Real-Time Fluid Dynamics for Games

Jos Stam
Research
Alias|wavefront

Patent

Parts of this work protected under:

US Patent 6,266,071

Motivation

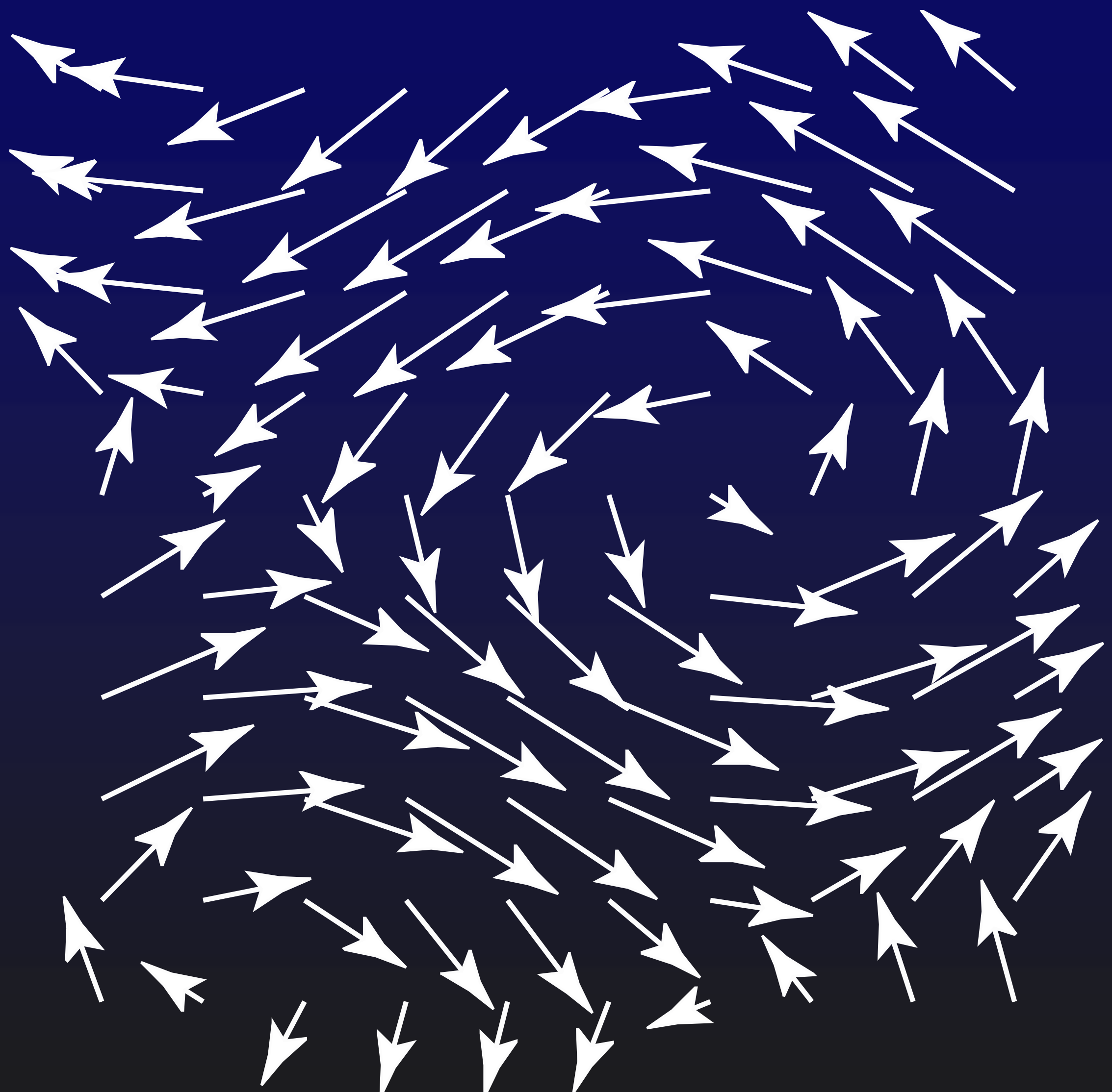
- Has to Look Good
- Be Fast – Real-Time (Stable)
- And Simple to Code

Important in Games !

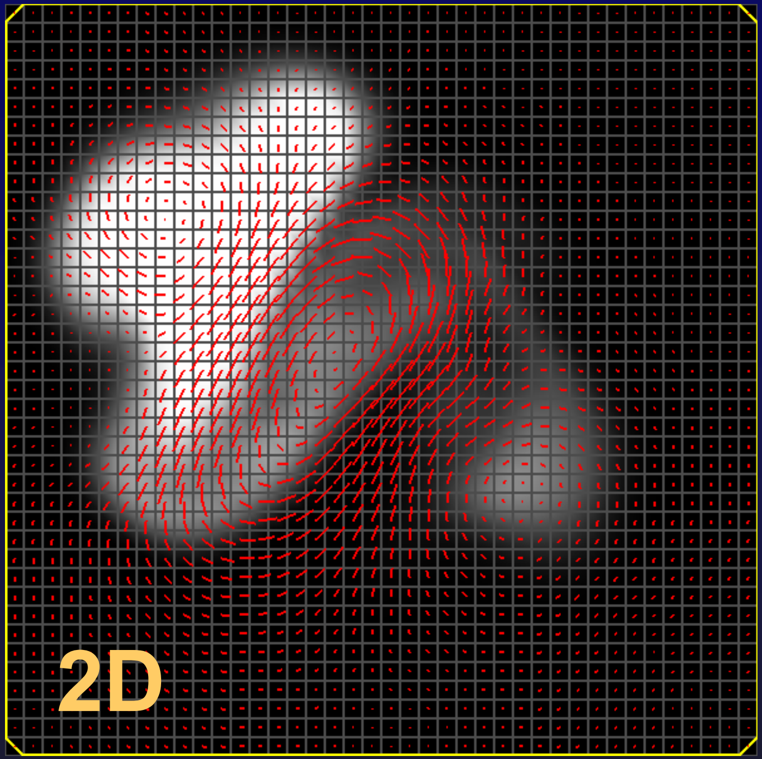
Fluid Mechanics

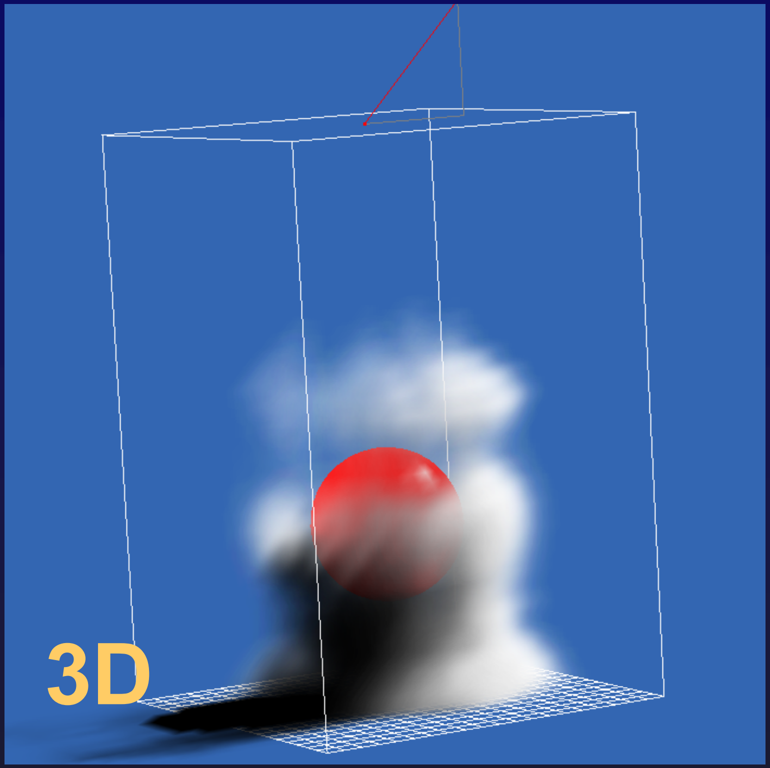
- Natural Framework
- Lots of Previous Work !
- Very Hard Problem
- Visual Accuracy ?

Fluid Mechanics



Main Application





Moving Densities

Main Application

While (Simulating)

- Get Forces from UI
- Get Source Densities from UI
- Update Velocity Field
- Update Density Field
- Display Density

Navier-Stokes Equations

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

Velocity

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

Density

Equations Very Similar

What Does it Mean ?

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

Over Time…

What Does it Mean ?

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

Density Follows Velocity…

What Does it Mean ?

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

Density Diffuses at a Rate … κ

What Does it Mean ?

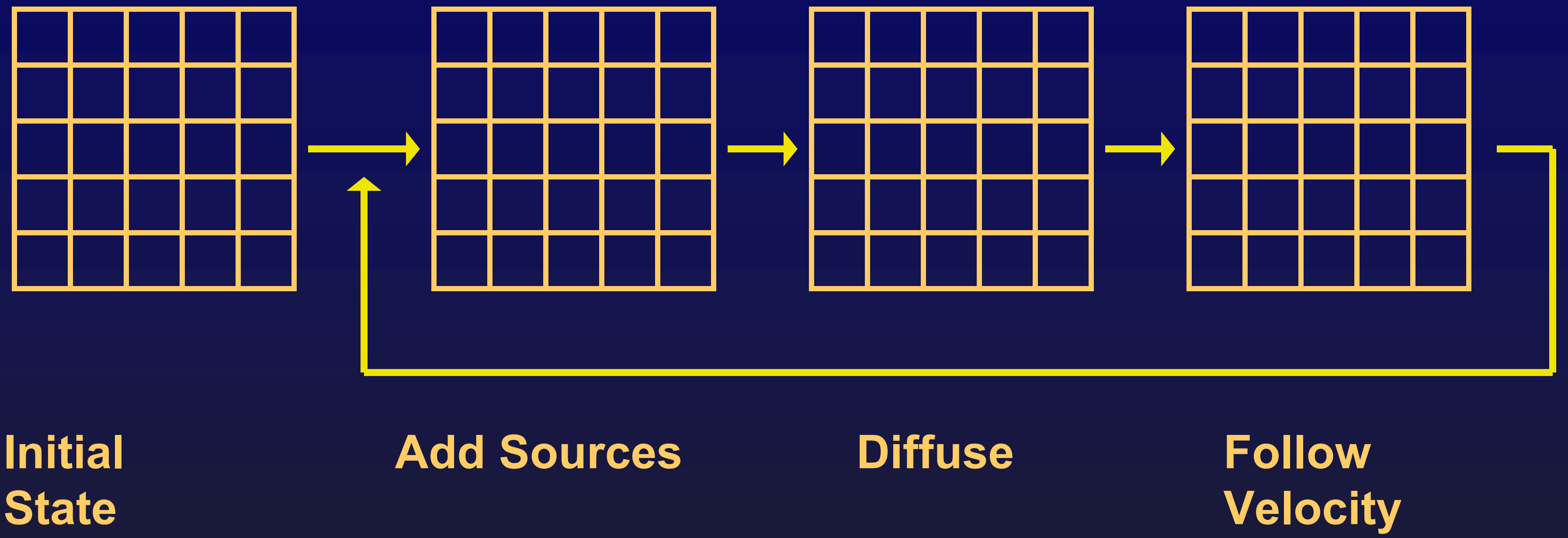
$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + \boxed{S}$$

Density Increases Due to Sources…

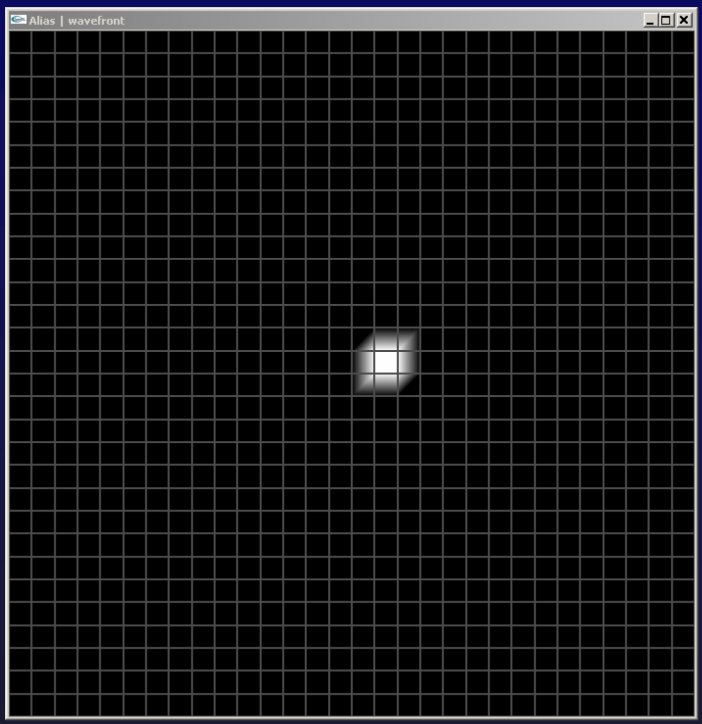
Fluid in a Box

Density Constant in Each Cell

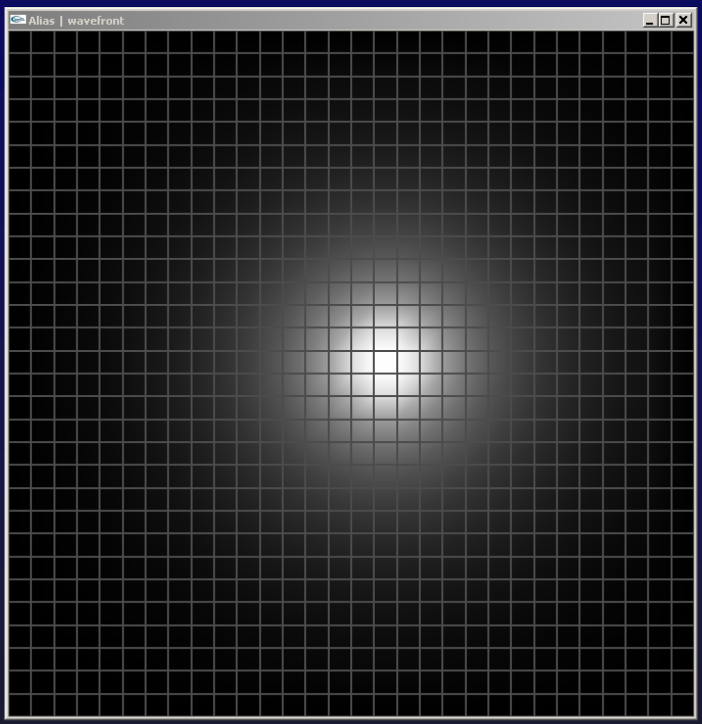
Simulation



Diffusion Step



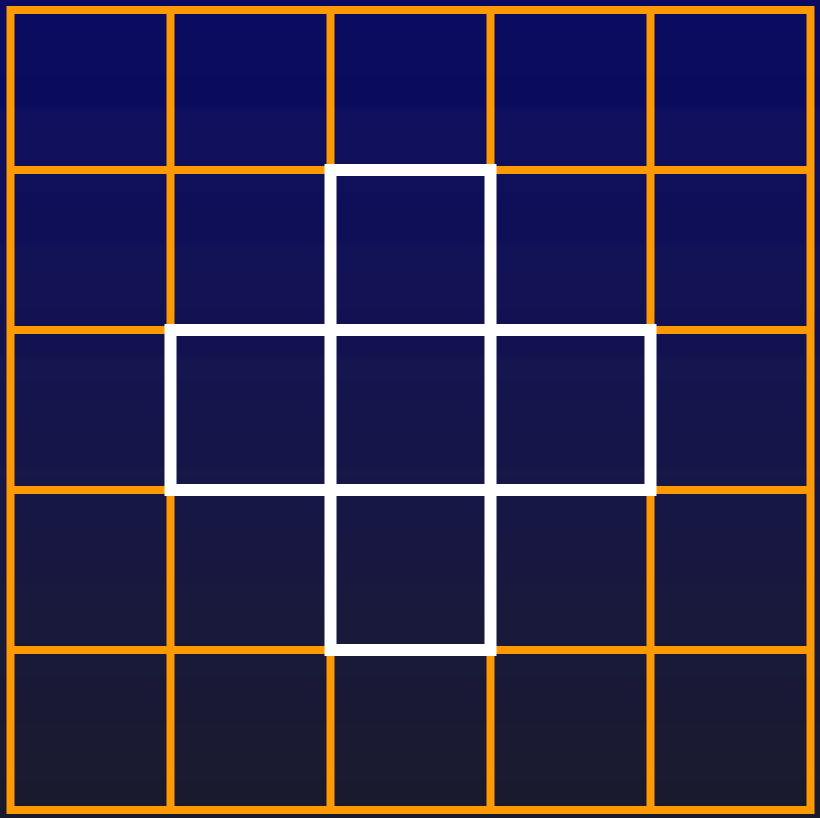
dt



`dens0[i,j]`

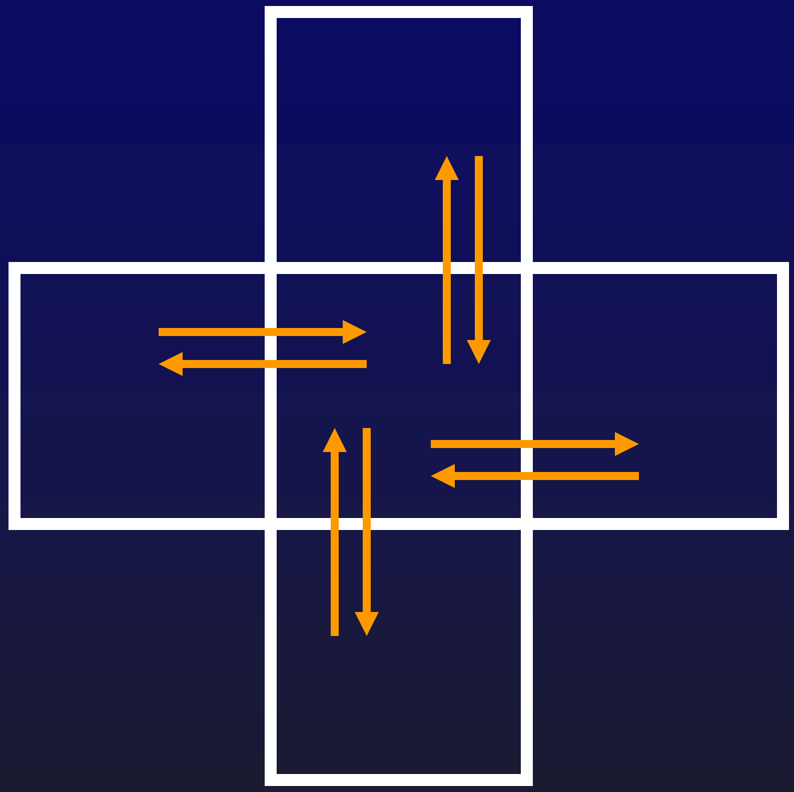
`dens[i,j]`

Diffusion Step



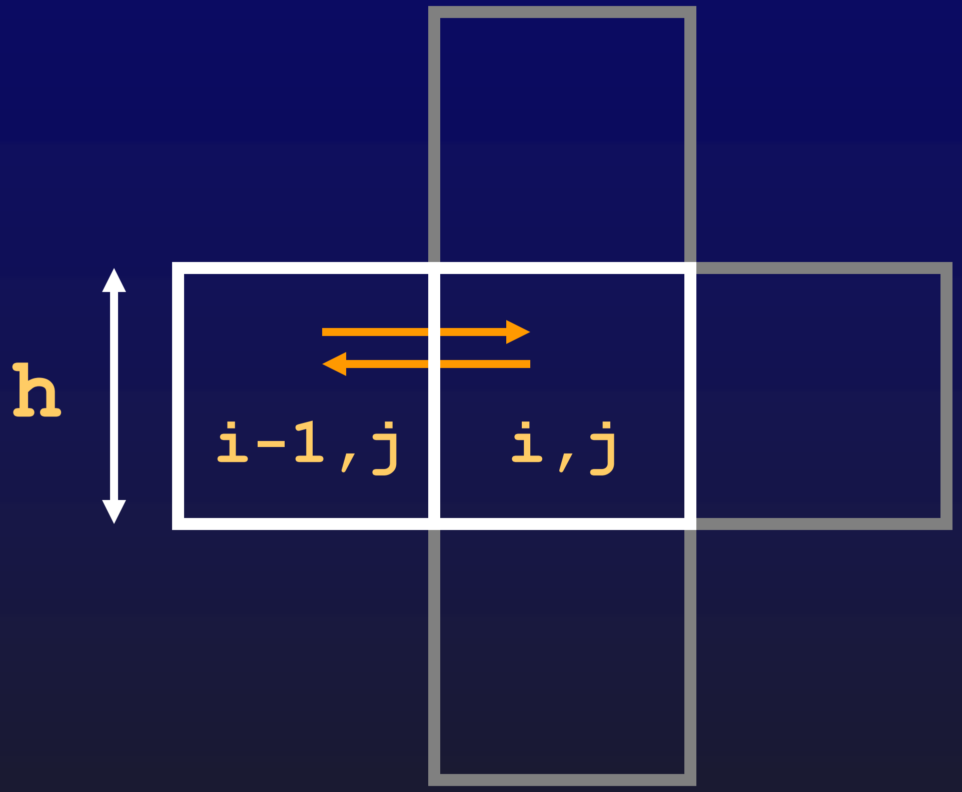
Exchanges Between Neighbors

Diffusion Step



Exchanges Between Neighbors

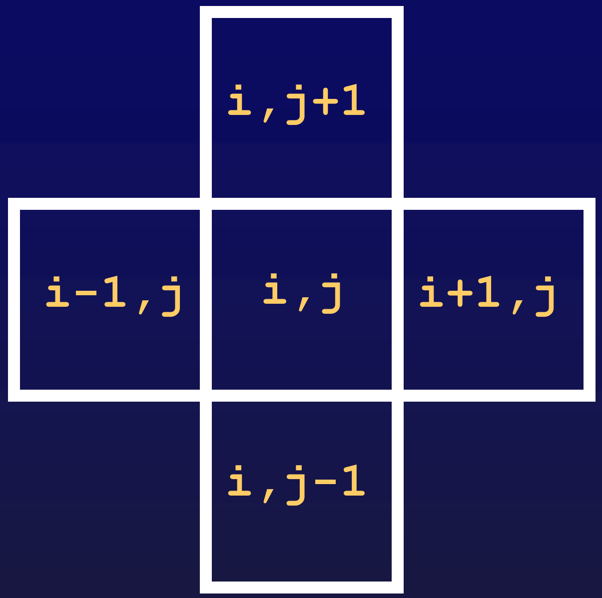
Diffusion Step



Change: Density Flux IN – Density Flux OUT

```
diff*dt*(dens0[i-1,j]-dens0[i,j])/(h*h)
```

Diffusion Step




```
dens[i,j] = dens0[i,j] + a*(dens0[i-1,j]+dens0[i+1,j]+
              dens0[i,j-1]+dens0[i,j+1]-4*dens0[i,j]);
```

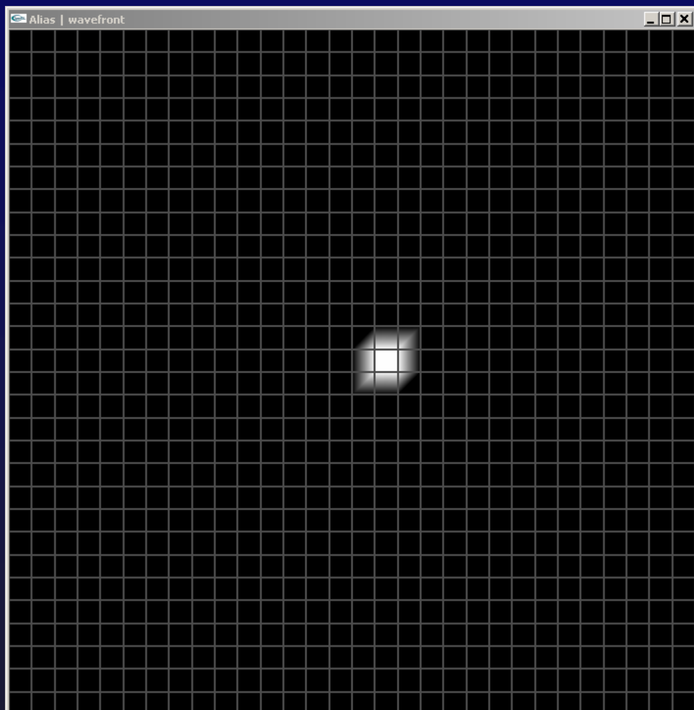
```
a = diff*dt/(h*h)
```

Diffusion Step

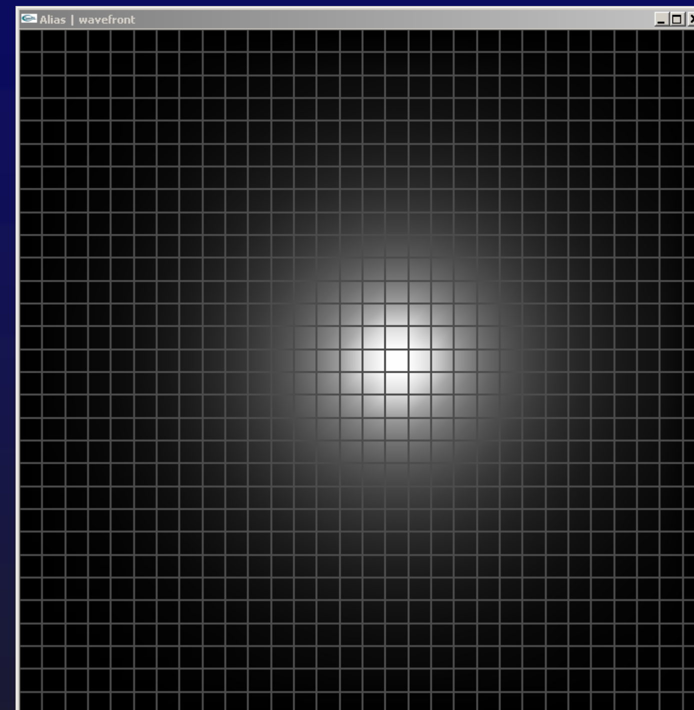
```
void diffuse_bad ( float * dens, float * dens0 )
{
  int i, j;
  float a = diff*dt/(h*h);

  for ( i=1 ; i<=N ; i++ ) {
    for ( j=1 ; j<=N ; j++ ) {
      dens[i,j] = dens0[i,j] + a*(dens0[i-1,j]+dens0[i+1,j]+
                        dens0[i,j-1]+dens0[i,j+1]-4*dens0[i,j]);
    }
  }
}
```

Simple But Doesn't Work: Unstable

Diffusion Step



-dt



`dens0[i,j]` `dens[i,j]`

Diffuse Backwards: Stable

Diffusion Step

```
dens0[i,j] = dens[i,j] -
      a*(dens[i-1,j]+dens[i+1,j]+
         dens[i,j-1]+dens[i,j+1]-4*dens[i,j]);
```

Linear System: Ax=b

Use a Fast Sparse Solver

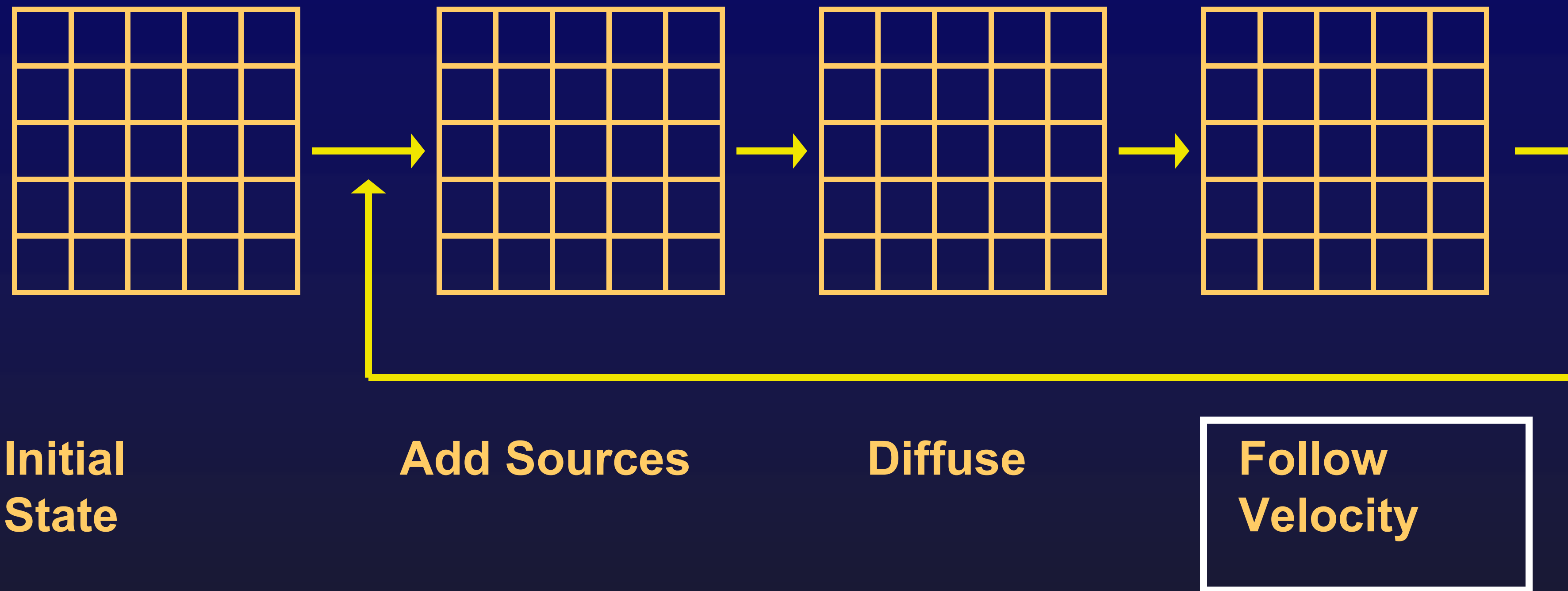
Linear Solvers

Gaussian Elimination	N^3
Gauss-Seidel Relaxation	N^2
Conjugate Gradient	$N^{1.5}$
Cyclical Reduction	N logN
Multi-Grid	N

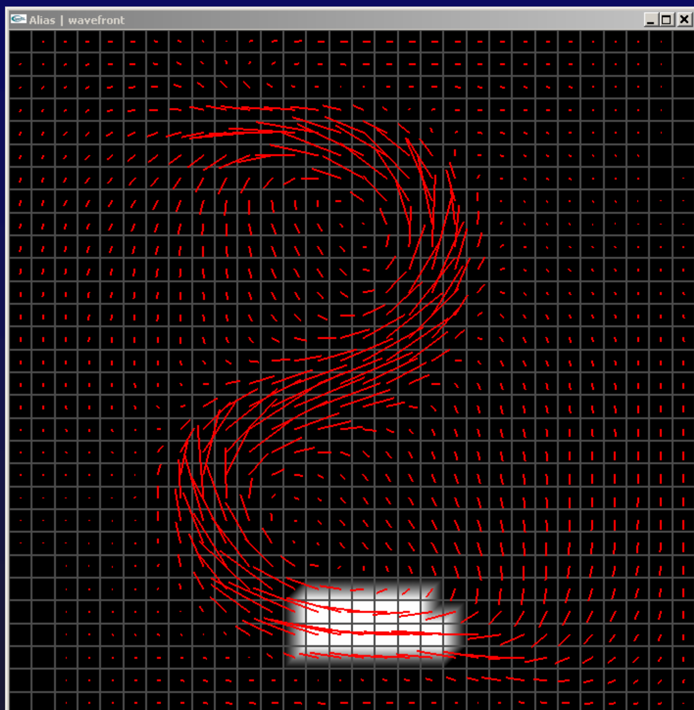
Diffusion Step

```
void lin_solve ( float * x, float * b, float a, float c )
{
  int i, j, n;

  for ( n=0 ; n<20 ; n++ ) {
    for ( i=1 ; i<=N ; i++ ) {
      for ( j=1 ; j<=N ; j++ ) {
        x[i,j] = (b[i,j] + a*(x[i-1,j]+x[i+1,j]+
                              x[i,j-1]+x[i,j+1]))/c;
      }
    }
  }
}

void diffuse ( float * dens, float * dens0 )
{
  float a = diff*dt/(h*h);
  lin_solve ( dens, dens0, a, 1+4*a );
}
```

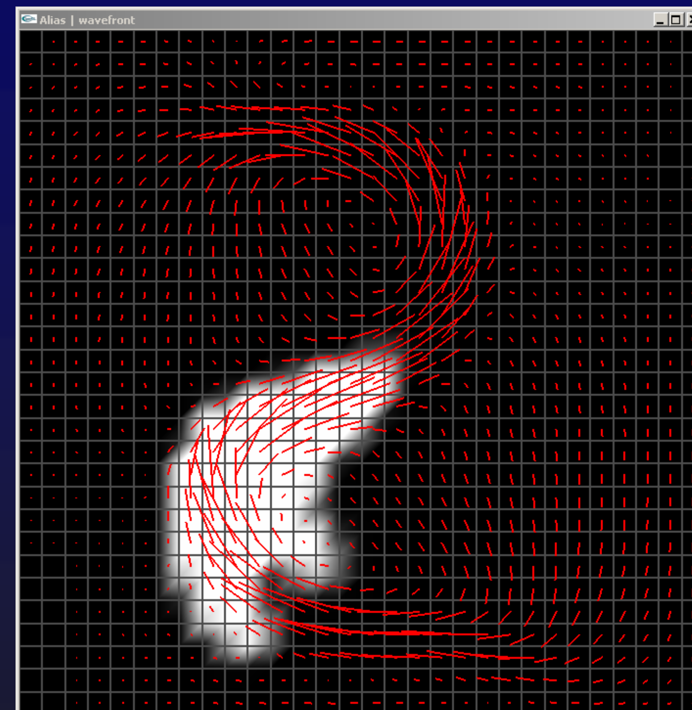
Simulation



Follow Velocity



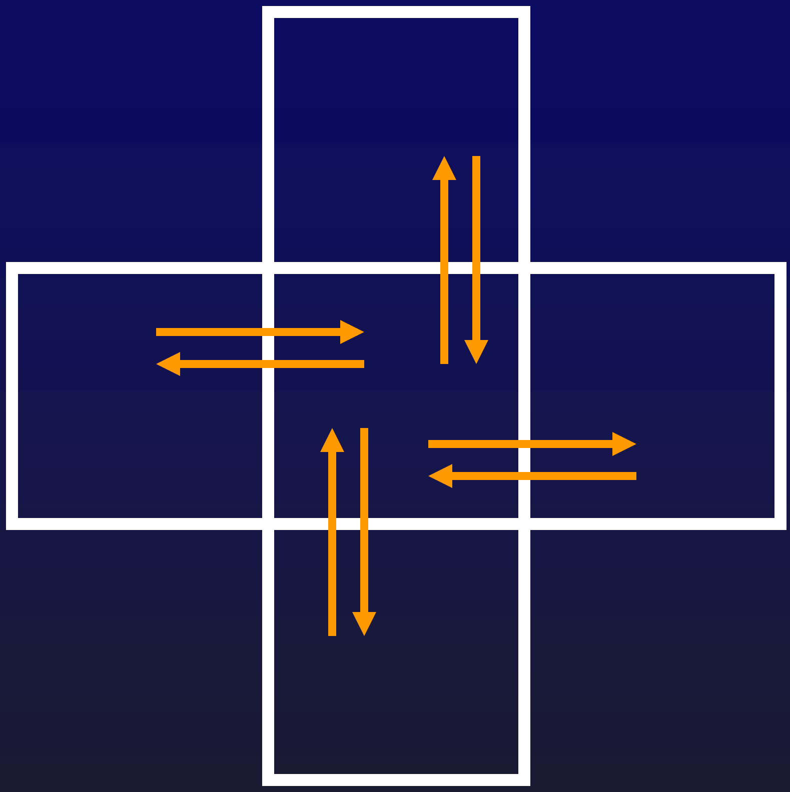
dt




```
dens0[i,j]
u[i,j], v[i,j]
```

```
dens[i,j]
```

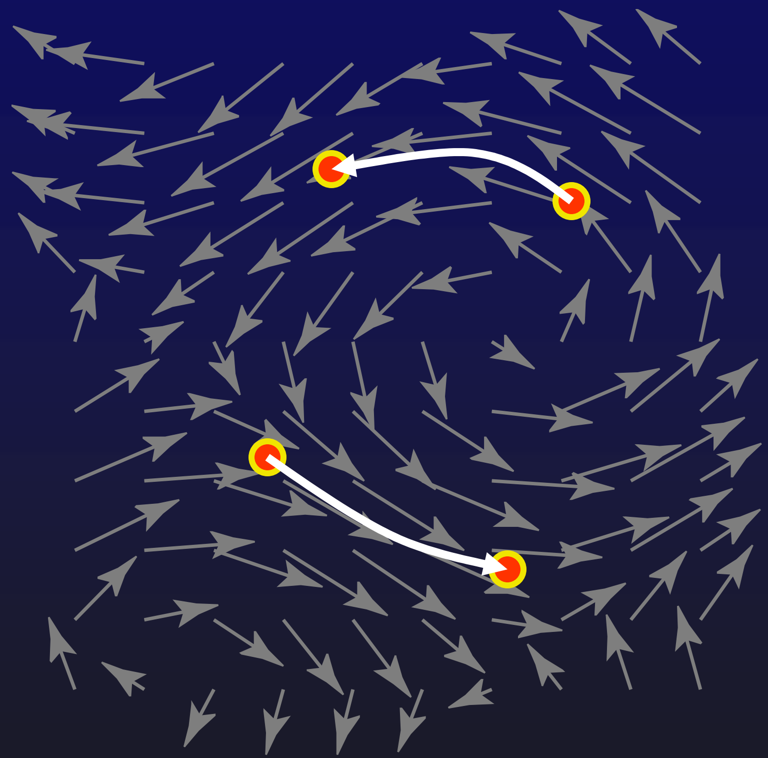
Follow Velocity



Fluxes Depend on Velocity

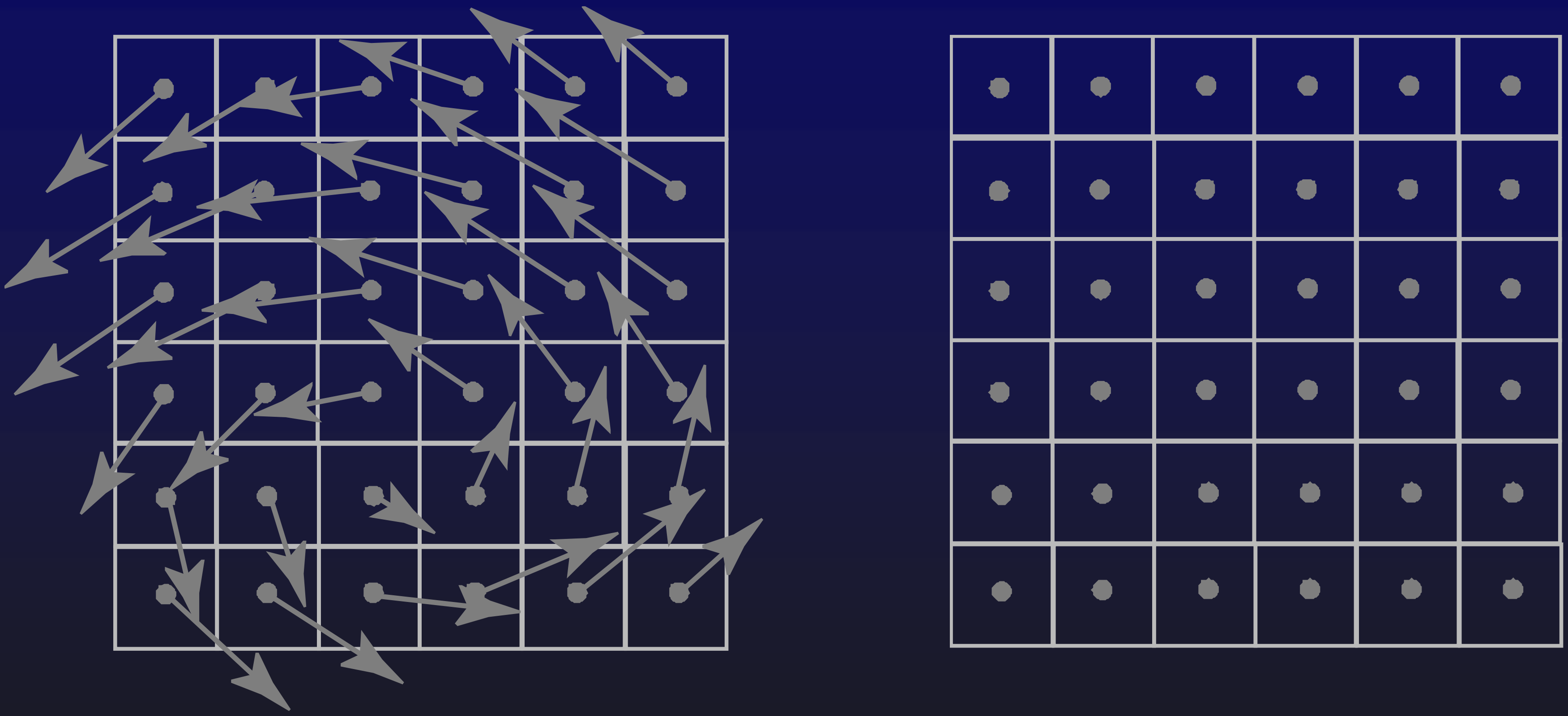
Follow Velocity

Better Idea:



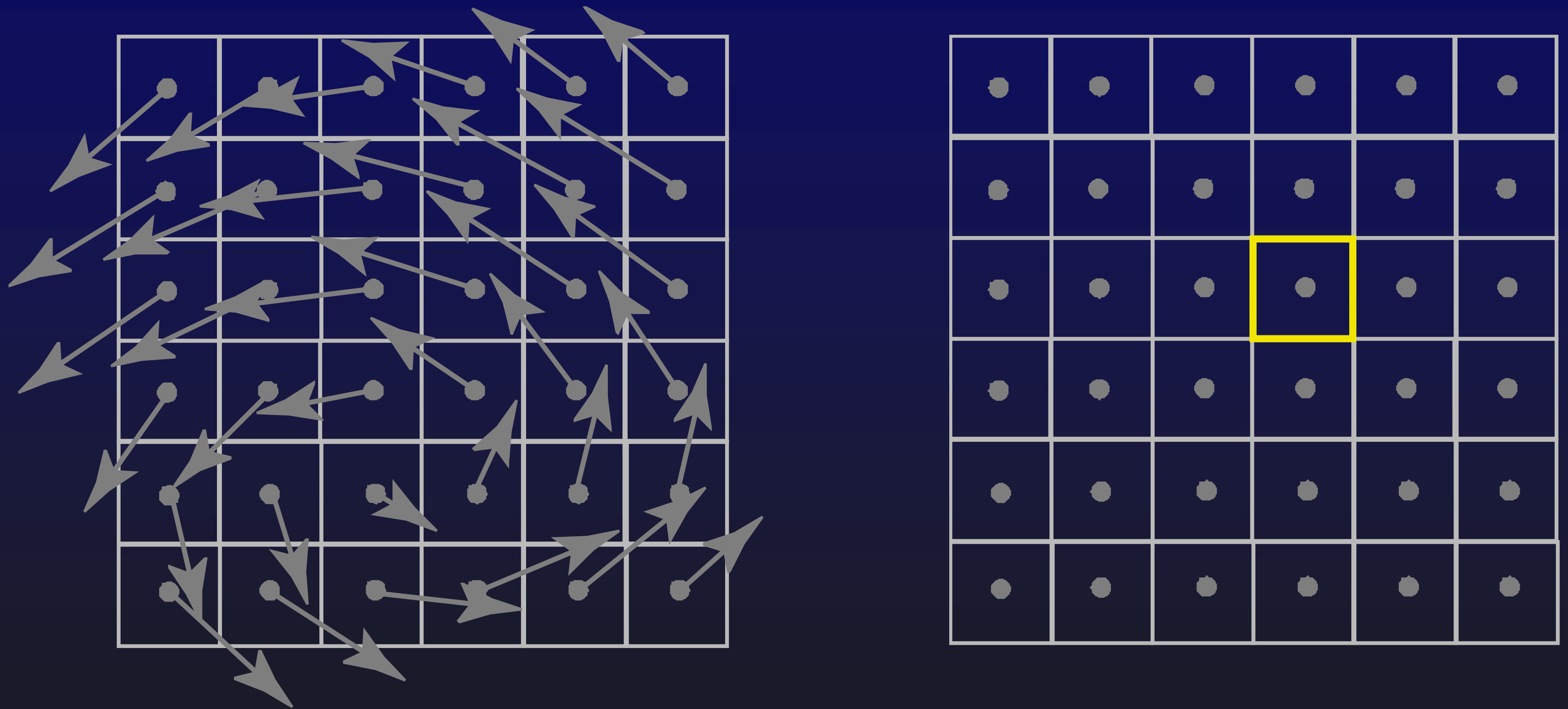
Step Easy If Density Were Particles

Follow Velocity



Follow Velocity

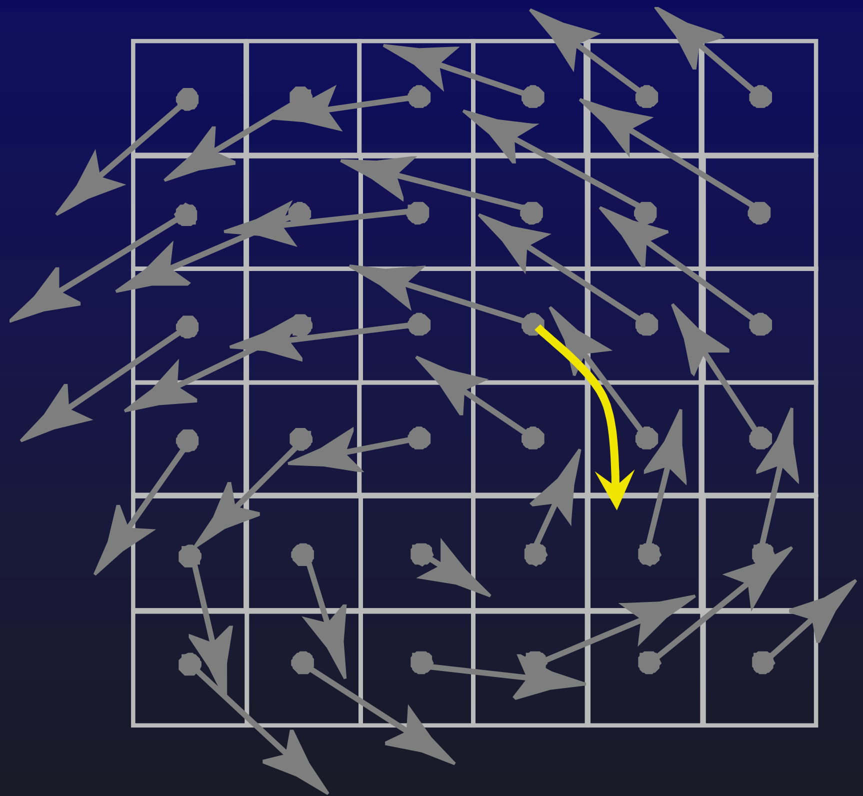
For Each Cell…

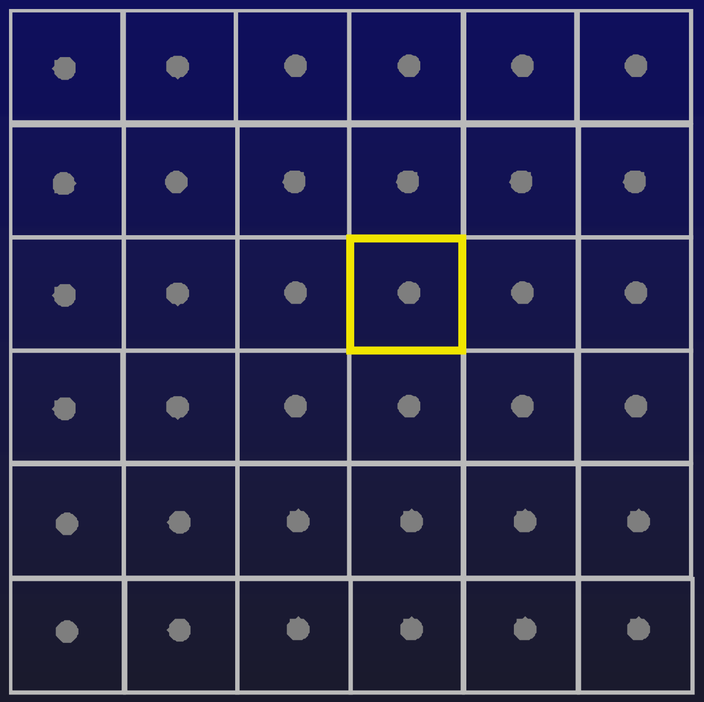


`dens[i,j]`

Follow Velocity

Trace BackWard

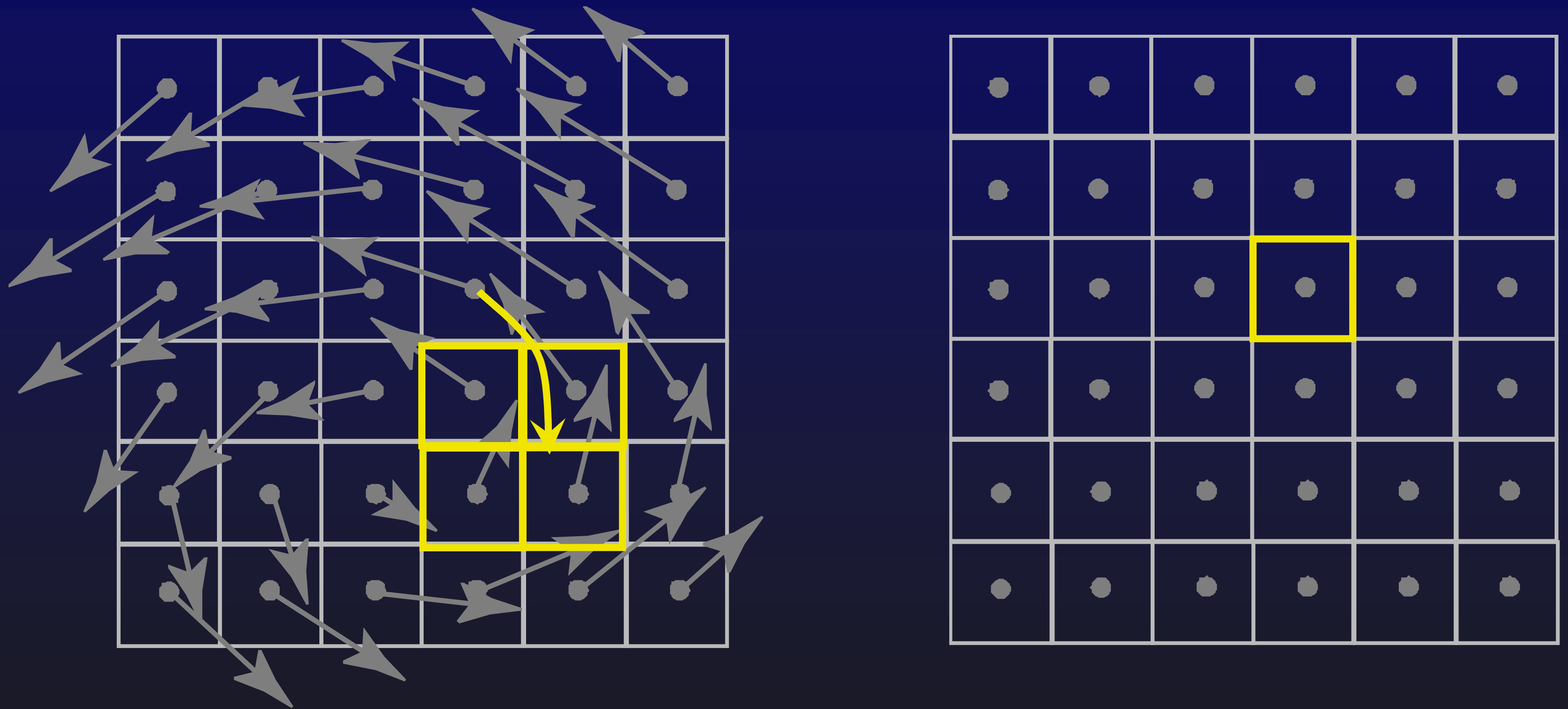




```
x = i-dt*u[i,j];
y = j-dt*v[i,j];
```

Follow Velocity

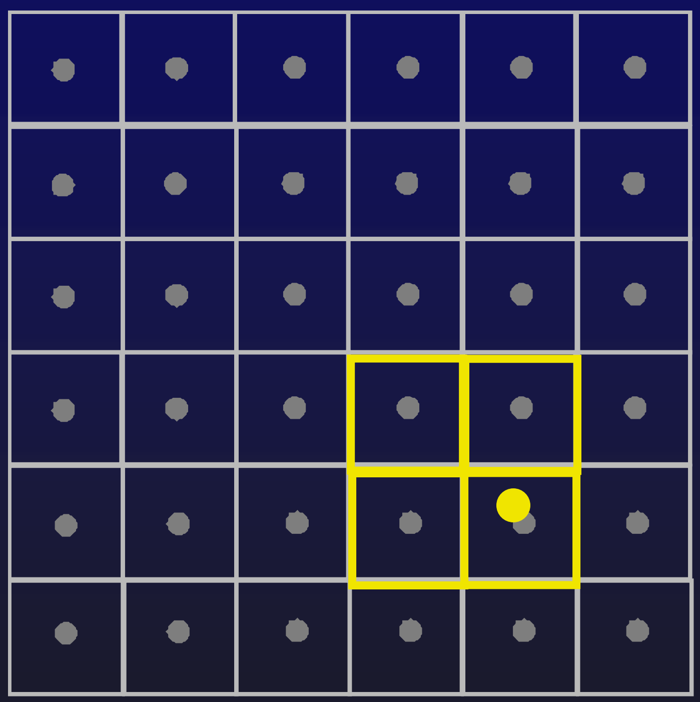
Find Four Neighbors

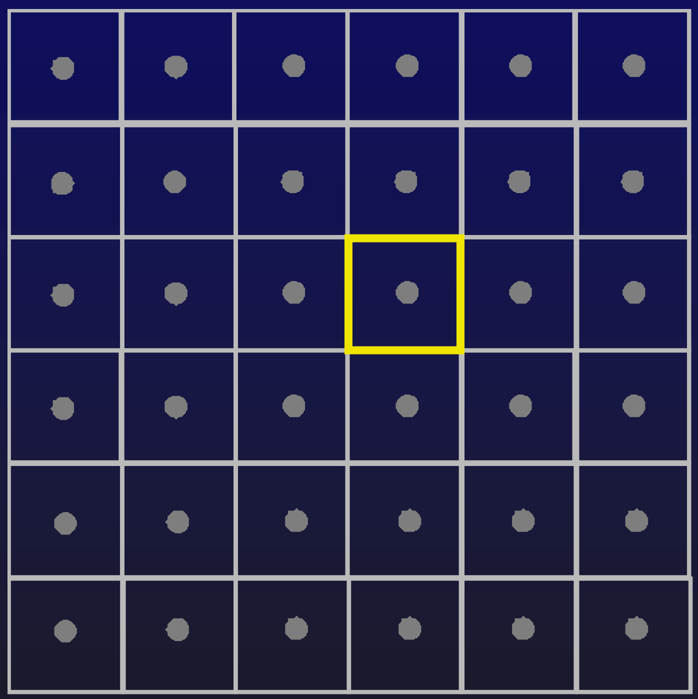


```
i0 = (int)x; i1=i0+1; s=x-i0;
j0 = (int)y; j1=j0+1; t=y-j0;
```

Follow Velocity

Interpolate From Neighbors





```
d = (1-s)*((1-t)*dens0[i0,j0]+t*dens0[i0,j1])+
     s *((1-t)*dens0[i1,j0]+t*dens0[i1,j1]);
```

Follow Velocity

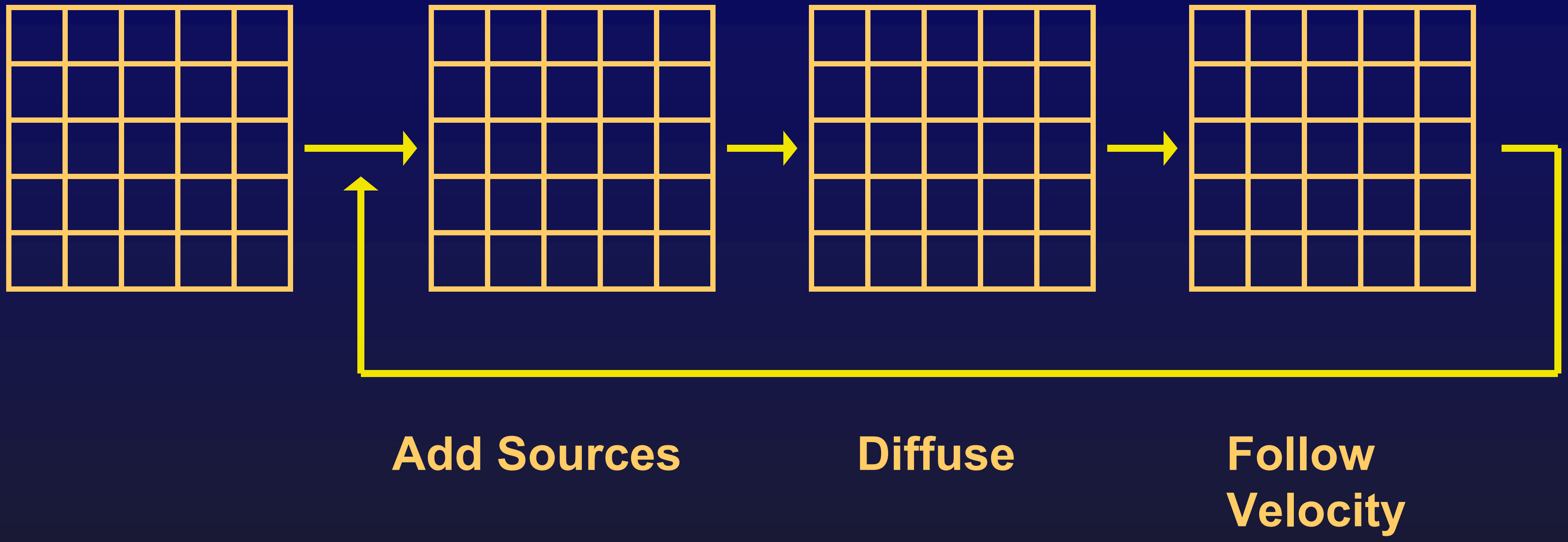
Set Interpolated Value in Cell

```
dens[i,j] = d;
```

Follow Velocity

```
void advect ( float * dens, float * dens0,
float * u, float * v )
{
  int i, j, i0, j0, i1, j1;
  float x, y, s0, t0, s1, t1;

  for ( i=1 ; i<=N ; i++ ) {
    for ( j=1 ; j<=N ; j++ ) {
      x = i-dt*u[i,j]; y = j-dt*v[i,j];
      if (x<0.5) x=0.5; if (x>N+0.5) x=N+0.5;
      if (y<0.5) y=0.5; if (y>N+0.5) y=N+0.5;
      i0=(int)x; i1=i0+1; j0=(int)y; j1=j0+1;
      s1 = x-i0; s0 = 1-s1; t1 = y-j0; t0 = 1-t1;
      dens[i,j] = t0*(s0*dens0[i0,j0]+s1*dens0[i0,j1])+
                  t1*(s0*dens0[i1,j0]+s1*dens0[i1,j1]);
    }
  }
}
```

Simulation




```
void dens_step ()
{
  add_sources(dens);
  SWAP(dens,dens0); diffuse(dens,dens0);
  SWAP(dens,dens0); advect(dens,dens0,u,v);
}
```

Navier-Stokes Equations

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

Velocity

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

Density

Velocity Solver

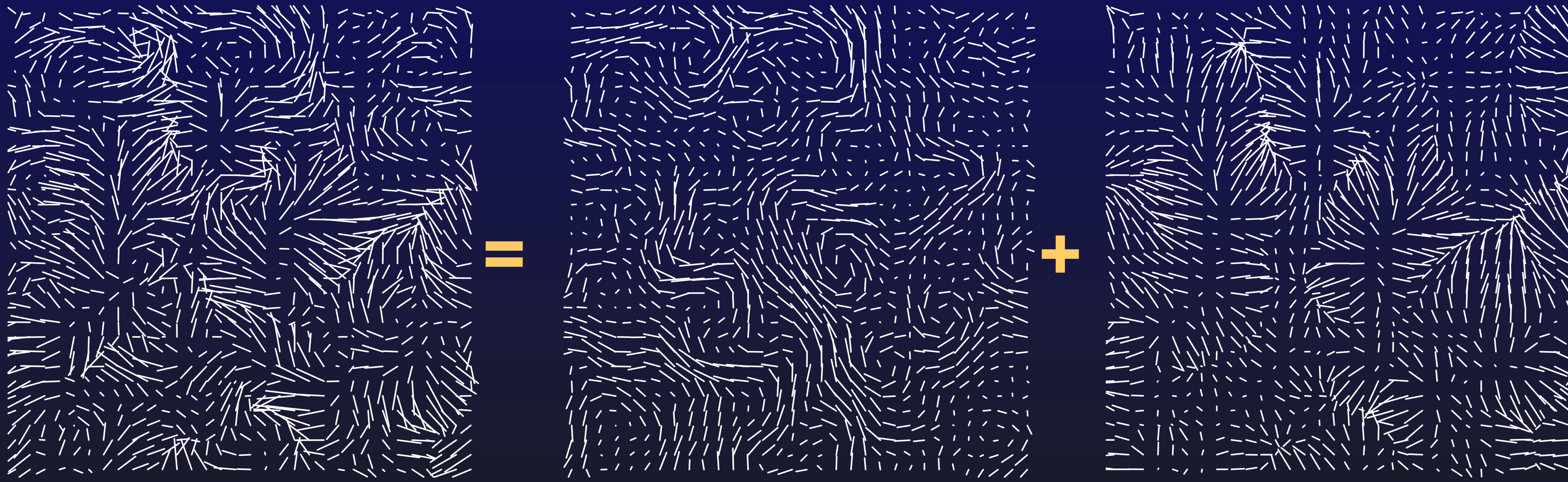
```
void velocity_step ()
{
  add_sources(u);
  add_sources(v);
  SWAP(u,u0); SWAP(v,v0);
  diffuse(u,u0);
  diffuse(v,v0);
  SWAP(u,u0); SWAP(v,v0);
  advect(u,u0,u0,v0);
  advect(v,v0,u0,v0);
  project(u,v,u0,v0);
}
```

```
void dens_step ()
{
  add_sources(dens);

  SWAP(dens,dens0);
  diffuse(dens,dens0);

  SWAP(dens,dens0);
  advect(dens,dens0,u,v);


}
```

Reuse Density Solver Code Except for one Routine…

Projection Step

Hodge Decomposition:



Any Field Mass Conserving Gradient

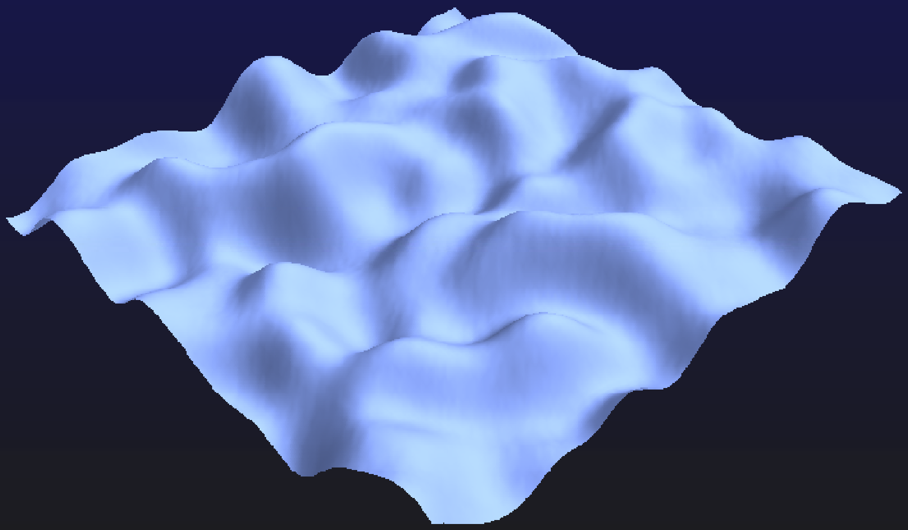
Projection Step

Subtract Gradient Field

Projection Step

Gradient: Direction of steepest Descent of a height field.

```
Gx[i,j] = 0.5*(p[i+1,j]-p[i-1,j])/h
Gy[i,j] = 0.5*(p[i,j+1]-p[i,j-1])/h
```



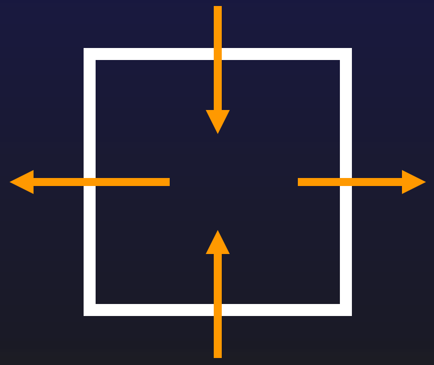
`p[i,j]`

Projection Step

Height Field satisfies a Poisson Equation

```
4*p[i,j]-p[i+1,j]+p[i-1,j]+p[i,j+1]+p[i,j-1] = div[i,j]

div[i,j] = -0.5*h*(u[i+1,j]-u[i-1,j]+v[i,j+1]-v[i,j-1])
```

Ideally div is zero: Flow IN = Flow OUT



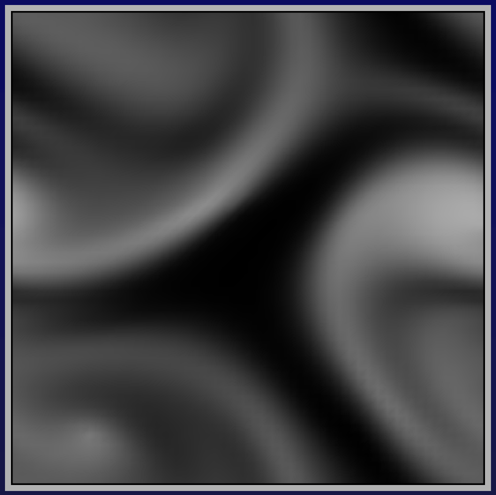
Reuse linear solver of the diffusion step

Projection Step

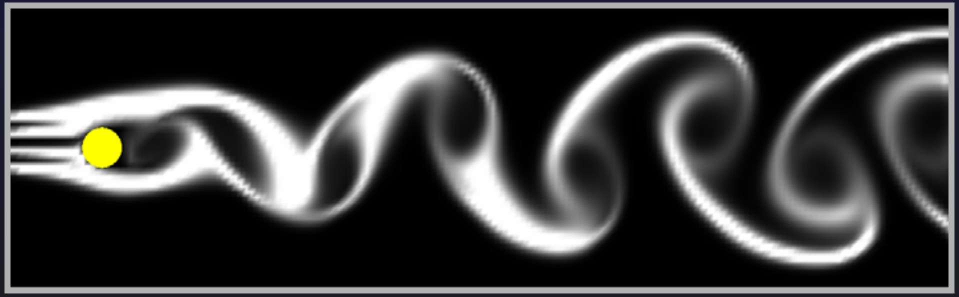
```
void project ( float * u, float * v, float * div, float * p )
{
  int i, j;
  // compute divergence
  for ( i=1 ; i<=N ; i++ ) {
    for ( j=1 ; j<=N ; j++ ) {
      div[i,j] = -0.5*h*(u[i+1,j]-u[i-1,j]+v[i,j+1]-v[i,j-1]);
      p[i,j] = 0.0;
    }
  } set_bnd ( 0, div ); set_bnd ( 0, p );
  // solve Poisson equation
  lin_solve ( p, div, 1, 4 );
  // subtract gradient field
  for ( i=1 ; i<=N ; i++ ) {
    for ( j=1 ; j<=N ; j++ ) {
      u[i,j] -= 0.5*(p[i+1,j]-p[i-1,j])/h;
      v[i,j] -= 0.5*(p[i,j+1]-p[i,j-1])/h;
    }
  }
}
```

Boundaries

Fixed Walls

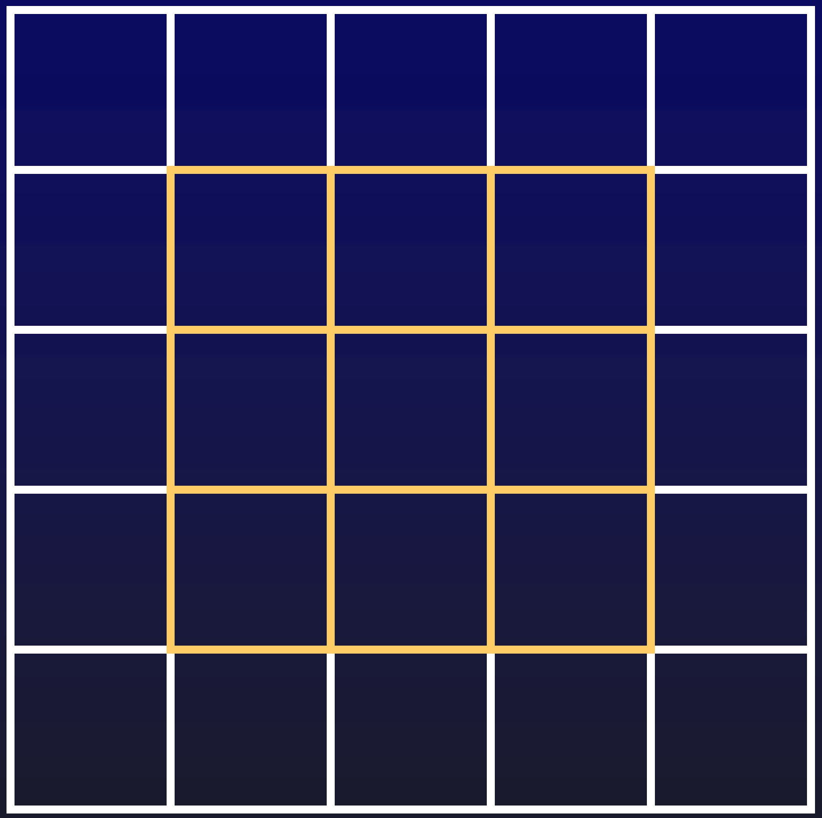


Periodic



Inflow + internal

Boundaries



Add another layer around the grid

Boundaries

	0.5	0.1	0.2	
	0.4	0.2	0.0	
	0.2	0.1	0.0	

Densities: simply copy over values

Boundaries

0.0	0.5	0.1	0.2	0.0
-0.5	0.5	0.1	0.2	-0.2
-0.4	0.4	0.2	0.0	-0.0
-0.2	0.2	0.1	0.0	-0.0
0.0	0.2	0.1	0.0	0.0

U-velocity: zero on vertical boundaries

Boundaries

0.0	-0.5	-0.1	-0.2	0.0
0.5	0.5	0.1	0.2	0.2
0.4	0.4	0.2	0.0	0.0
0.2	0.2	0.1	0.0	0.0
0.0	-0.2	-0.1	-0.0	0.0

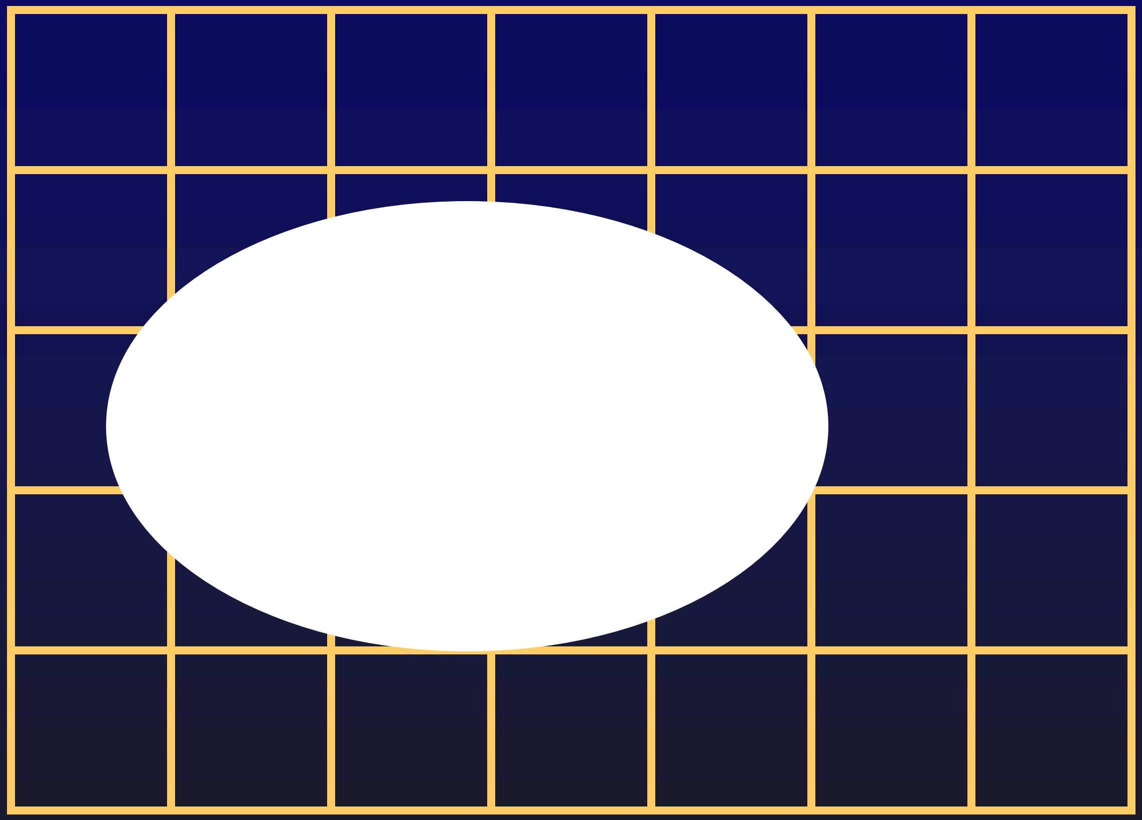
V-velocity: zero on horizontal boundaries

Boundaries

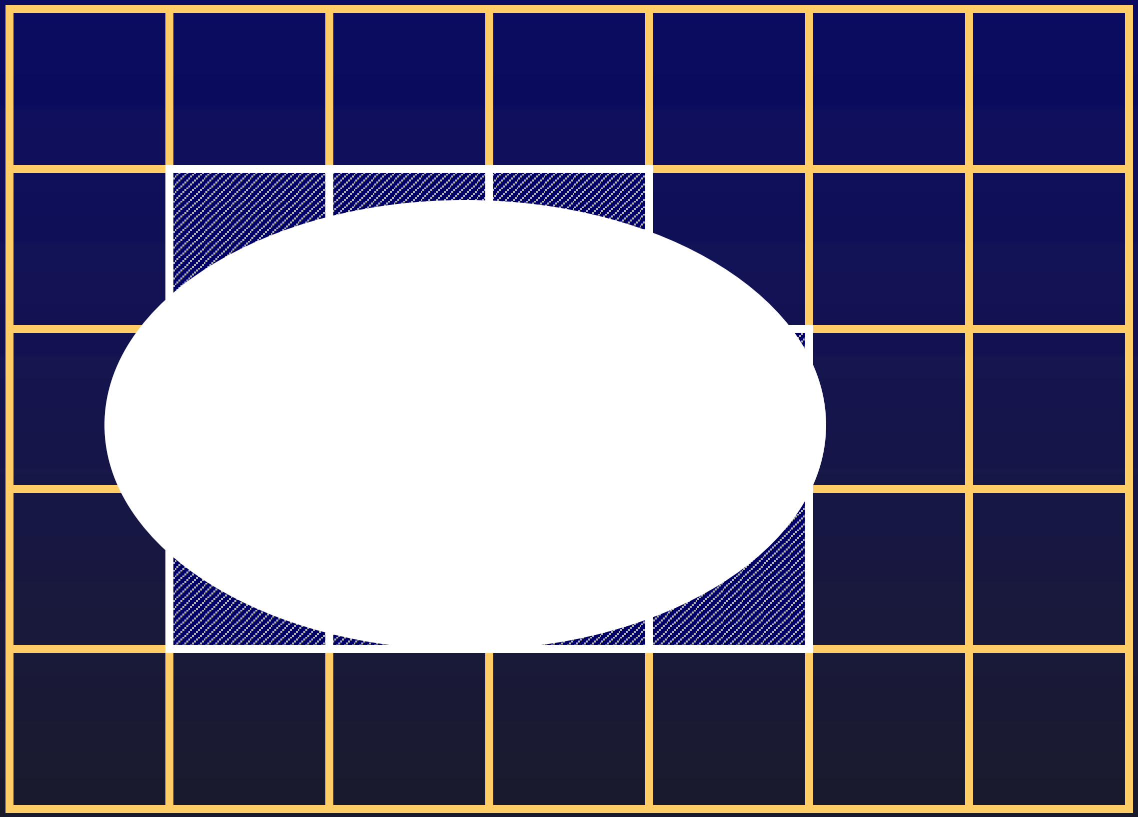
```
void set_bnd ( int b, float * x )
{
  int i;

  for ( i=1 ; i<=N ; i++ ) {
    x[0  ,i  ] = b==1 ? -x[1,i] : x[1,i];
    x[N+1,i  ] = b==1 ? -x[N,i] : x[N,i];
    x[i  ,0  ] = b==2 ? -x[i,1] : x[i,1];
    x[i  ,N+1] = b==2 ? -x[i,N] : x[i,N];
  }
  x[0  ,0  ] = 0.5*(x[1,0  ]+x[0  ,1]);
  x[0  ,N+1] = 0.5*(x[1,N+1]+x[0  ,N]);
  x[N+1,0  ] = 0.5*(x[N,0  ]+x[N+1,1]);
  x[N+1,N+1] = 0.5*(x[N,N+1]+x[N+1,N]);
}
```

Call after every update of the grids

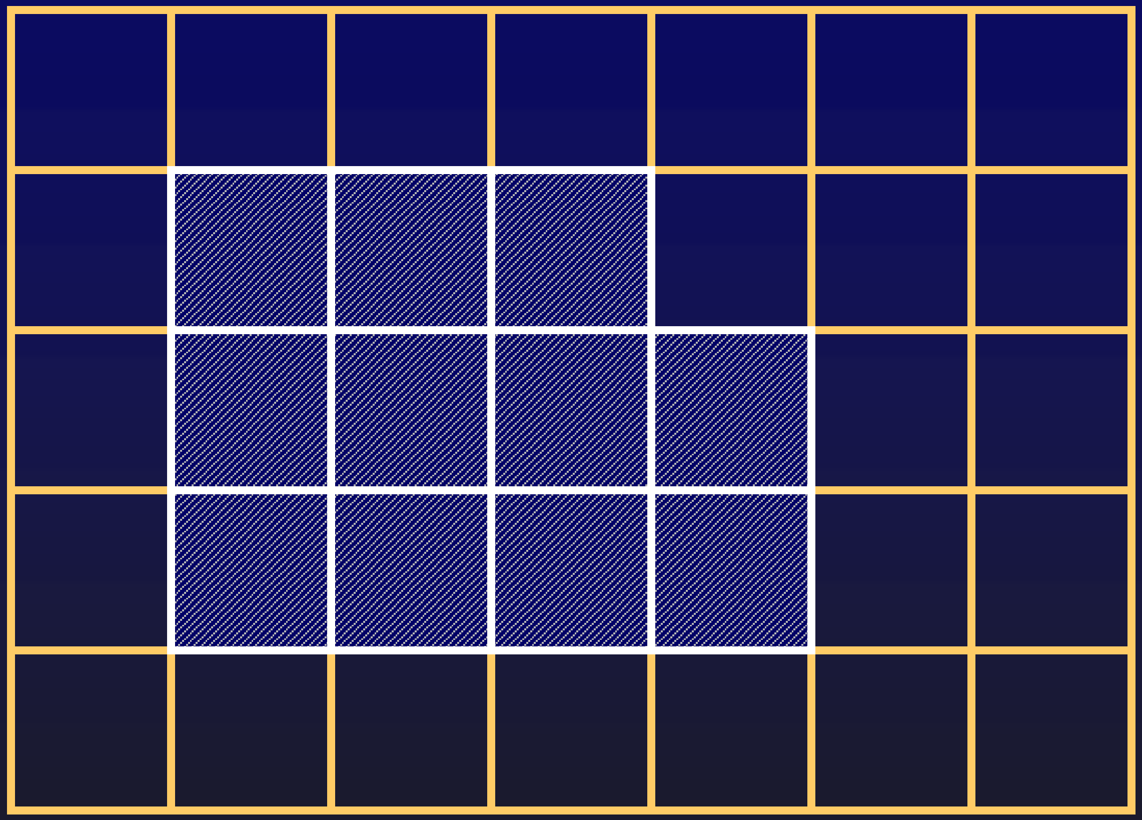
Internal Boundaries



Internal Boundaries



Internal Boundaries



Internal Boundaries

0.0	0.0	0.5	0.1	0.2	0.1	0.3
0.4				0.3	0.2	0.5
0.6					0.5	0.6
0.5					0.8	0.7
0.5	0.1	0.2	0.1	0.2	0.7	0.9

For density

Internal Boundaries

0.0	0.0	0.5	0.1	0.2	0.1	0.3
0.4	0.2	0.5	0.2	0.3	0.2	0.5
0.6	0.6	0.0	0.0	0.4	0.5	0.6
0.5	0.2	0.2	0.1	0.5	0.8	0.7
0.5	0.1	0.2	0.1	0.2	0.7	0.9

For density

The Code

Entire solver in 100 lines of (readable) C-code…

```c
#define IX(i,j) ((i)+(N+2)*(j))
#define SWAP(x0,x) {float * tmp=x0;x0=x;x=tmp;}
#define FOR_EACH_CELL for ( i=1 ; i<=N ; i++ ) {\
                      for ( j=1 ; j<=N ; j++ ) {
#define END_FOR }}

void add_source(int N, float *x, float *s, float dt)
{
  int i, size=(N+2)*(N+2);
  for ( i=0 ; i<size ; i++ ) x[i] += dt*s[i];
}

void set_bnd(int N, int b, float *x)
{
  int i;

  for ( i=1 ; i<=N ; i++ ) {
    x[IX(0  ,i)] = b==1 ? -x[IX(1,i)] : x[IX(1,i)];
    x[IX(N+1,i)] = b==1 ? -x[IX(N,i)] : x[IX(N,i)];
    x[IX(i,0  )] = b==2 ? -x[IX(i,1)] : x[IX(i,1)];
    x[IX(i,N+1)] = b==2 ? -x[IX(i,N)] : x[IX(i,N)];
  }
  x[IX(0  ,0  )] = 0.5f*(x[IX(1,0  )]+x[IX(0  ,1)]);
  x[IX(0  ,N+1)] = 0.5f*(x[IX(1,N+1)]+x[IX(0  ,N)]);
  x[IX(N+1,0  )] = 0.5f*(x[IX(N,0  )]+x[IX(N+1,1)]);
  x[IX(N+1,N+1)] = 0.5f*(x[IX(N,N+1)]+x[IX(N+1,N)]);
}

void lin_solve(int N, int b, float *x, float *x0,
float a, float c)
{
  int i, j, n;

  for ( n=0 ; n<20 ; n++ ) {
    FOR_EACH_CELL
      x[IX(i,j)] = (x0[IX(i,j)]+a*(x[IX(i-1,j)]+
        x[IX(i+1,j)]+x[IX(i,j-1)]+x[IX(i,j+1)]))/c;
    END_FOR
    set_bnd ( N, b, x );
  }
}

void diffuse(int N, int b, float *x, float *x0,
float diff, float dt)
{
  float a=dt*diff*N*N;
  lin_solve ( N, b, x, x0, a, 1+4*a );
}

void advect(int N, int b, float *d, float *d0, float *u, float *v, float dt)
{
  int i, j, i0, j0, i1, j1;
  float x, y, s0, t0, s1, t1, dt0;

  dt0 = dt*N;
  FOR_EACH_CELL
    x = i-dt0*u[IX(i,j)]; y = j-dt0*v[IX(i,j)];
    if (x<0.5f) x=0.5f; if (x>N+0.5f) x=N+0.5f; i0=(int)x; i1=i0+1;
    if (y<0.5f) y=0.5f; if (y>N+0.5f) y=N+0.5f; j0=(int)y; j1=j0+1;
    s1 = x-i0; s0 = 1-s1; t1 = y-j0; t0 = 1-t1;
    d[IX(i,j)] = s0*(t0*d0[IX(i0,j0)]+t1*d0[IX(i0,j1)])+
                 s1*(t0*d0[IX(i1,j0)]+t1*d0[IX(i1,j1)]);
  END_FOR
  set_bnd ( N, b, d );
}

void project(int N, float * u, float * v, float * p, float * div)
{
  int i, j;

  FOR_EACH_CELL
    div[IX(i,j)] = -0.5f*(u[IX(i+1,j)]-u[IX(i-1,j)]+v[IX(i,j+1)]-v[IX(i,j-1)])/N;
    p[IX(i,j)] = 0;
  END_FOR
  set_bnd ( N, 0, div ); set_bnd ( N, 0, p );

  lin_solve ( N, 0, p, div, 1, 4 );

  FOR_EACH_CELL
    u[IX(i,j)] -= 0.5f*N*(p[IX(i+1,j)]-p[IX(i-1,j)]);
    v[IX(i,j)] -= 0.5f*N*(p[IX(i,j+1)]-p[IX(i,j-1)]);
  END_FOR
  set_bnd ( N, 1, u ); set_bnd ( N, 2, v );
}

void dens_step(int N, float *x, float *x0, float *u, float *v, float diff, float dt)
{
  add_source ( N, x, x0, dt );
  SWAP ( x0, x ); diffuse ( N, 0, x, x0, diff, dt );
  SWAP ( x0, x ); advect ( N, 0, x, x0, u, v, dt );
}

void vel_step(int N, float *u, float *v, float *u0, float *v0, float visc, float dt)
{
  add_source ( N, u, u0, dt ); add_source ( N, v, v0, dt );
  SWAP ( u0, u ); diffuse ( N, 1, u, u0, visc, dt );
  SWAP ( v0, v ); diffuse ( N, 2, v, v0, visc, dt );
  project ( N, u, v, u0, v0 );
  SWAP ( u0, u ); SWAP ( v0, v );
  advect ( N, 1, u, u0, u0, v0, dt ); advect ( N, 2, v, v0, u0, v0, dt );
  project ( N, u, v, u0, v0 );
}
```

Guide to the Litterature

(Computational Fluid Dynamics)

Diffusion Step -- Implicit Methods

- Any standard text in numerical methods

Advection Step -- Semi-Lagrangian

- Courant et al., *Comm. Pure & App. Math.*, 1952.
- Weather Forecasting
- Rediscovered many times…

Projection Step -- Projection Methods

- Chorin, *Math. Comput.*, 1969.

Guide to the Litterature
(Computer Graphics)

Vortex Blob – Restricted to 2D

- Upson & Yaeger, Proc. *SIGGRAPH*, 1986.
- Gamito et al., *Eurographics*, 1995.

Explicit Finite Differences -- Unstable

- Foster & Metaxas, *GMIP*, 1996.
- Foster & Metaxas, Proc. *SIGGRAPH*, 1997.
- Chen et al., *IEEE CG&A*, 1997.

Implicit—Semi-Lagrangian -- Stable

- Stam, Proc. *SIGGRAPH*, 1999.

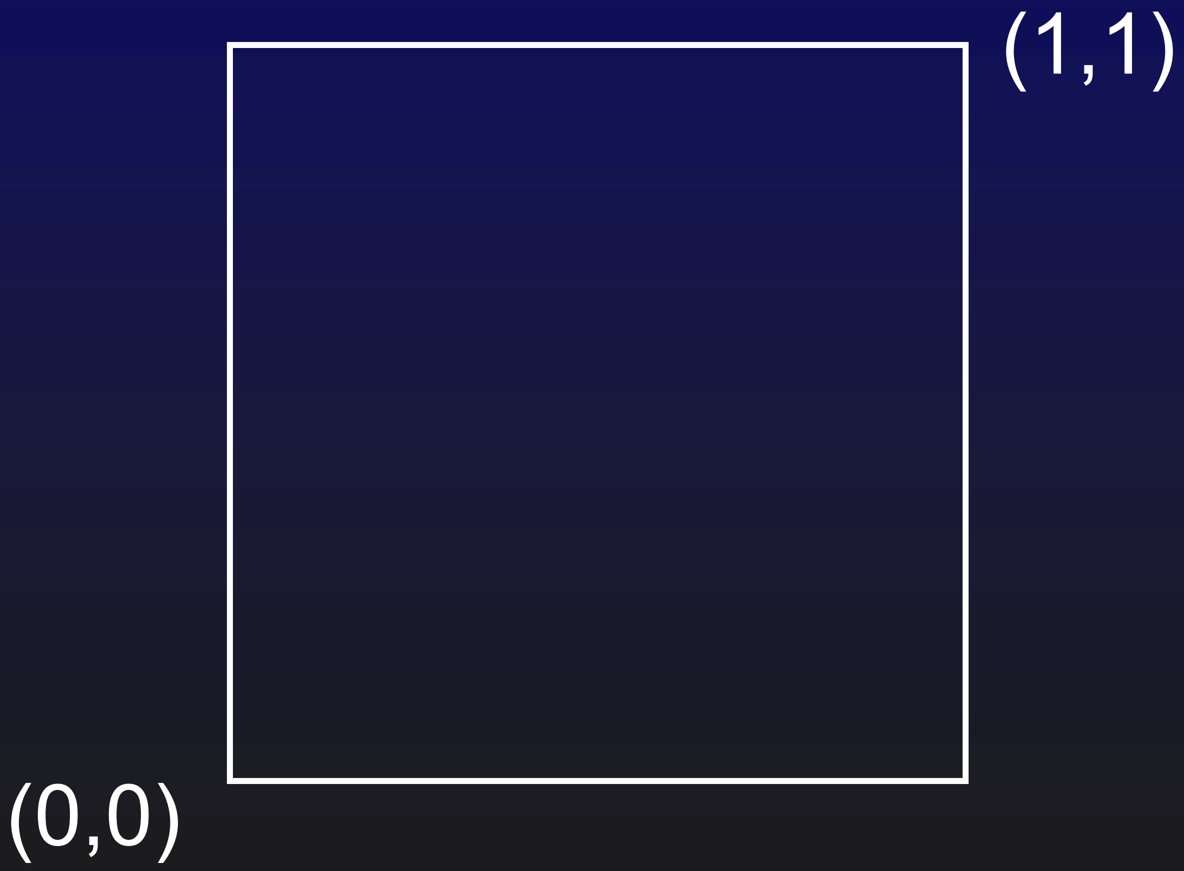
Fedkiw's Group at Stanford

Demo

Show 2D Demos

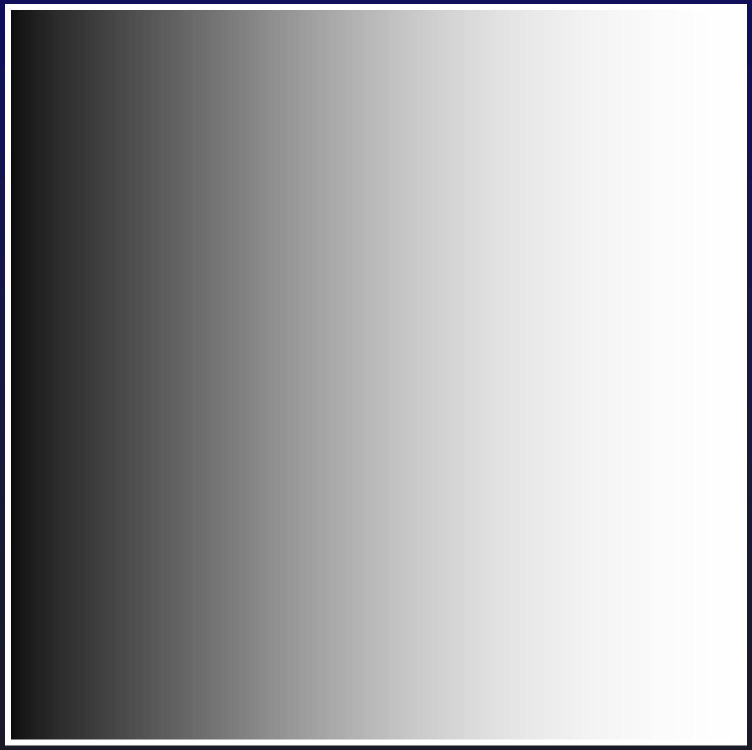
Liquid Textures

Animate texture coordinates

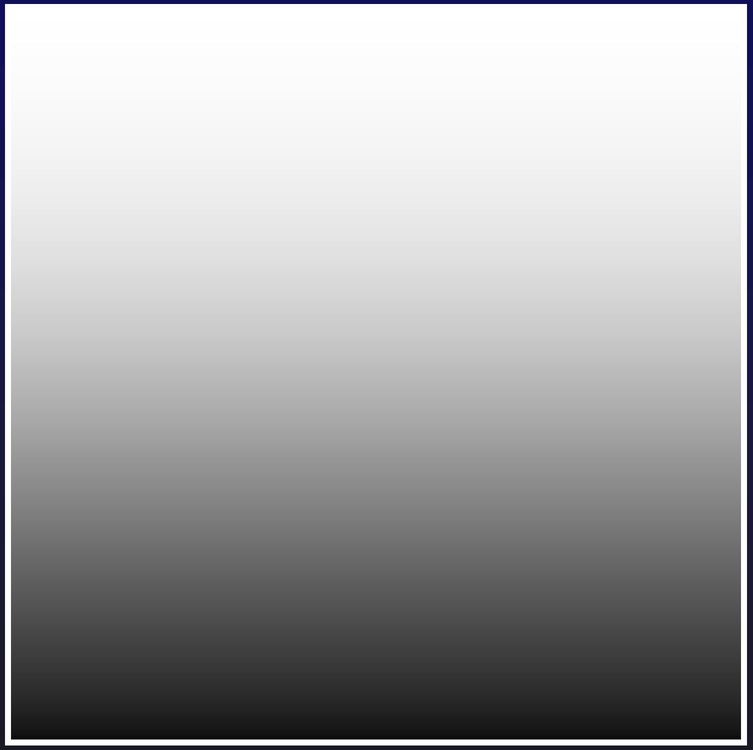


Liquid Textures

Treat texture coordinate as a density

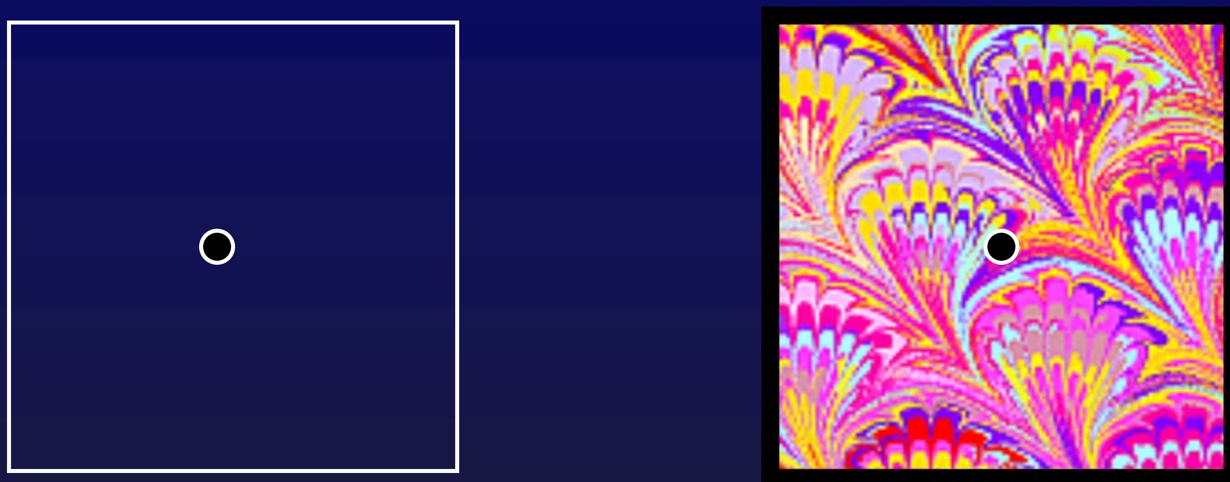


U-coordinate

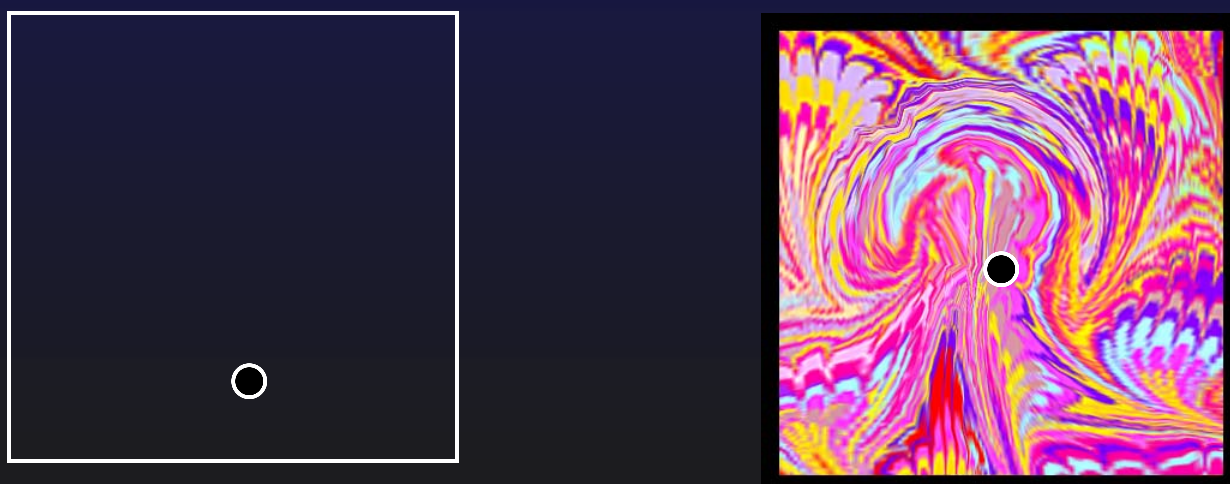


V-coordinate

Liquid Textures



(0.5,0.5)



(0.2,0.52)

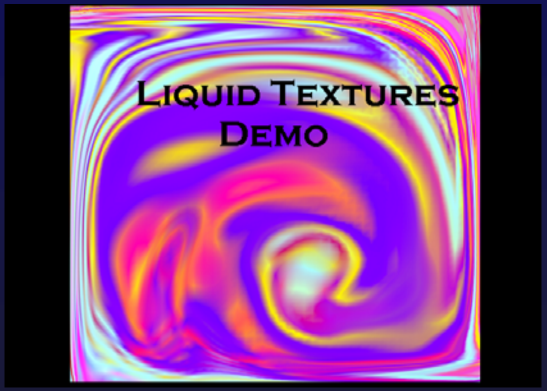
Liquid Textures

```
void tex_step ()
{
  SWAP(u_tex,u0_tex); SWAP(v_tex,v0_tex);
  advect(u_tex,u0_tex,u,v);
  advect(v_tex,v0_tex,u,v);
}
```

Demo


Liquid Textures
Demo


Liquid Textures
Demo

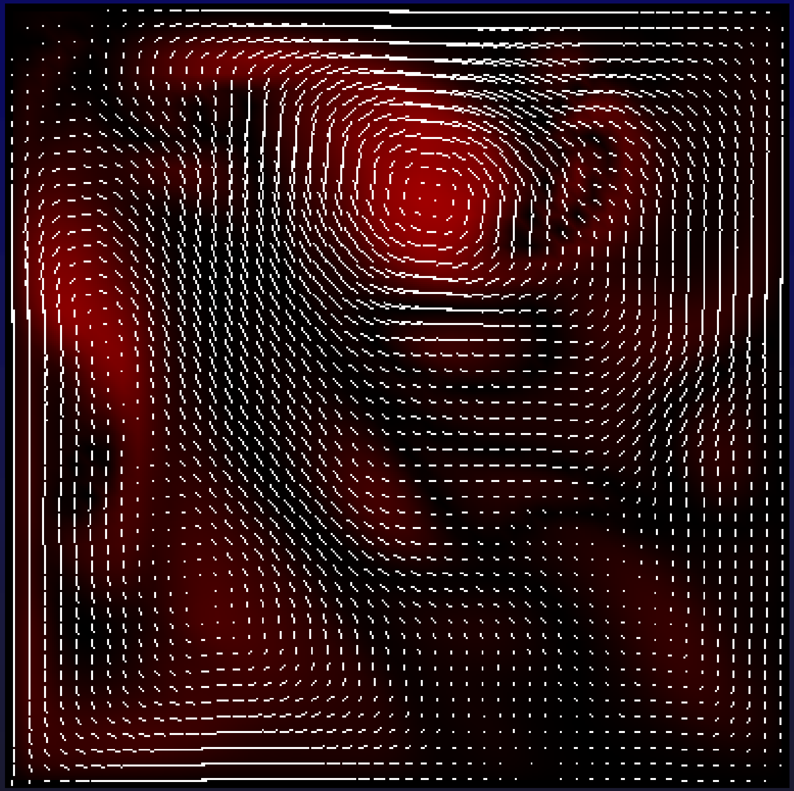

Liquid Textures
Demo

Vorticity Confinement

Technique to defeat dissipation

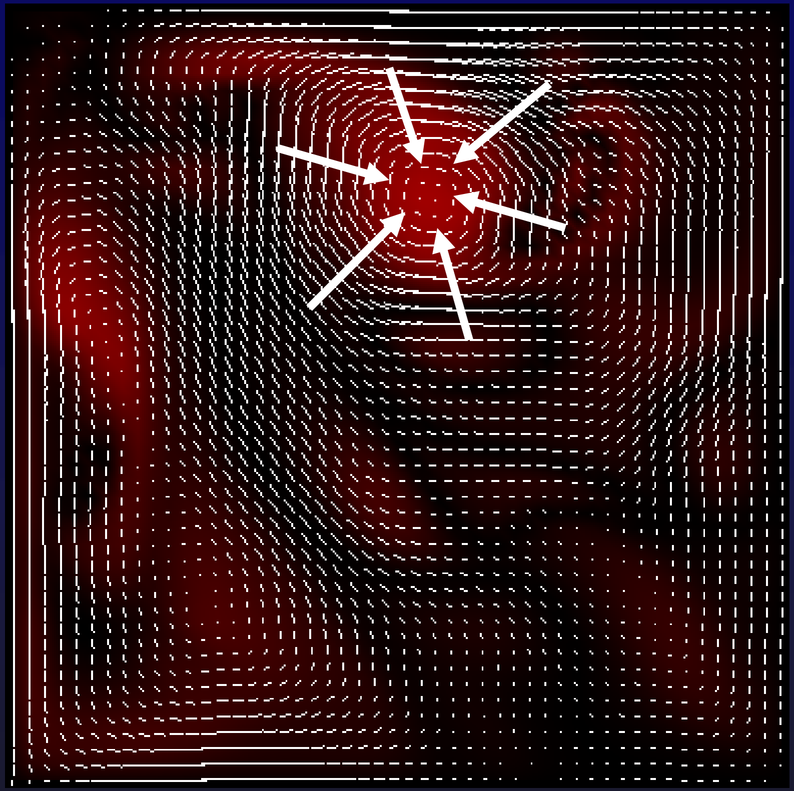
- **Steinhoff, *Physics of Fluids*, 1994.**
- **Fedkiw, Stam & Jensen, *SIGGRAPH*, 2001.**

Vorticity Confinement



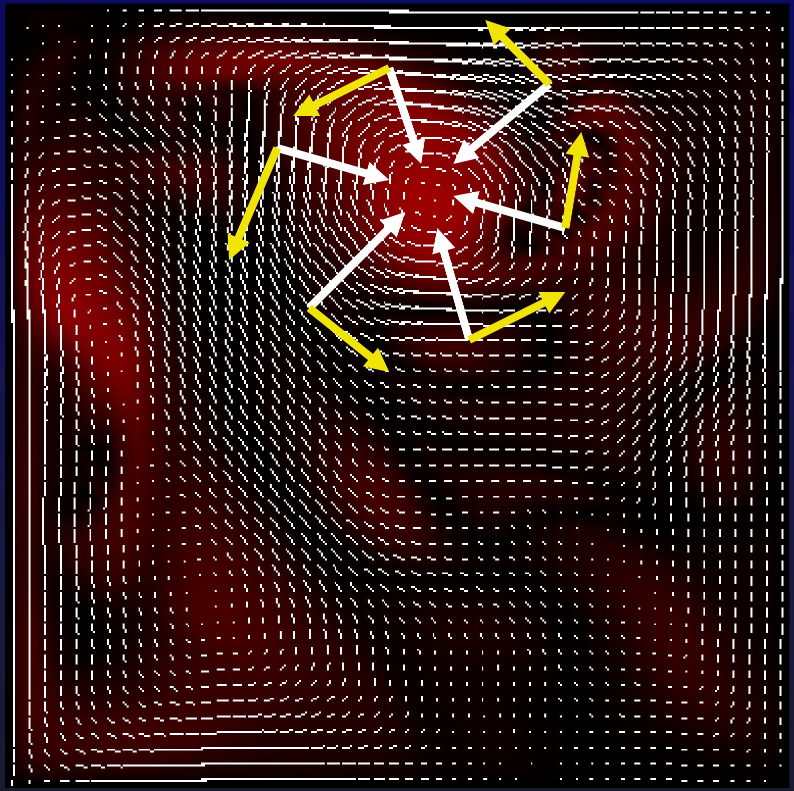
$$\omega = \nabla \times \mathbf{u}$$

Vorticity Confinement



Compute gradient of vorticity

Vorticity Confinement



Add force perpendicular to the gradient

Demo

Vorticity Confinement Demo

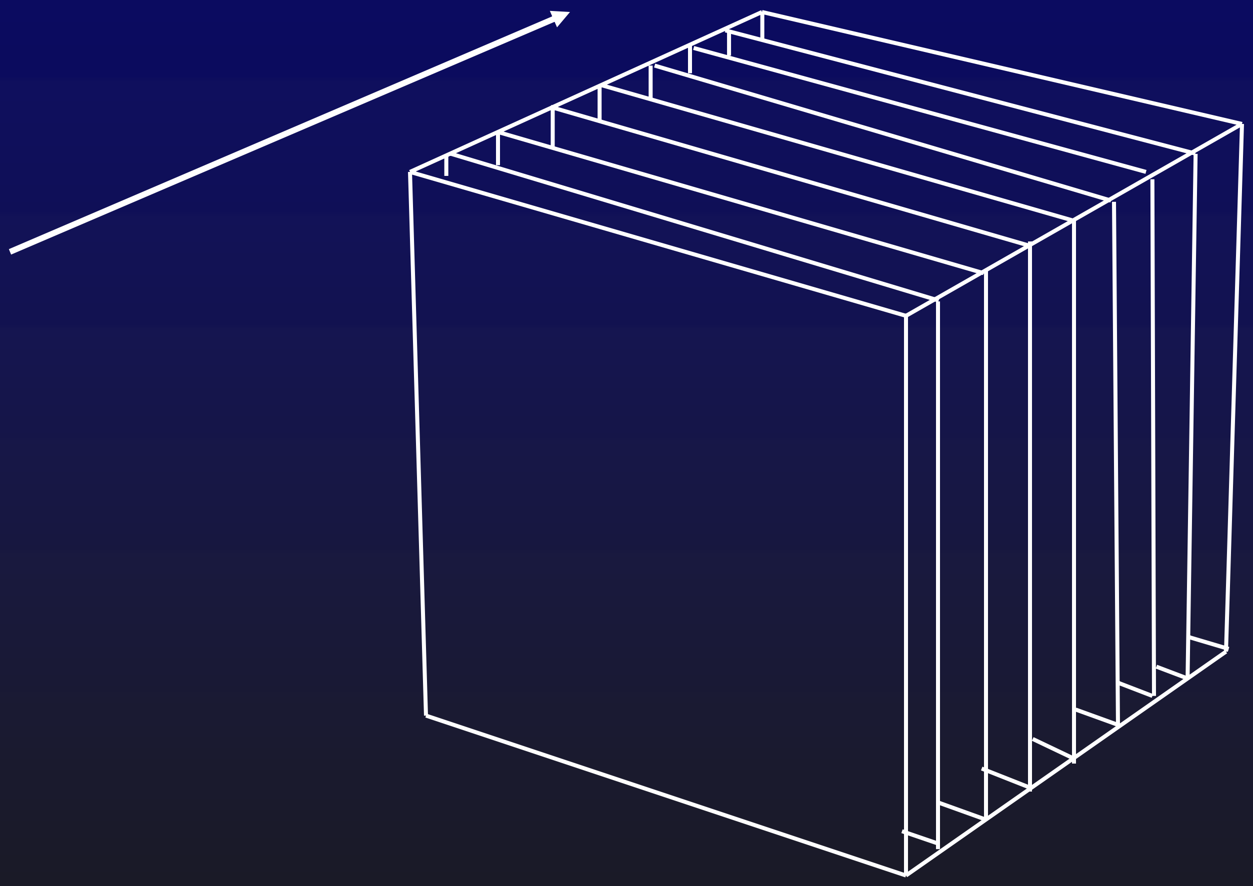
3D Solver

```
for ( i=1 ; i<=N ; i++ ) {
  for ( j=1 ; j<=N ; j++ ) {
    density[i,j] = ...
  }
}
```

Becomes...

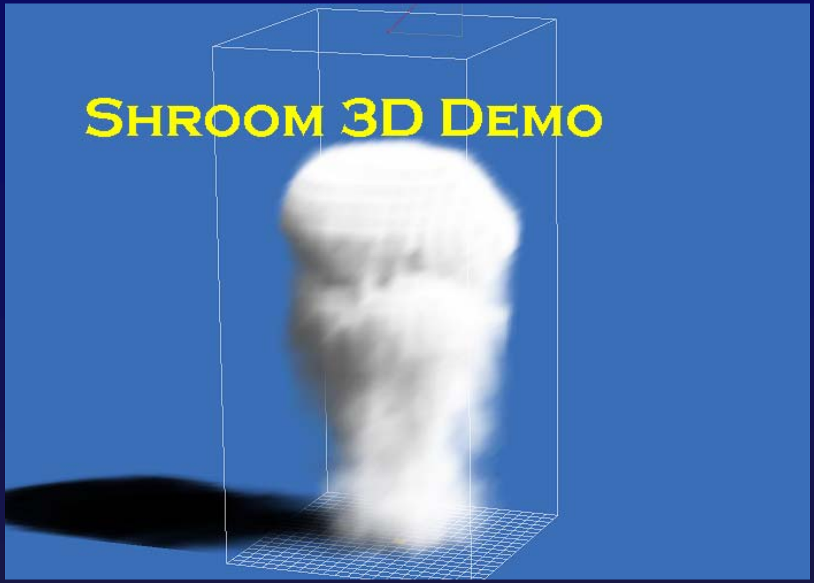
```
for ( i=1 ; i<=N ; i++ ) {
  for ( j=1 ; j<=N ; j++ ) {
    for ( k=1 ; k<=N ; k++ ) {
      density[i,j,k] = ...
    }
  }
}
```

3D Solver

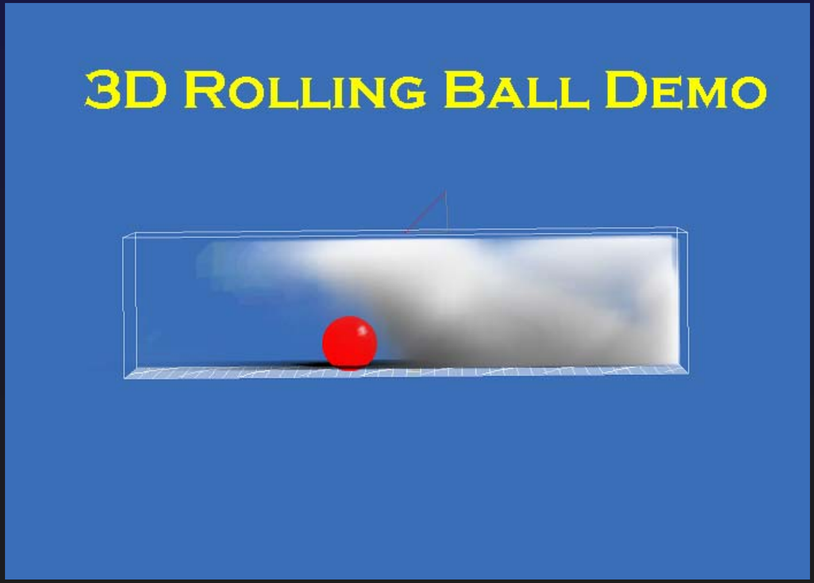


Volume render density

Demo

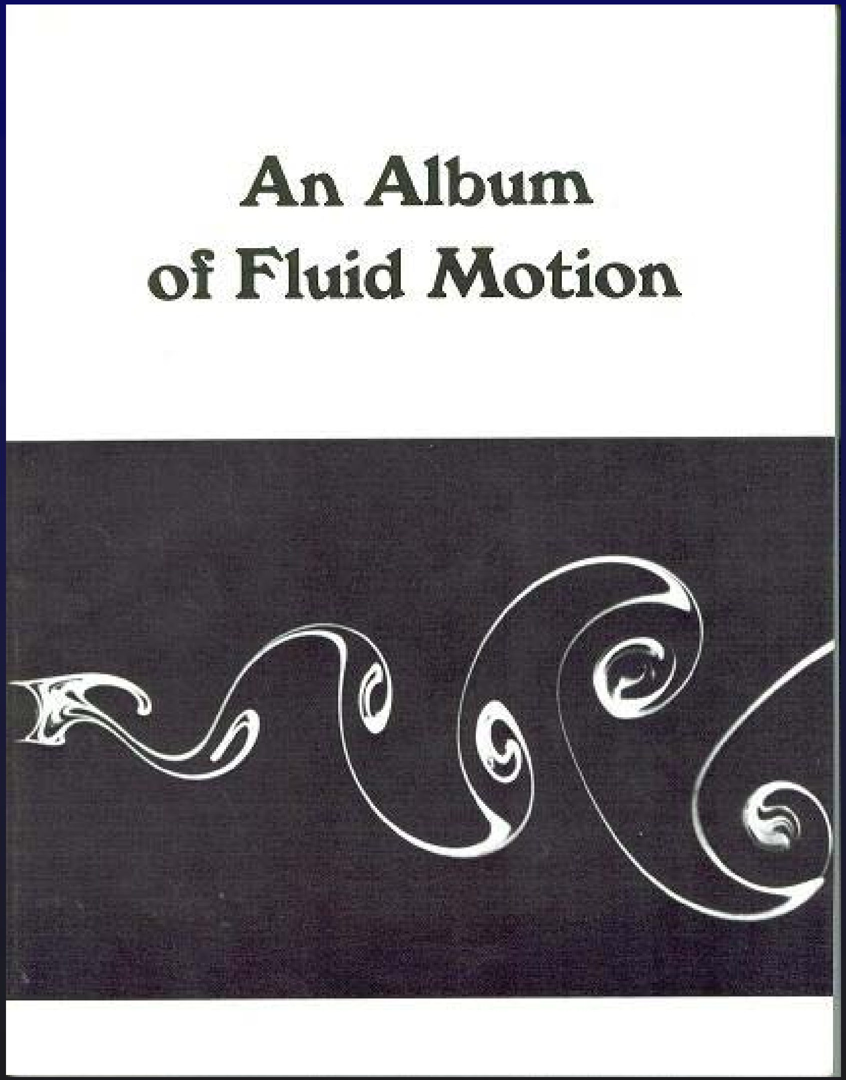

Shroom 3D Demo


3D Falling Ball Demo

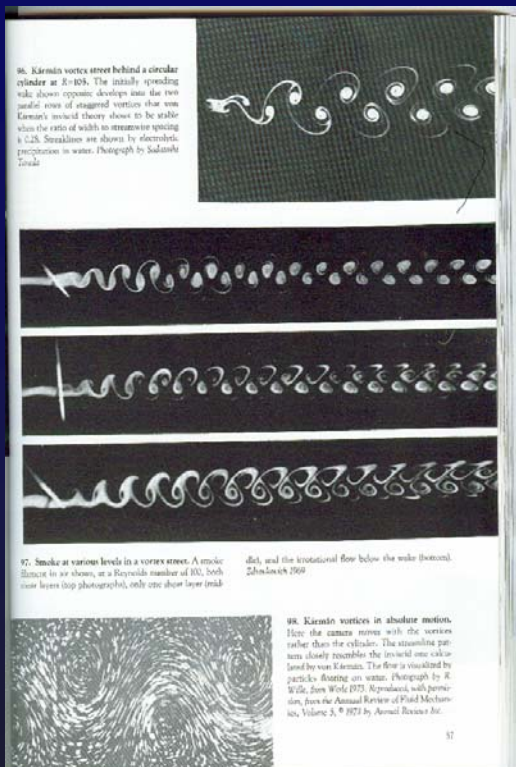

3D Rolling Ball Demo


3D Explosion Demo

An Album Of Fluid Motion



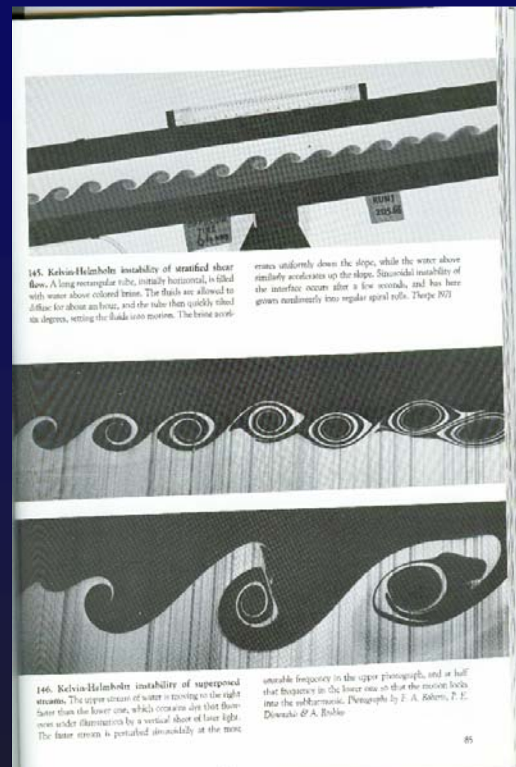
An Album Of Fluid Motion


96. **Kármán vortex street behind a circular cylinder at R=105.** The initially spreading wake shown opposite develops into the two parallel rows of staggered vortices that von Kármán's inviscid theory shows to be stable when the ratio of width to streamwise spacing is 0.28. Streaklines are shown by electrolytic precipitation in water. *Photograph by Sadatoshi Taneda*

97. **Smoke at various levels in a vortex street.** A smoke filament in air shows, at a Reynolds number of 100, both shear layers (top photograph), only one shear layer (middle), and the irrotational flow below the wake (bottom). *Zdravkovich 1969*

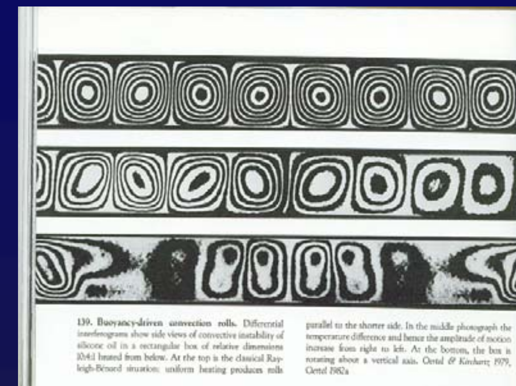
98. **Kármán vortices in absolute motion.** Here the camera moves with the vortices rather than the cylinder. The streamline pattern closely resembles the inviscid one calculated by von Kármán. The flow is visualized by particles floating on water. *Photograph by R. Wille, from Wille 1973. Reproduced, with permission, from the Annual Review of Fluid Mechanics, Volume 5, © 1973 by Annual Reviews Inc.*

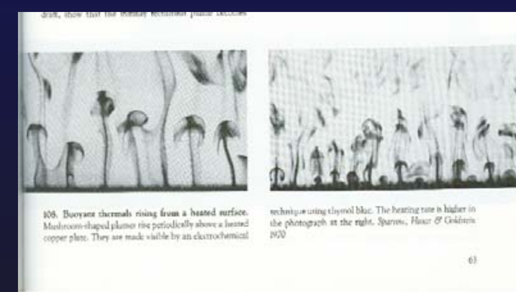
57


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 accelerates 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*

146. **Kelvin-Helmholtz instability of superposed streams.** The upper stream of water is moving to the right faster than the lower one, which contains dye that fluoresces under illumination by a vertical sheet of laser light. The faster stream is perturbed sinusoidally at the most unstable frequency in the upper photograph, and at half that frequency in the lower one so that the motion locks into the subharmonic. *Photographs by F. A. Roberts, P. E. Dimotakis & A. Roshko*

85


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 photograph the temperature difference and hence the amplitude of motion increase from right to left. At the bottom, the box is rotating about a vertical axis. *Oertel & Kirchartz 1979, Oertel 1982a*


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 technique using thymol blue. The heating rate is higher in the photograph at the right. *Sparrow, Husar & Goldstein 1970*

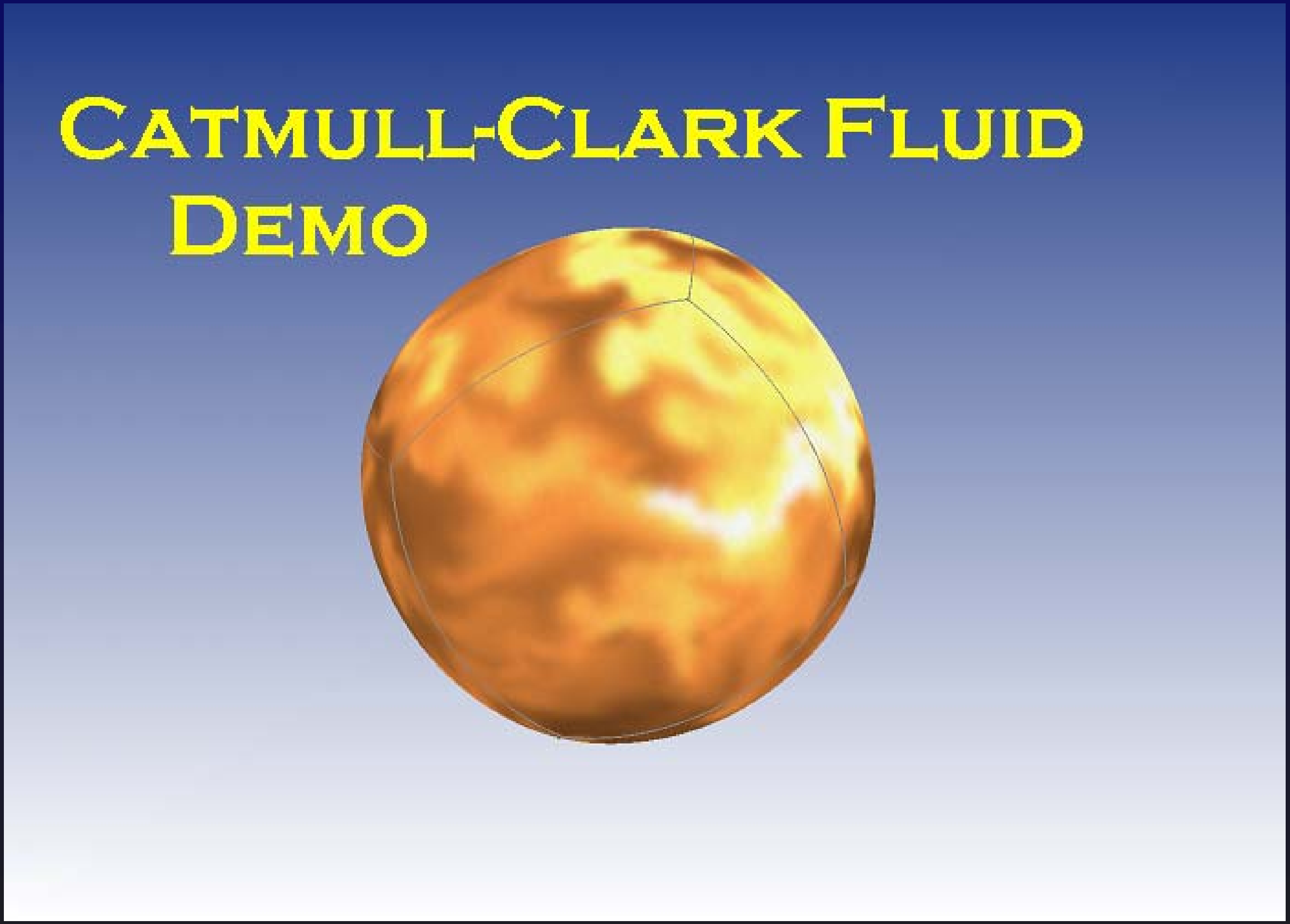
63

Von Karmann

Kelvin-Helmholtz

Rayleigh-Benard

Flows on Surfaces



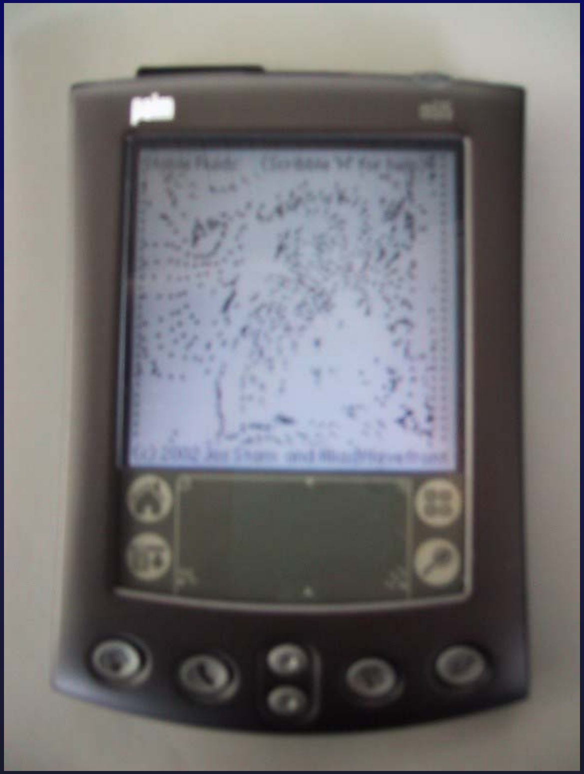
Fluids on PDAs

Fixed point math:

8 bits	.	8 bits

```
#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))
```

Fluids on PDAs



Palm

PocketPC

MAYA Fluid Effects

Fluid Solver now available
in Maya 4.5 Unlimited

Download the screensaver

`http://www.aliaswavefront.com`

MAYA Fluid Effects






Future Work

- Real-Time Water

- Out of the box

- Smart Texture Maps

- (...)