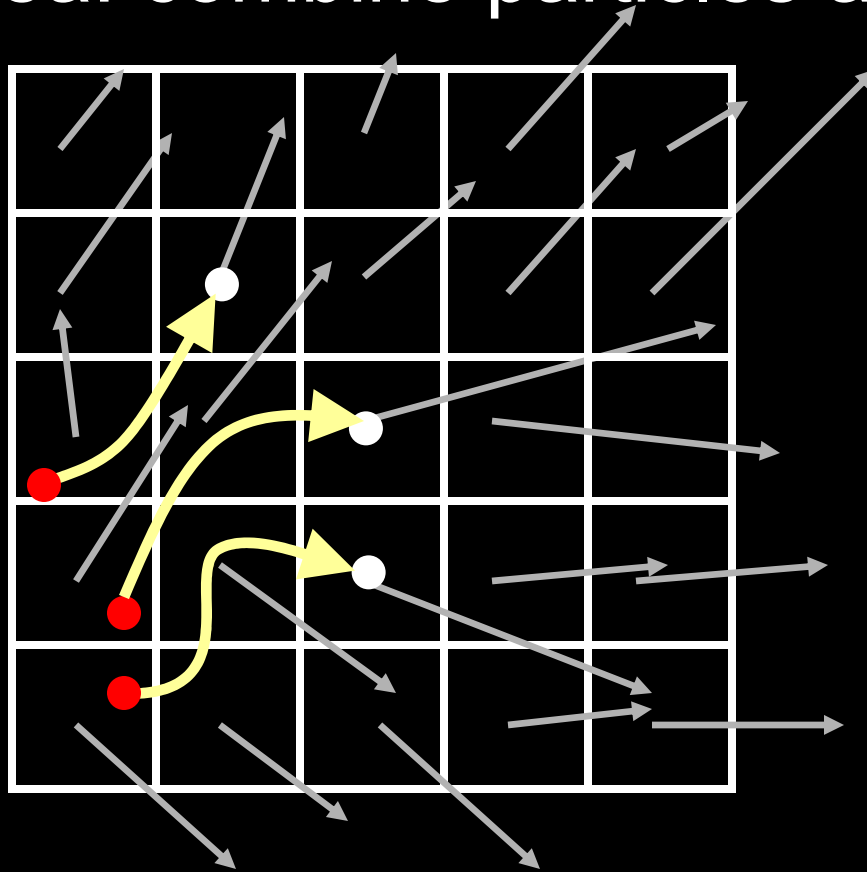
Easy if density defined on particles



Any time step ok

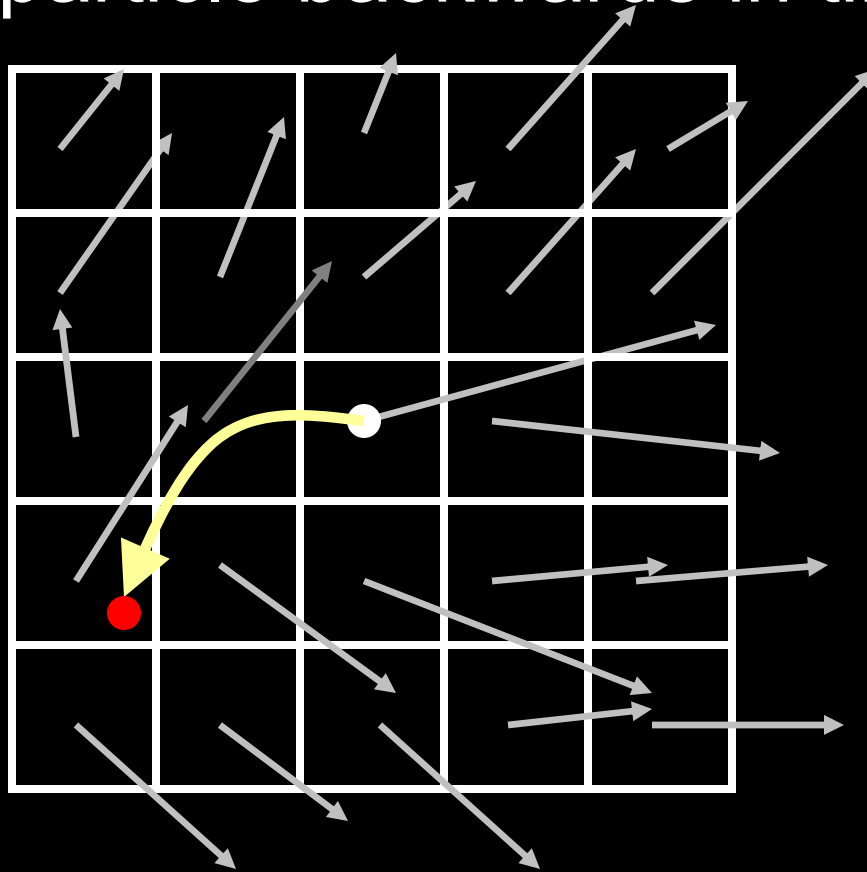
Moving Densities

Key Idea: combine particles and grids



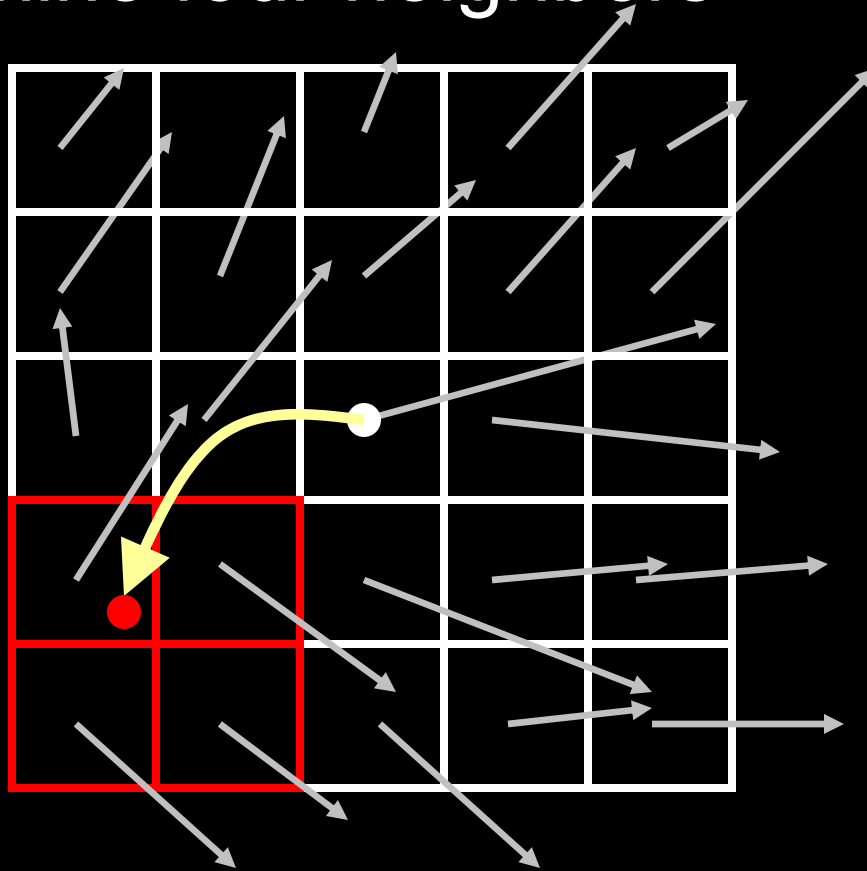
Moving Densities

Trace particle backwards in time



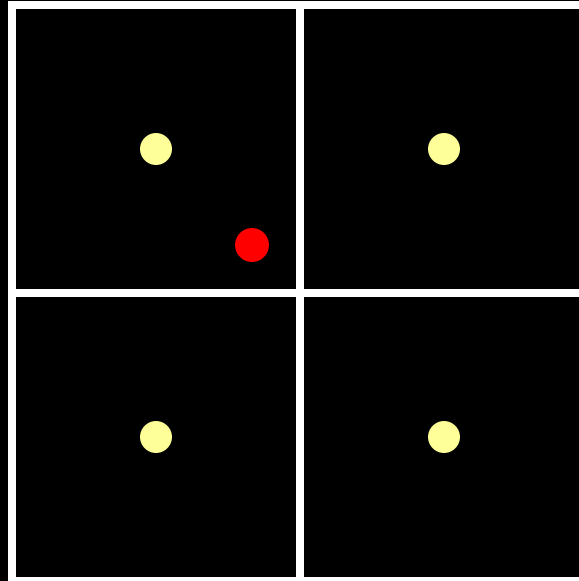
Moving Densities

Determine four neighbors



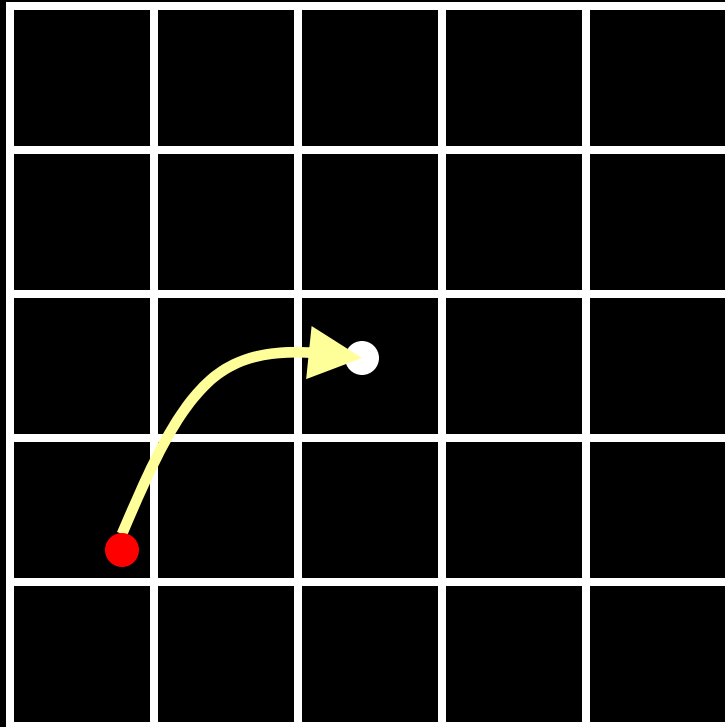
Moving Densities

Interpolate the density at new location



Moving Densities

Set interpolated density at grid location



Requires two grids

Moving Densities

This scheme is unconditionally stable:

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

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

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

→ density is always bounded

Computing Velocities

Computing Velocities

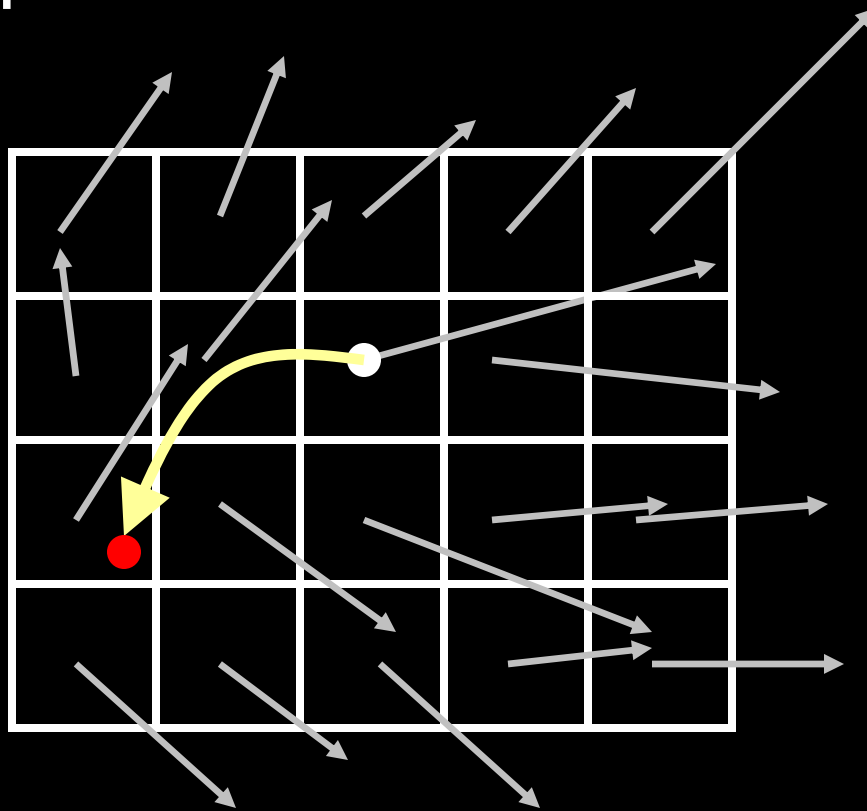
$$\frac{\partial \rho}{\partial t} = - (\mathbf{u} \cdot \nabla) \rho + \kappa \nabla^2 \rho + S$$

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

Velocity is moved by itself

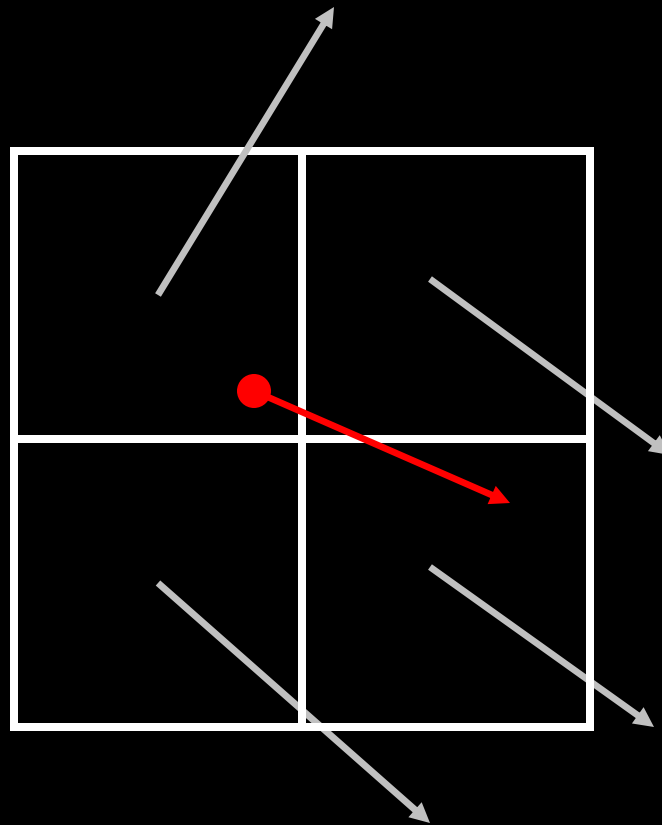
Moving Velocity

Trace particle backwards in time



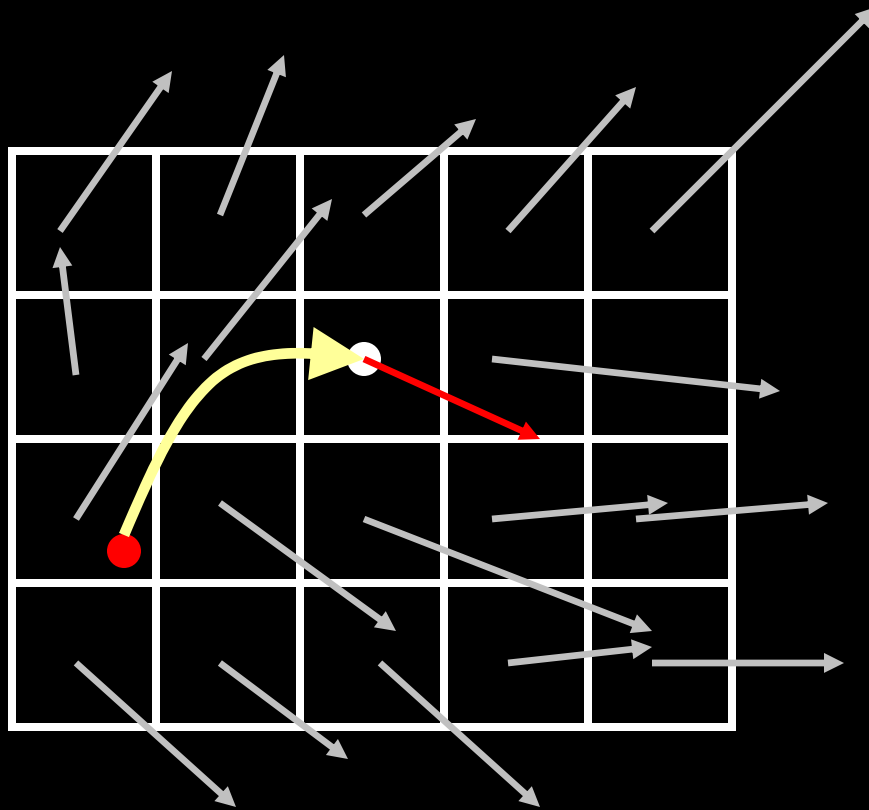
Moving Velocity

Interpolate the velocity at new location



Moving Velocity

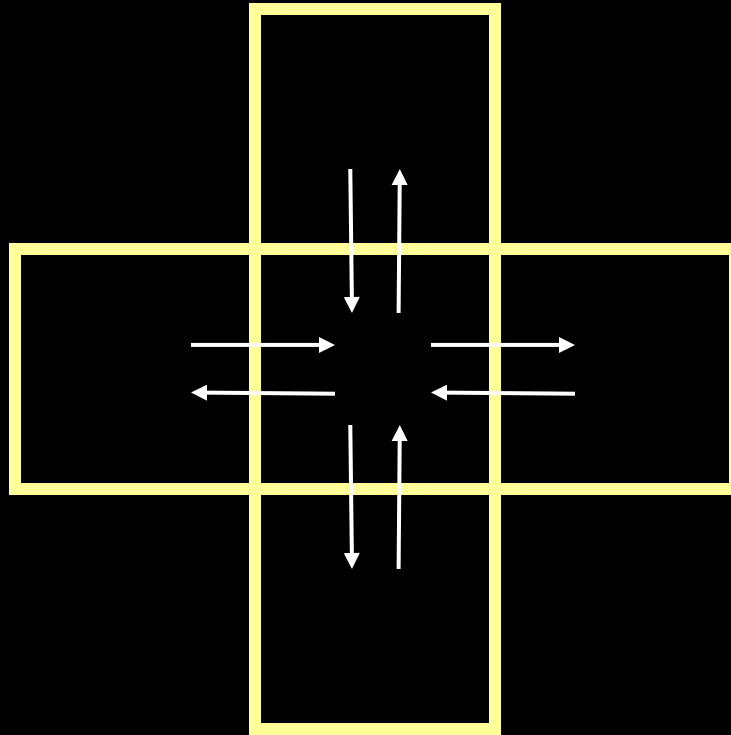
Set interpolated velocity at grid location



Requires two grids

Conservation of Mass

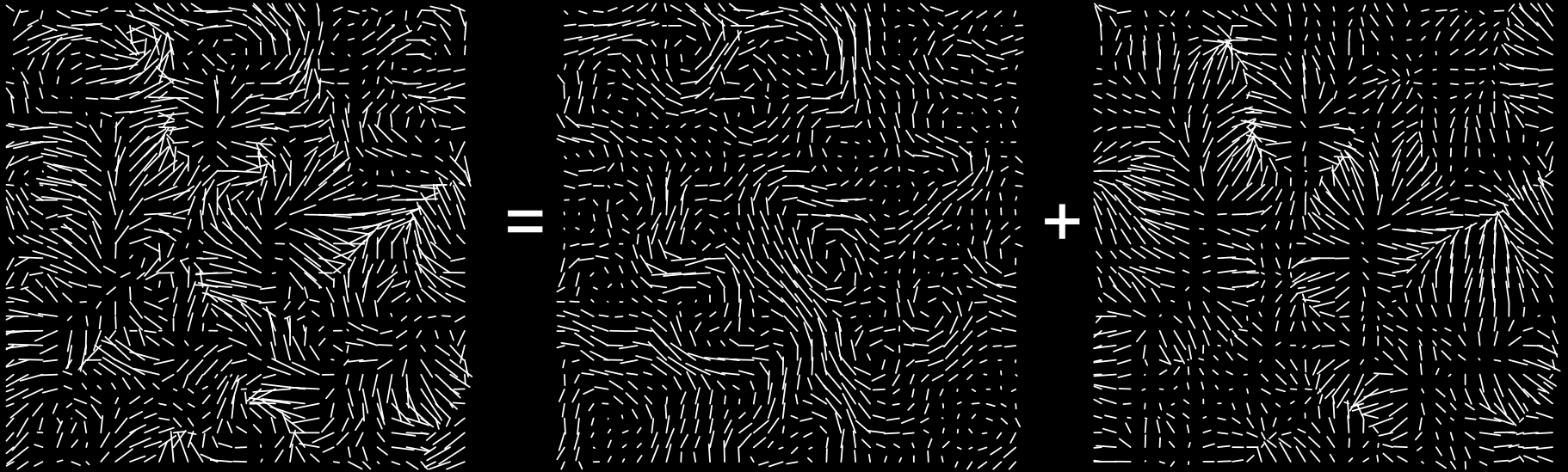
Conservation of Mass



Flow into cell = Flow out of the cell

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

Conservation of Mass



Our field

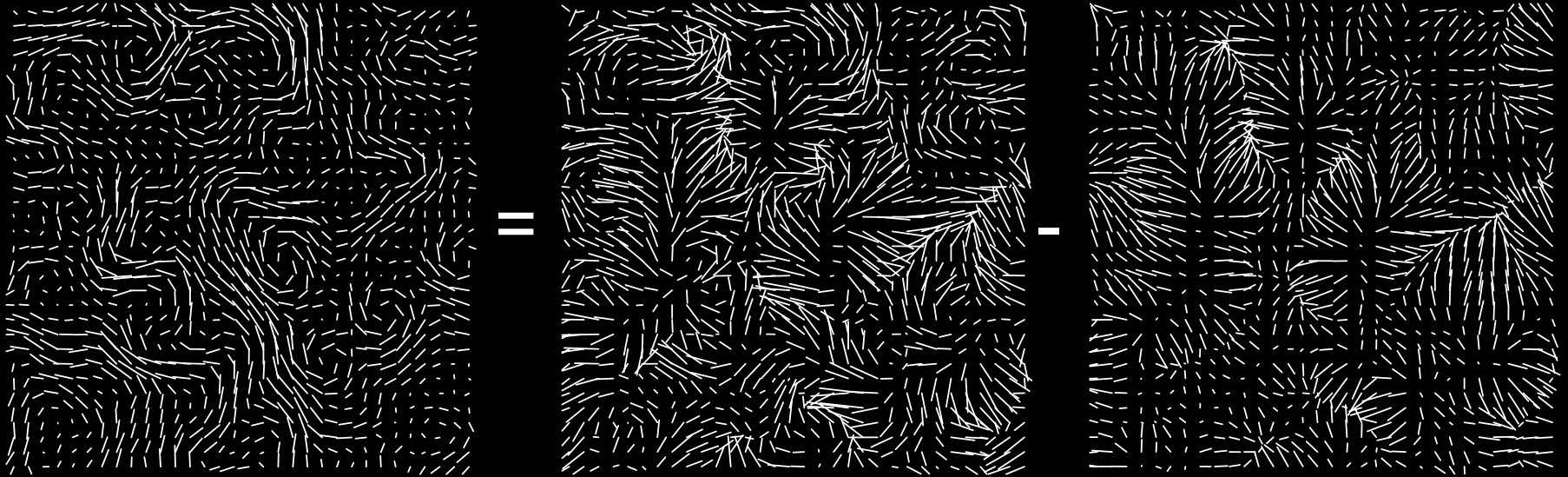
=

mass conserving +

gradient

Hodge decomposition

Conservation of Mass



Mass conserving = our field - gradient

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

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:

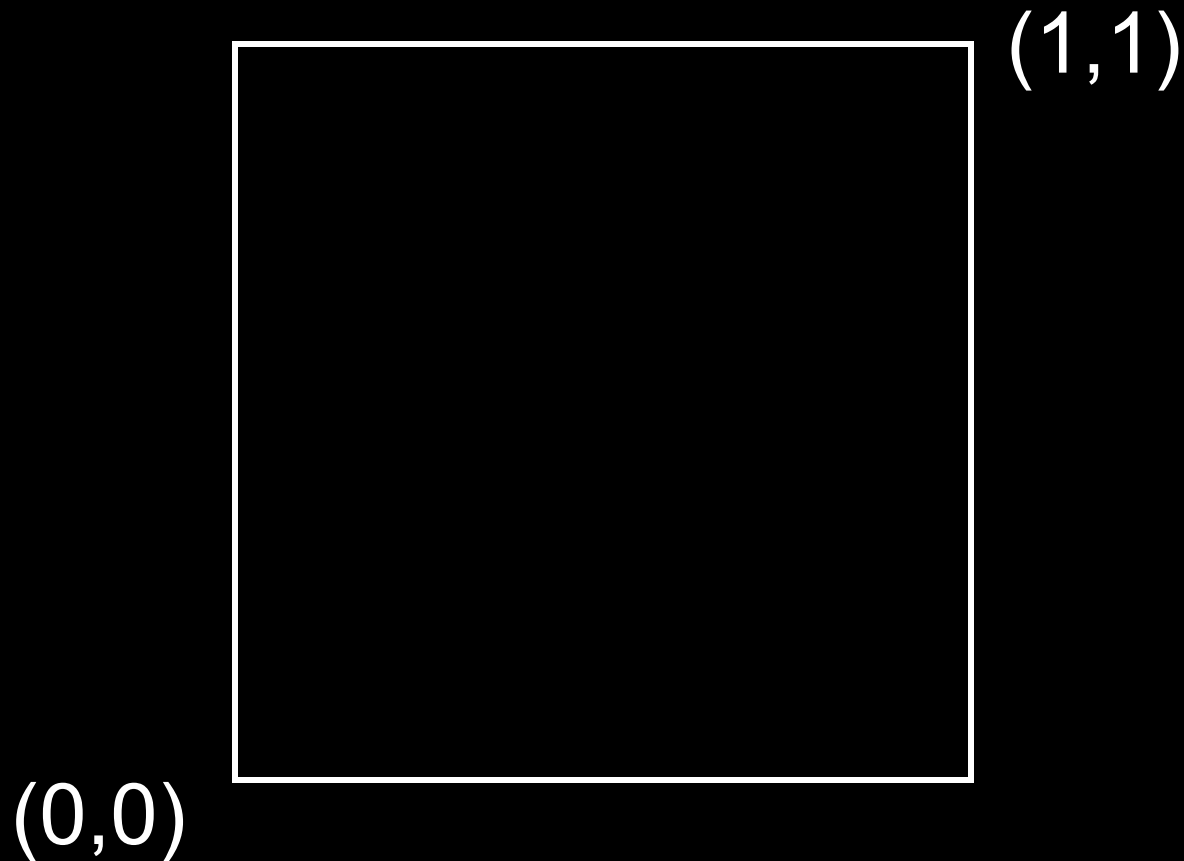
Particle tracer + grid interpolator

Linear solver (FISHPAK or CG)

Show 2D demo

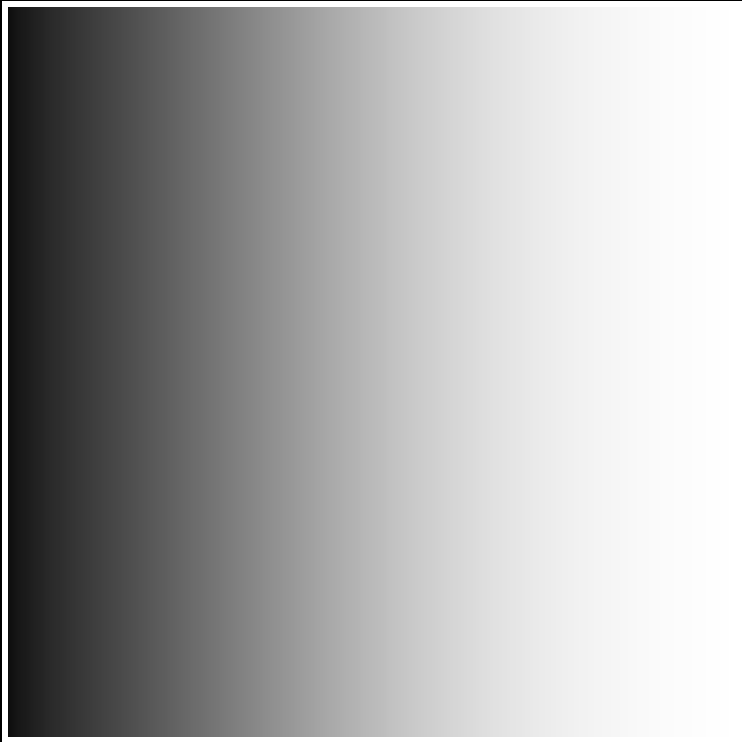
Liquid Textures

Animate texture coordinates

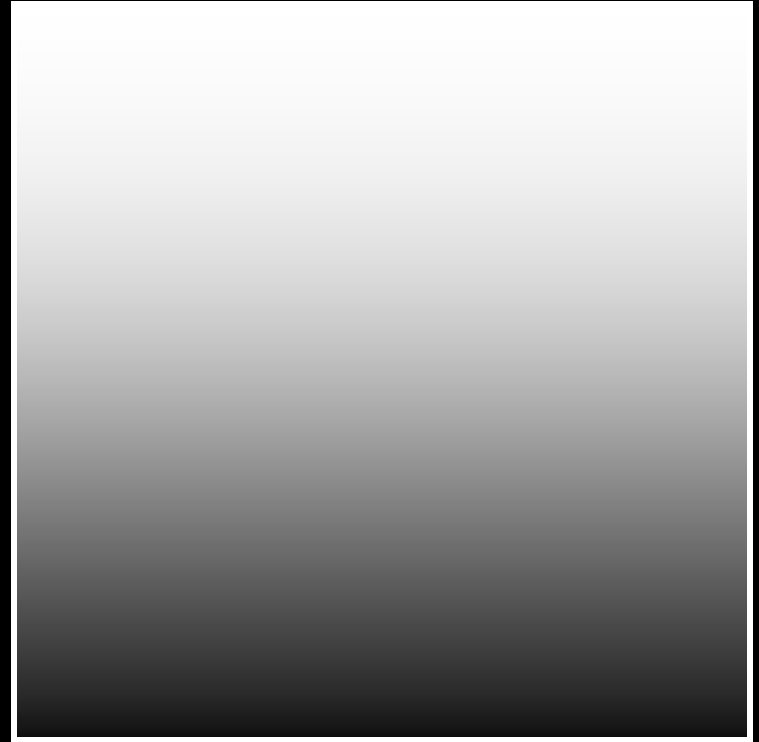


Liquid Textures

Treat texture coordinates as densities

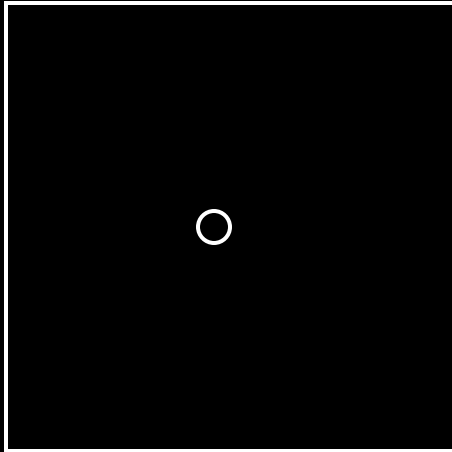


U-coordinate

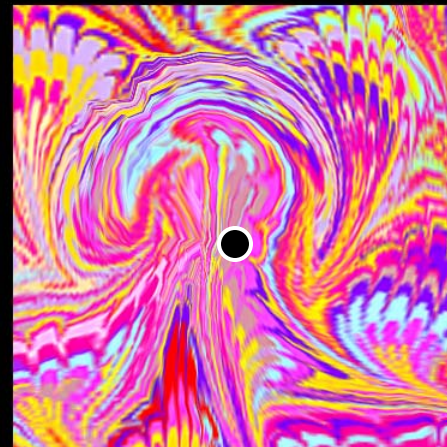
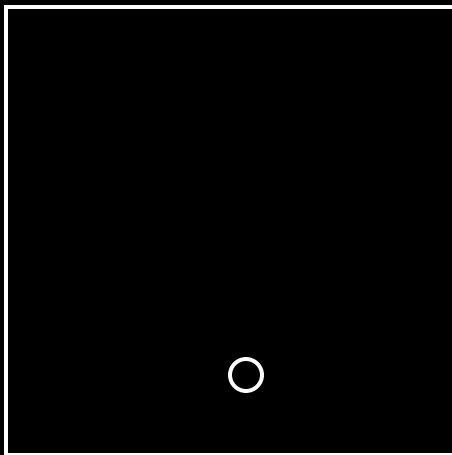


V-coordinate

Liquid Textures



(0.5,0.5)

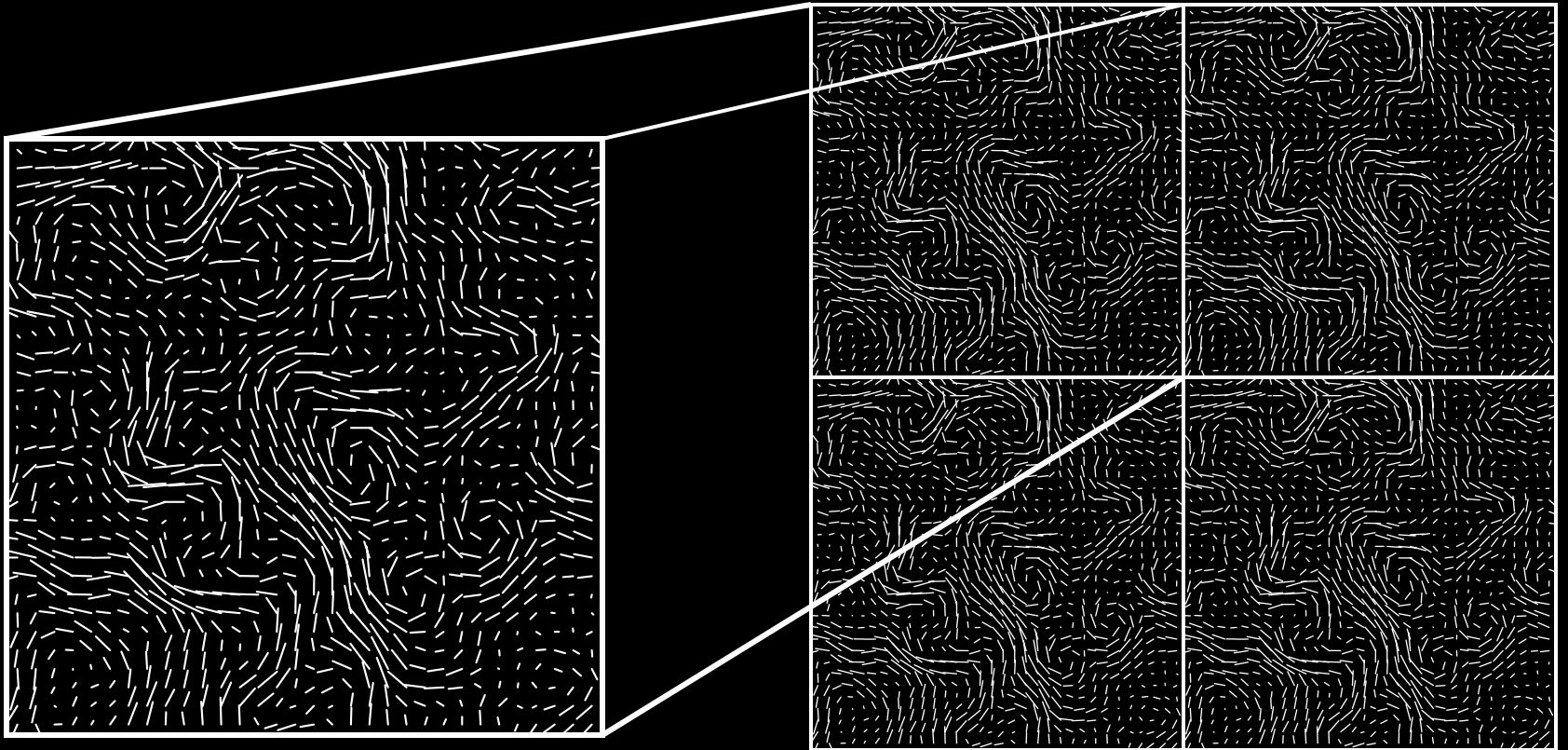


(0.2,0.52)

Show 2D texture demo

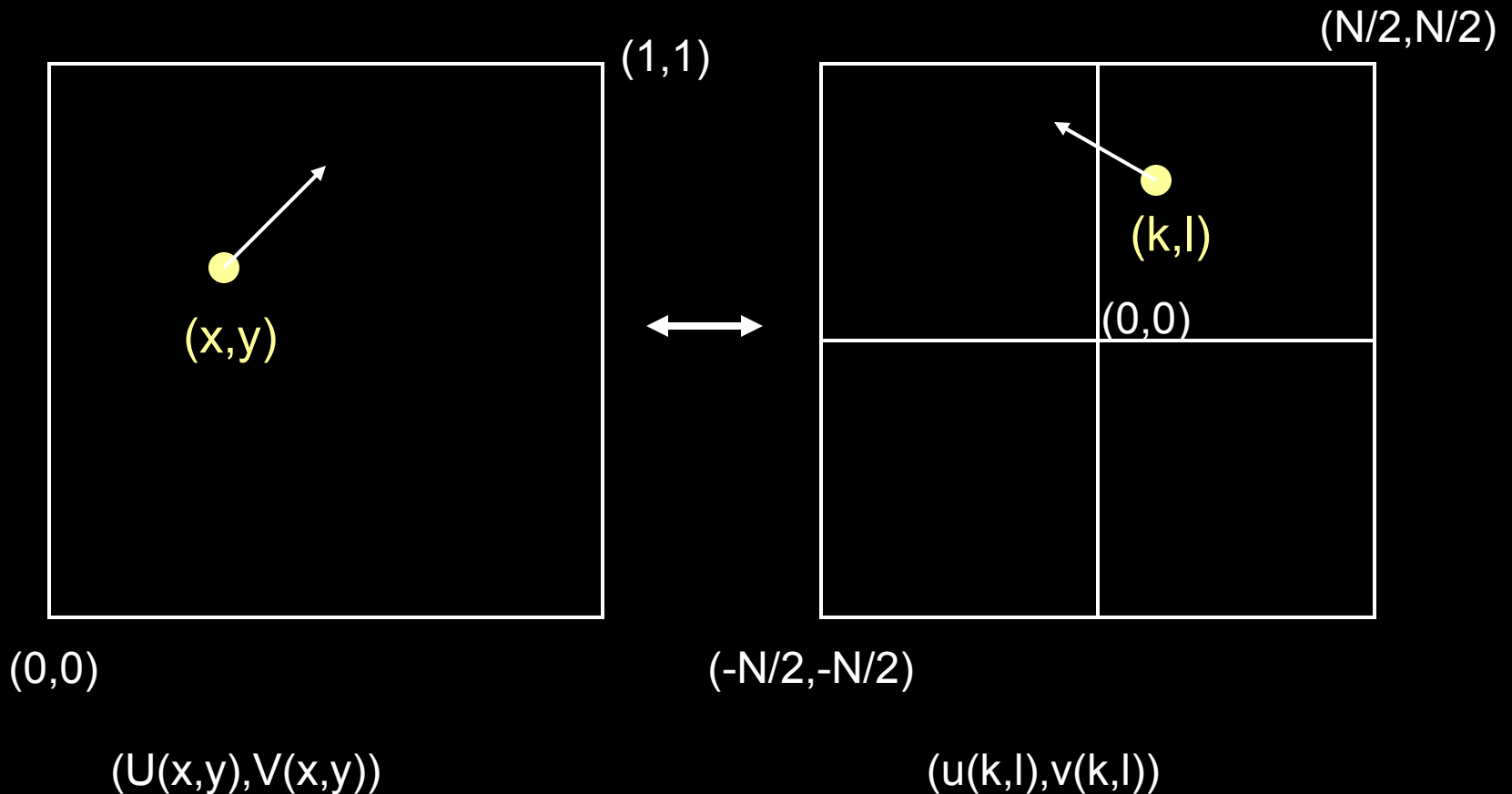
Simple Stable Fluid Solver

Periodic boundaries



Simple Stable Fluid Solver

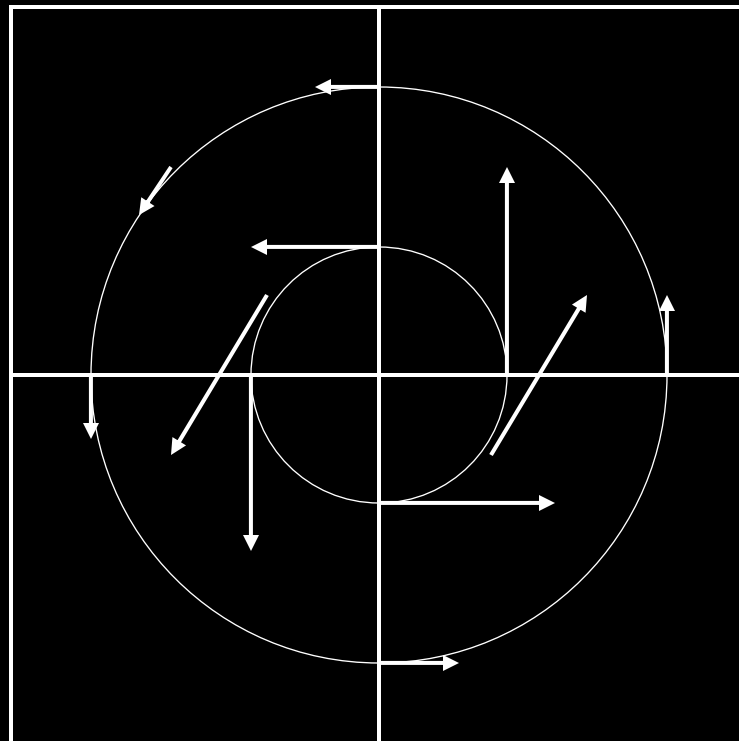
Fourier space



Simple Stable Fluid Solver

diffusion = low pass filter: $\exp(-(k^2 + \rho^2)v\Delta t)$

mass conservation = velocity perpendicular to the fourier directions



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

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

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

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

Show 2D simple demo

Defeating Dissipation

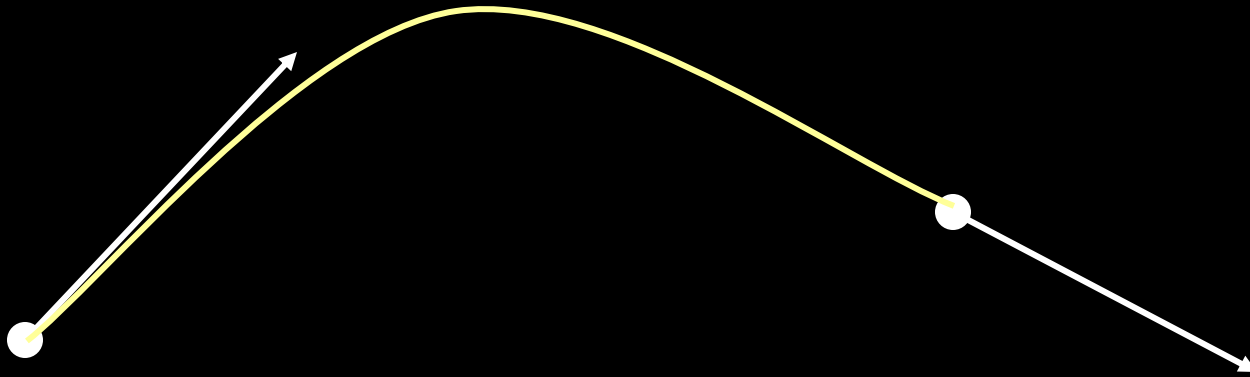
Work with

Ronald Fedkiw & Henrik Wann Jensen

Stanford University

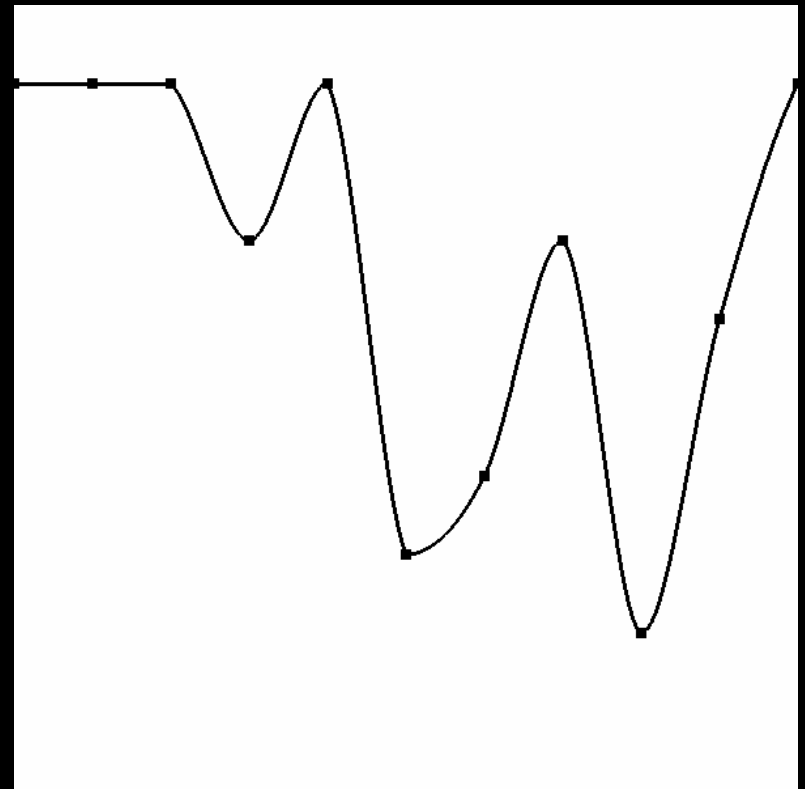
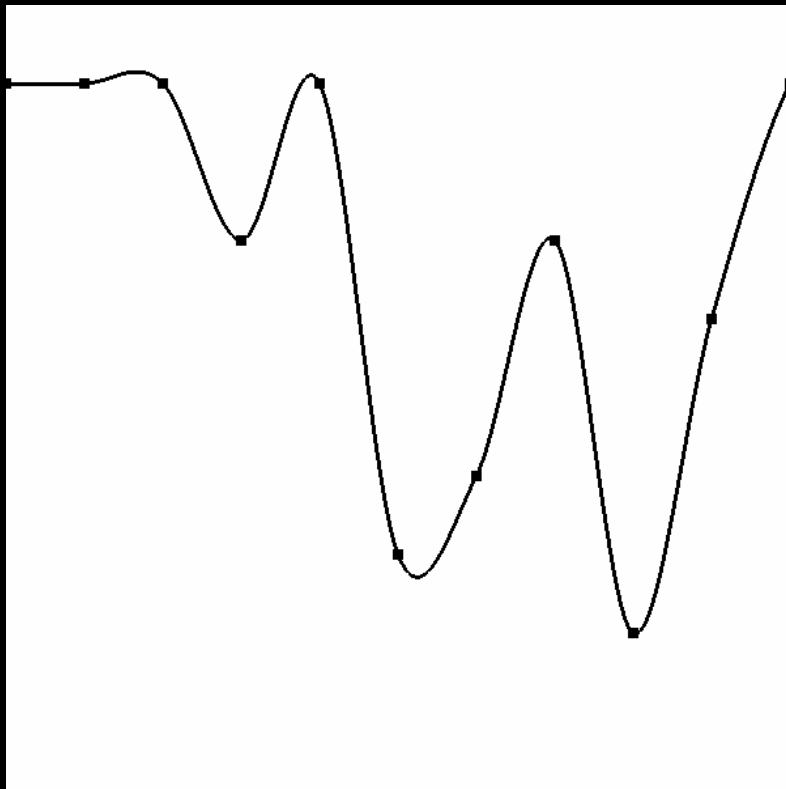
Higher Order Interpolation

Use Hermite interpolation instead of Linear interpolation.

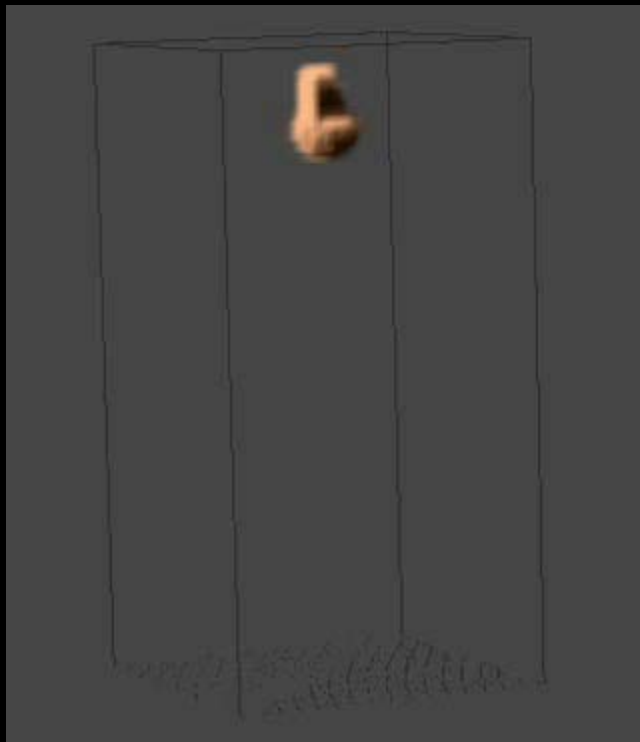


Higher Order Interpolation

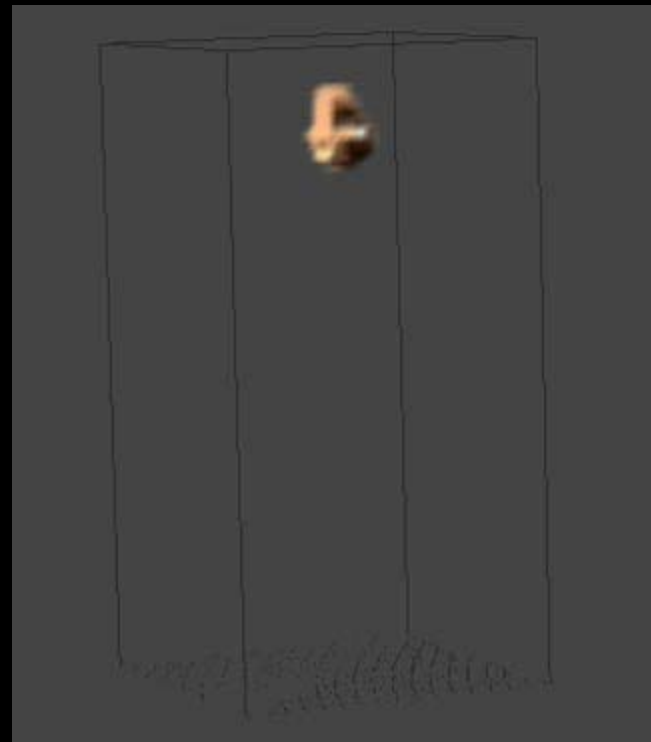
Force interpolant to be monotonic to avoid instabilities



Higher Order Interpolation

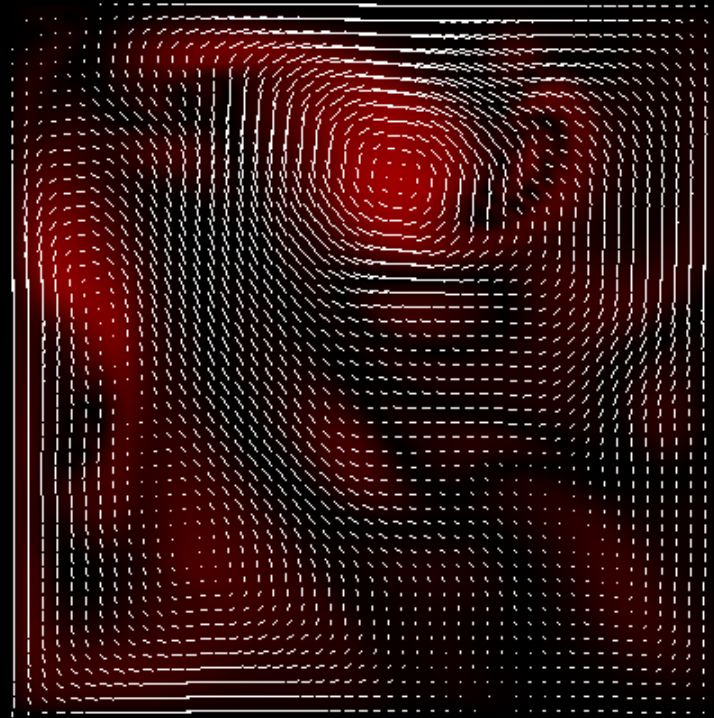


linear



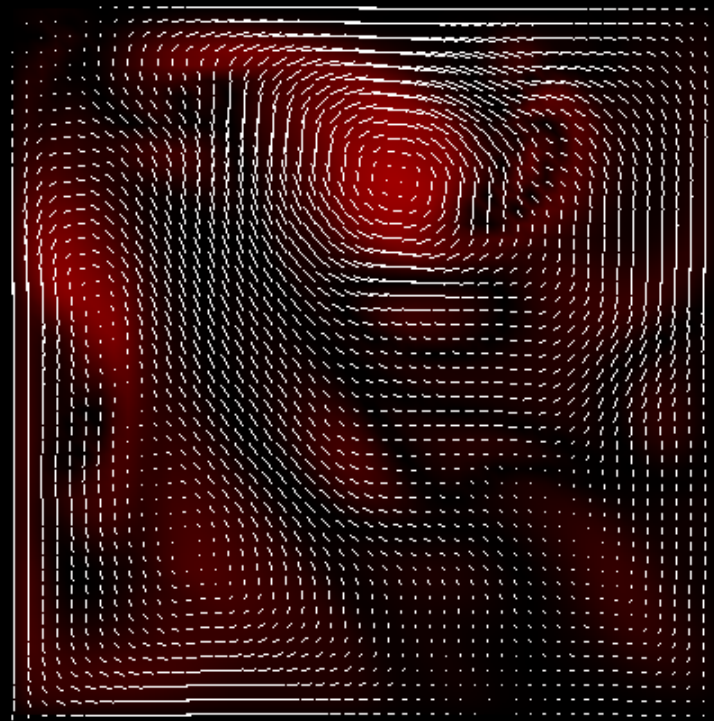
Hermite

Vorticity Confinement



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

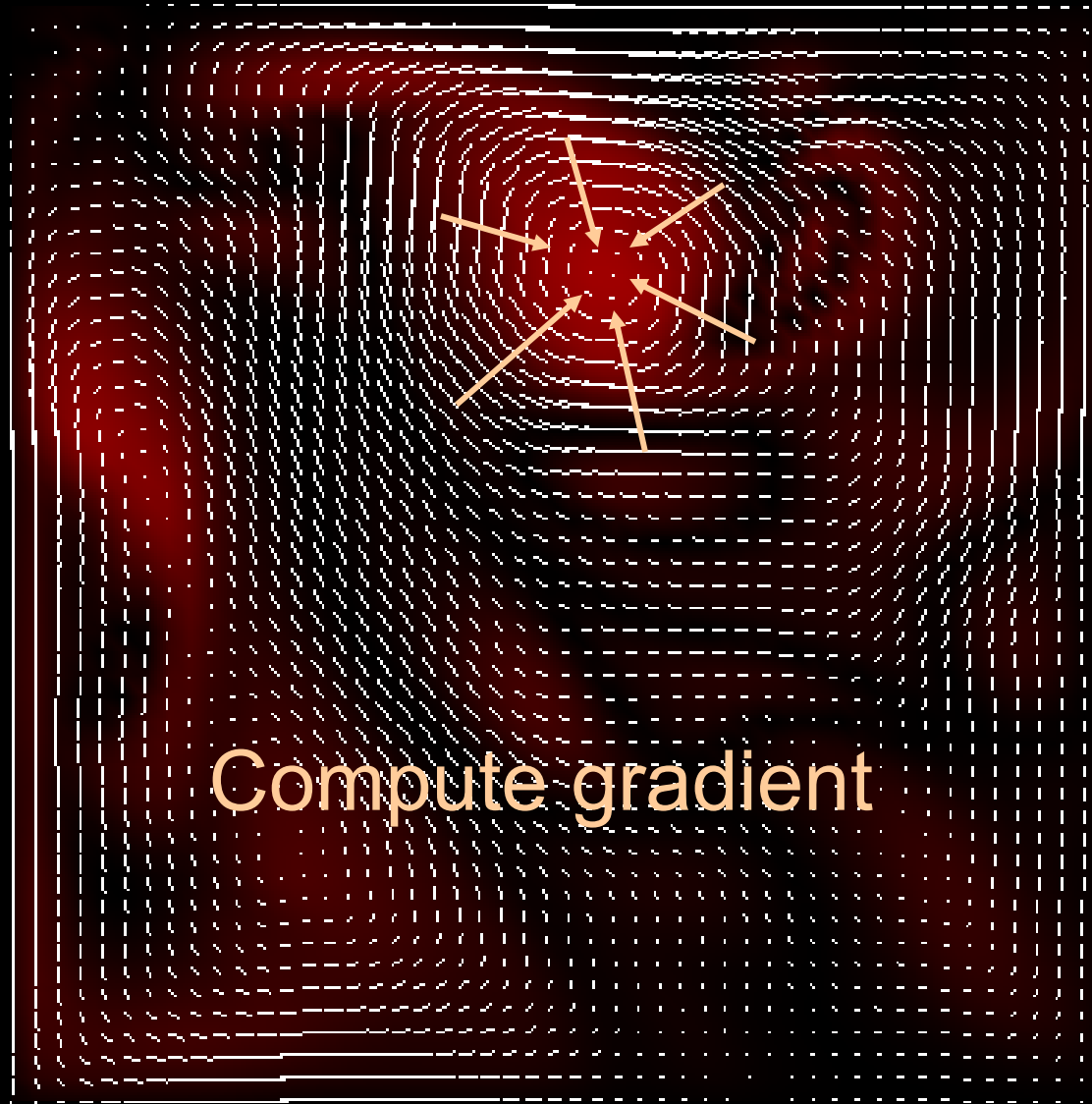
Vorticity Confinement



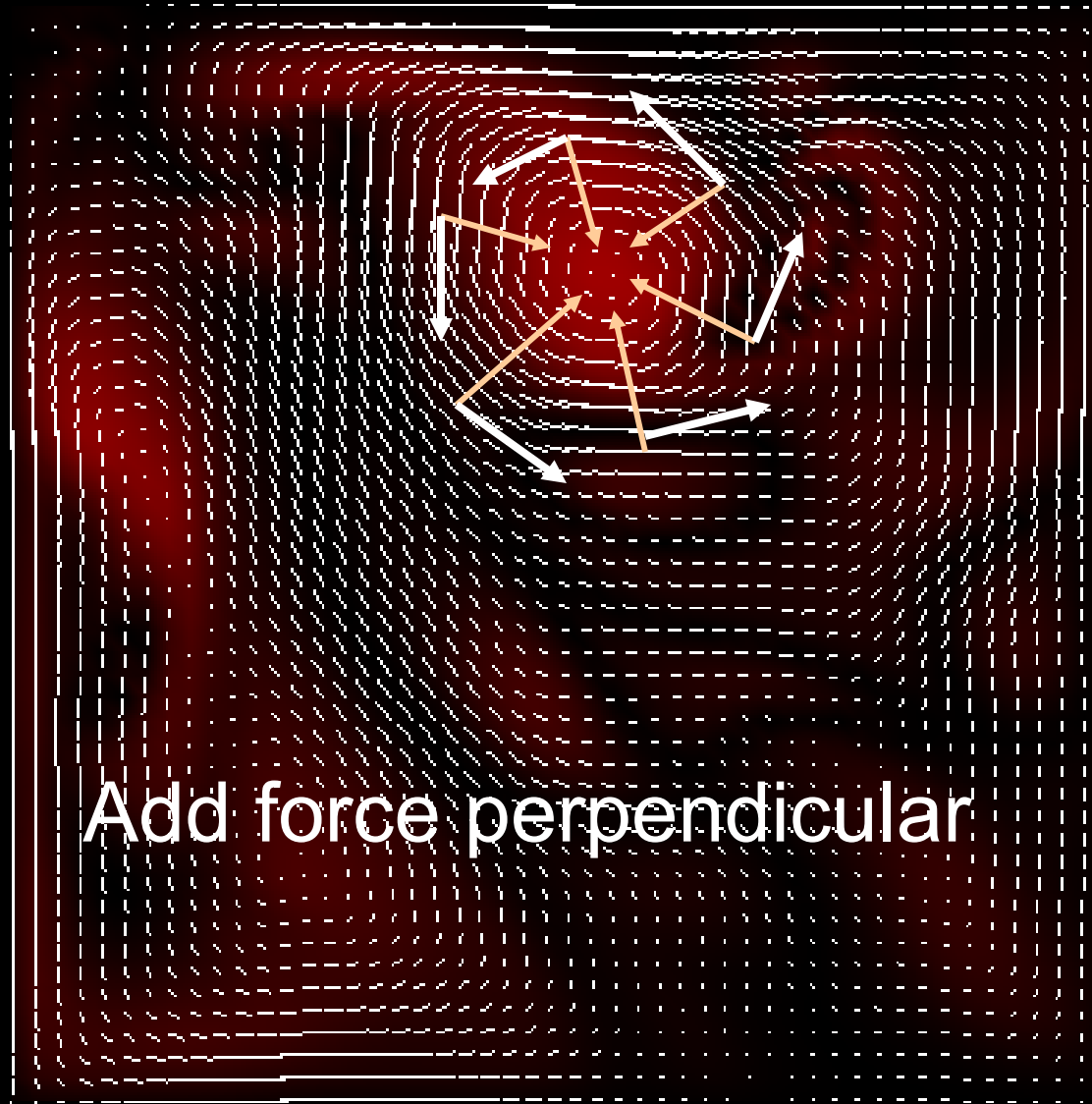
Basic idea : Increase vorticity

John Steinhoff 1987 (Flow Analysis)

Vorticity Confinement

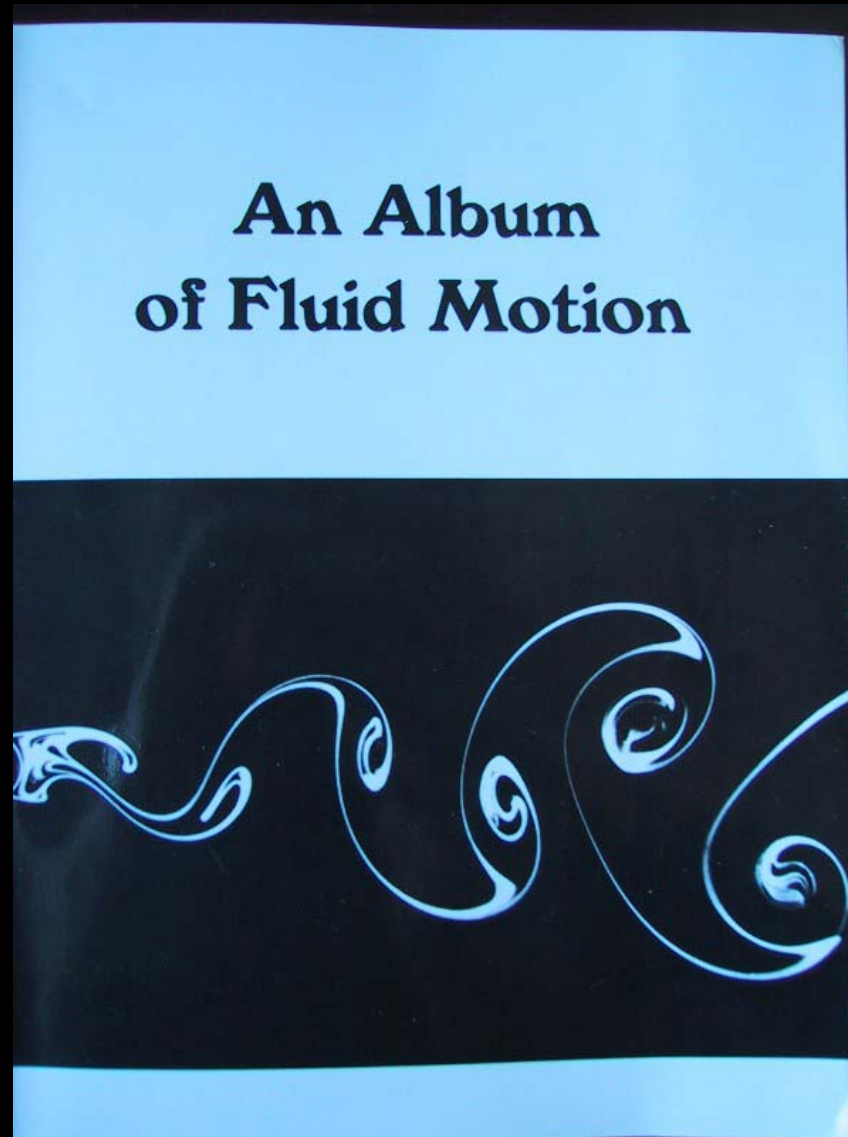


Vorticity Confinement

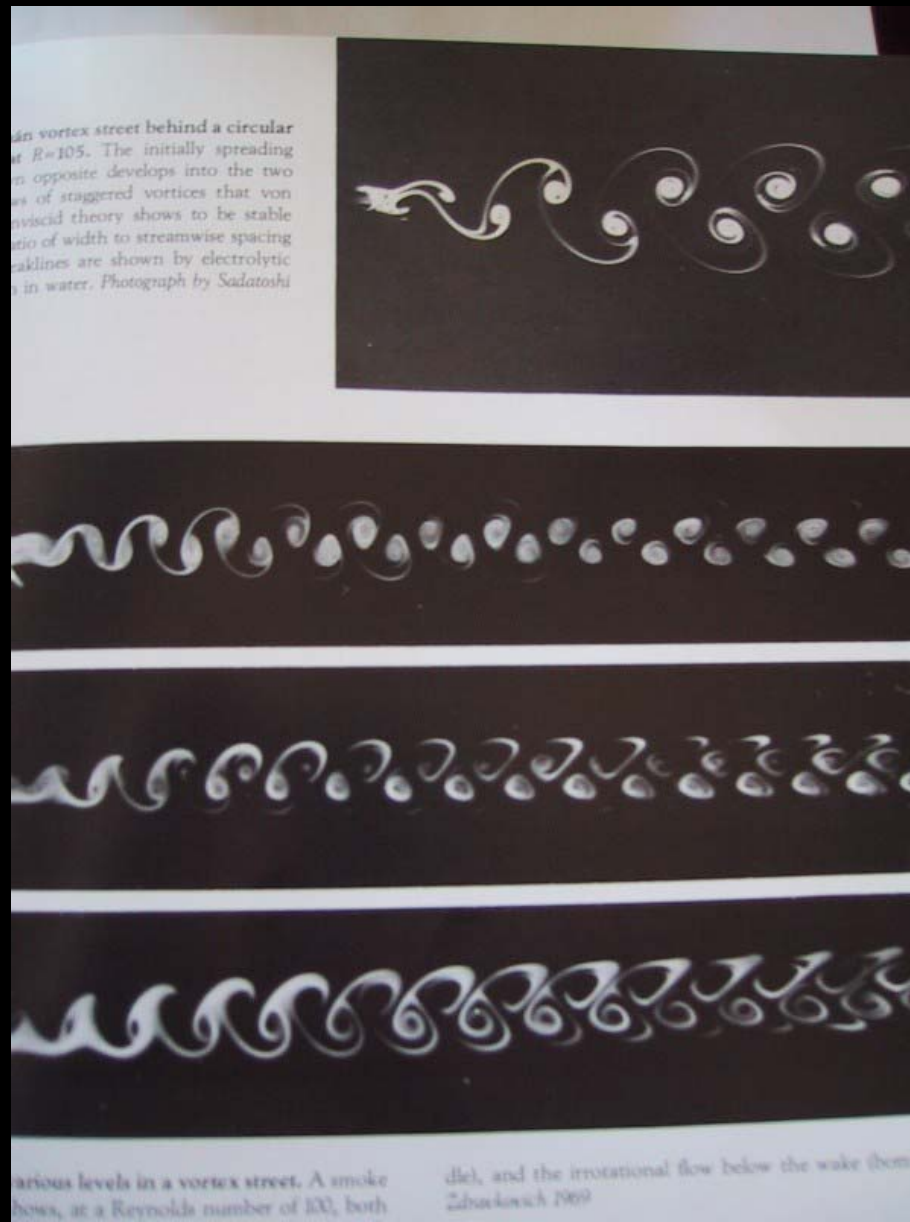


Show confinement demo

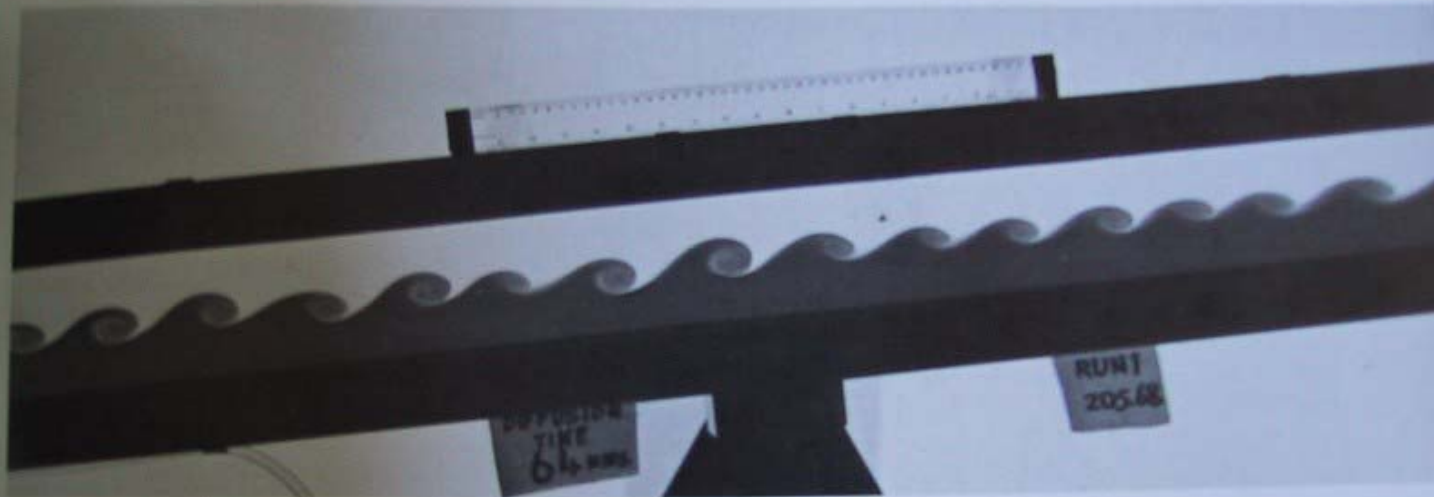
An Album of Fluid Motion



An Album of Fluid Motion



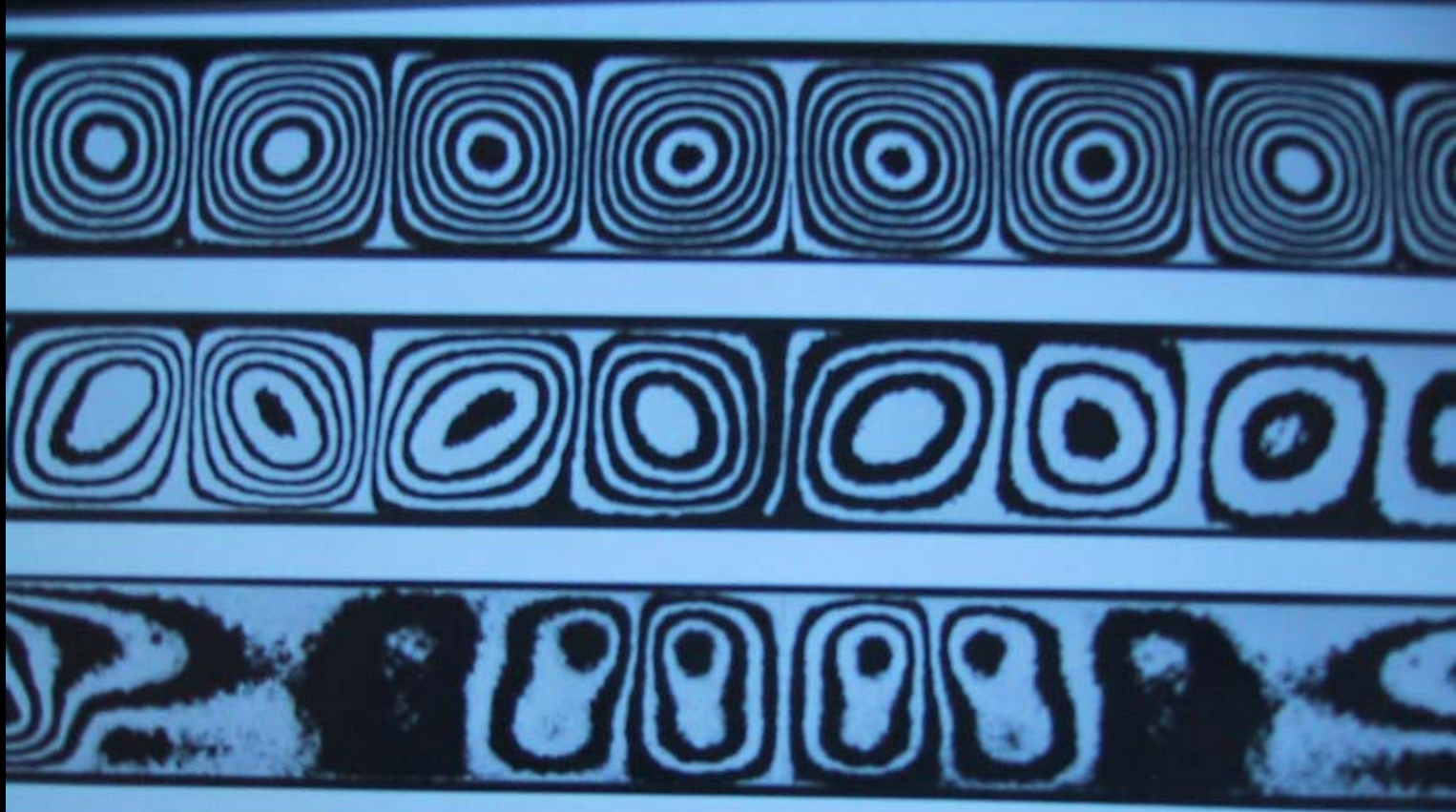
An Album of Fluid Motion



145. Kelvin-Helmholtz instability of stratified shear flow. A long rectangular tube, initially horizontal, is filled with water above colored brine. The fluids are allowed to diffuse for about an hour, and the tube then quickly tilted six degrees, setting the fluids into motion. The brine accel-

erates uniformly down the slope, while the water above similarly accelerates up the slope. Sinusoidal instability of the interface occurs after a few seconds, and has here grown nonlinearly into regular spiral rolls. *Thorpe 1971*

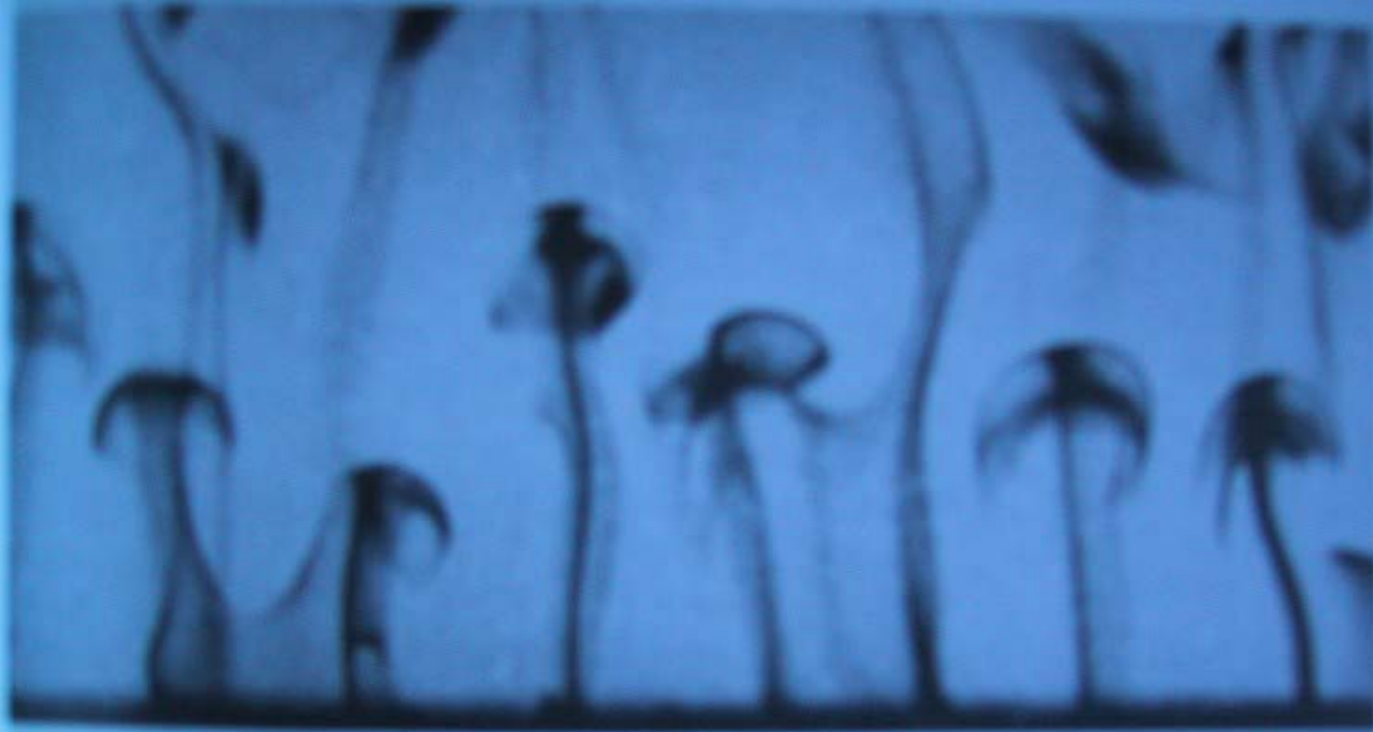
An Album of Fluid Motion



139. Buoyancy-driven convection rolls. Differential interferograms show side views of convective instability of silicone oil in a rectangular box of relative dimensions 10:4:1 heated from below. At the top is the classical Rayleigh-Bénard situation: uniform heating produces rolls

parallel to the shorter side. In the middle photographs temperature difference and hence the amplitude of the rolls increase from right to left. At the bottom, the rolls are rotating about a vertical axis. Oertel & Kirchhoff, *Oertel 1982a*

An Album of Fluid Motion



108. Buoyant thermals rising from a heated surface. Mushroom-shaped plumes rise periodically above a heated copper plate. They are made visible by an electrochemical

An Album of Fluid Motion

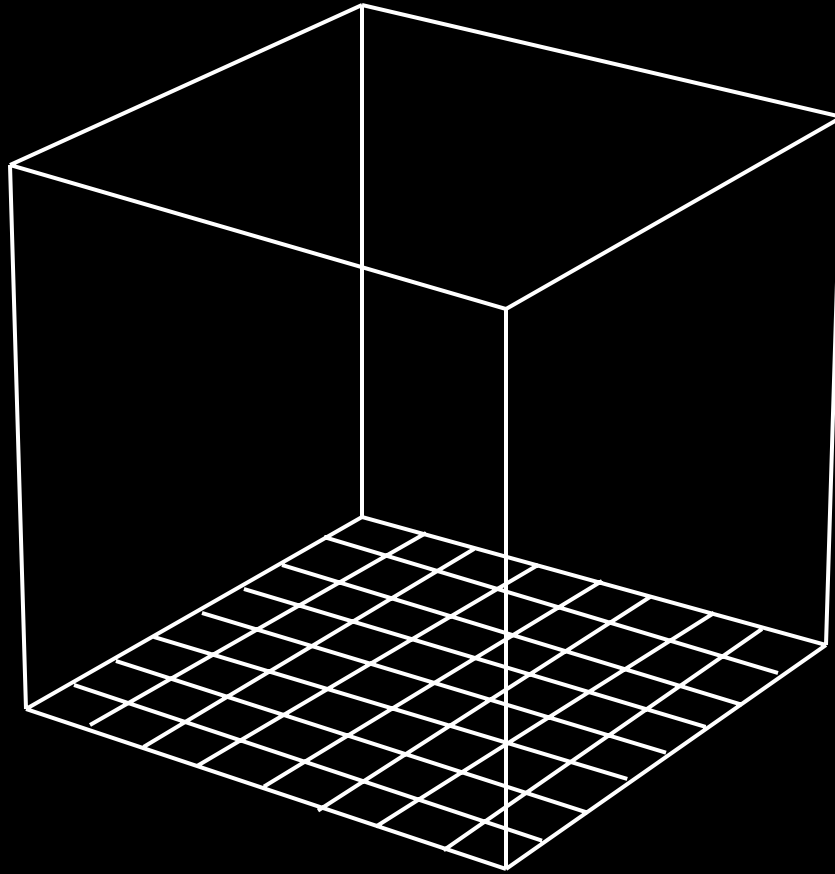
Show demos

3D Demos

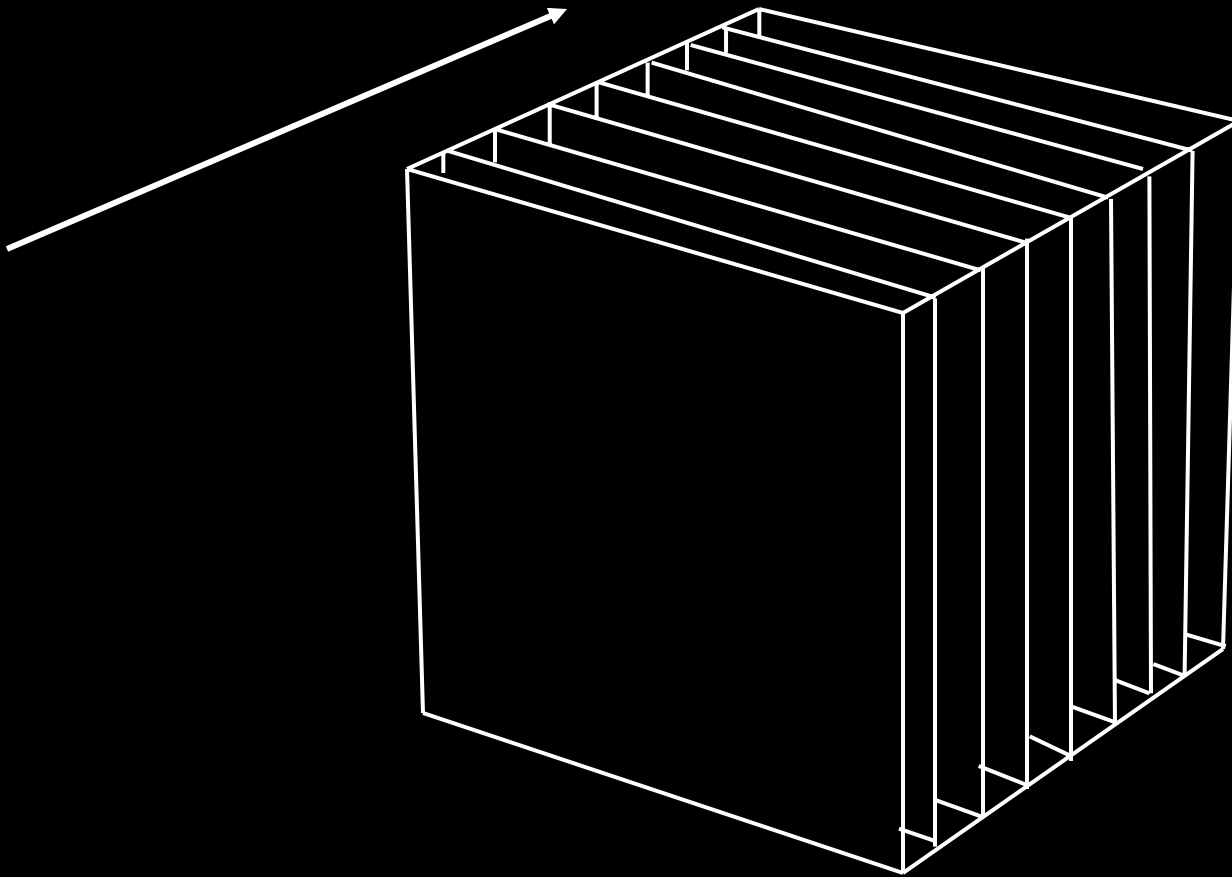
Move 3D solid texture coordinates

Interactive Volume Rendering

Volume Rendering



Volume Rendering



Render slices from front to back

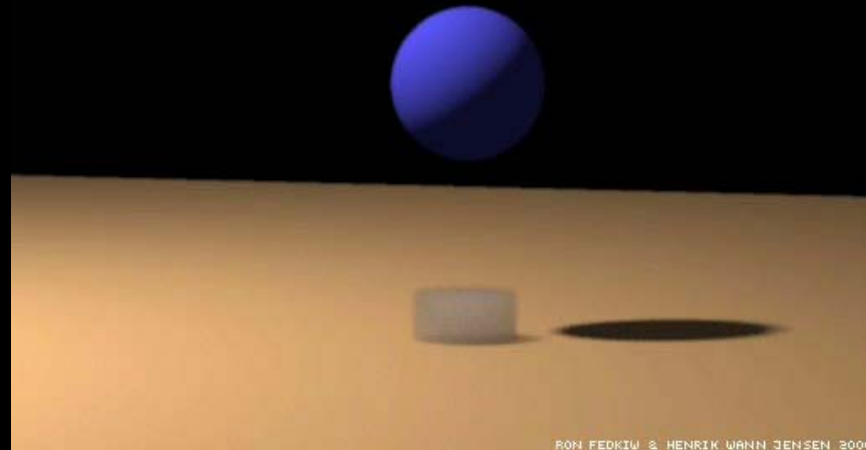
Show 3D demo

High Quality Renderings



RON FEDKIW & HENRIK WANN JENSEN 2000

High Quality Renderings



RON FEDKIW & HENRIK WANN-JENSEN, 2000

PocketPC demo

Show demo

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))
```

Work in Progress...

- Clouds
- Flows on Surfaces

Future Work

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

MAYA 4.5

Maya Fluid Effects

See our booth for demos