Looking back at Thursday's lectures

Thomas Sangild
GPGPU lecture
University of Aarhus
30/11-2004

Vector field operations

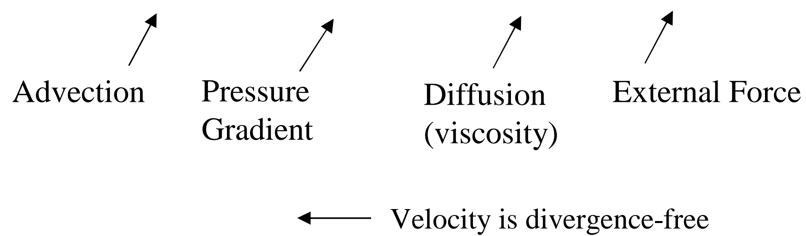
- Pressure (scalar field) p: $R^m \rightarrow R$; $m \in \{2,3\}$
- Velocity (vector field) **u**: $R^m \rightarrow R^m$; $m \in \{2,3\}$
  - $\nabla = (\partial/\partial x, \partial/\partial y)$

Table 38-1. Vector Calculus Operators Used in Fluid Simulation

Operator	Definition	Finite Difference Form
Gradient	$\nabla p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}\right)$	$\frac{p_{i+1,j} - p_{i-1,j}}{2\delta x}, \frac{p_{i,j+1} - p_{i,j-1}}{2\delta y}$
Divergence	$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$	$\frac{u_{i+1,j} - u_{i-1,j}}{2\delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\delta y}$
Laplacian	$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2}$	$\frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{(\delta x)^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{(\delta y)^2}$

Navier-Stokes Equations

- Describe flow of an incompressible fluid



Source: Mark Harris / GPGPU tutorial@Siggraph 04

A few more words on

- numerical integration and
- solutions to linear systems A**x**=**b**

- as they appear in the stable fluid simulation articles.

Numerical integration (1)

- Explicit time step
  - Stability depends on f

$$\frac{\partial \mathbf{u}(\mathbf{x},t)}{\partial t} = f(\mathbf{u}(\mathbf{x},t),t) \Rightarrow$$
$$\frac{\mathbf{u}(\mathbf{x},t+h)-\mathbf{u}(\mathbf{x},t)}{h} \approx f(\mathbf{u}(\mathbf{x},t),t) \Rightarrow$$
$$\mathbf{u}(\mathbf{x},t+h) \approx \mathbf{u}(\mathbf{x},t) + hf(\mathbf{u}(\mathbf{x},t),t)$$

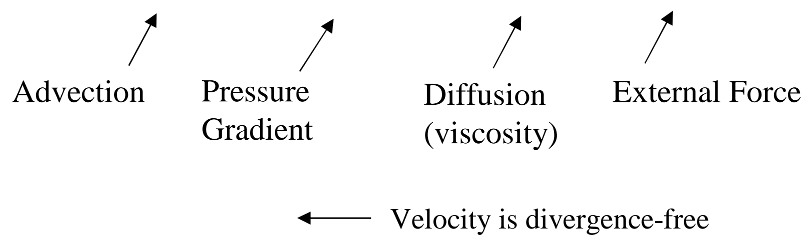
Numerical integration (2)

- Implicit time step
  - Stable for large timesteps

$$\frac{\partial \mathbf{u}(\mathbf{x},t)}{\partial t} = f(\mathbf{u}(\mathbf{x},t+h),t+h) \Rightarrow$$
$$\frac{\mathbf{u}(\mathbf{x},t+h)-\mathbf{u}(\mathbf{x},t)}{h} \approx f(\mathbf{u}(\mathbf{x},t+h),t+h) \Rightarrow$$
$$\mathbf{u}(\mathbf{x},t+h) \approx \mathbf{u}(\mathbf{x},t) + hf(\mathbf{u}(\mathbf{x},t+h),t+h)$$

Navier-Stokes Equations

- Describe flow of an incompressible fluid



Source: Mark Harris / GPGPU tutorial@Siggraph 04

The Helmholtz-Hodge decomposition

Helmholtz-Hodge Decomposition Theorem

A vector field $\mathbf{w}$ on D can be uniquely decomposed in the form:

$$\mathbf{w} = \mathbf{u} + \nabla p, \qquad (7)$$

where $\mathbf{u}$ has zero divergence and is parallel to ∂D; that is, $\mathbf{u} \cdot \mathbf{n} = 0$ on ∂D.

Proof in: Chorin and Marsden. A mathematical introduction to fluid mechanics. 1993.

From: Harris MJ. Fast Fluid Dynamics Simulation on the GPU. Chapter 38. GPU Gems. 2004.

Realizations

First Realization

Solving the Navier-Stokes equations involves three computations to update the velocity at each time step: advection, diffusion, and force application. The result is a new velocity field, w, with *nonzero* divergence. But the continuity equation requires that we end each time step with a divergence-free velocity. Fortunately, the Helmholtz-Hodge Decomposition Theorem tells us that the divergence of the velocity can be corrected by subtracting the gradient of the resulting pressure field:

$$\mathbf{u} = \mathbf{w} - \nabla p. \tag{8}$$

From: Harris MJ. Fast Fluid Dynamics Simulation on the GPU. Chapter 38. GPU Gems. 2004.

Realizations

Second Realization

The theorem also leads to a method for computing the pressure field. If we apply the divergence operator to both sides of Equation 7, we obtain:

$$\nabla \cdot \mathbf{w} = \nabla \cdot (\mathbf{u} + \nabla p) = \nabla \cdot \mathbf{u} + \nabla^2 p. \tag{9}$$

But since Equation 2 enforces that $\nabla \cdot \mathbf{u} = 0$, this simplifies to:

$$\nabla^2 p = \nabla \cdot \mathbf{w}, \tag{10}$$

which is a Poisson equation (see Section 38.2.3) for the pressure of the fluid, sometimes called the *Poisson-pressure equation*. This means that after we arrive at our divergent velocity, w, we can solve Equation 10 for p, and then use w and p to compute the new divergence-free field, u, using Equation 8. We'll return to this later.

From: Harris MJ. Fast Fluid Dynamics Simulation on the GPU. Chapter 38. GPU Gems. 2004.

Projection operator

- Define a projection operator $\mathcal{P}$ that projects a vector field **w** onto divergence-free component **u**.
- $\mathcal{P}\mathbf{w} = \mathcal{P}\mathbf{u} + \mathcal{P}(\nabla p)$
  - Since by definition $\mathcal{P}\mathbf{w} = \mathcal{P}\mathbf{u} = \mathbf{u}$ then $\mathcal{P}(\nabla p)=0$
- And then

$$\mathbb{P}\frac{\partial \mathbf{u}}{\partial t} = \mathbb{P}\left(-(\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{u} + \mathbf{F}\right)$$

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbb{P}\left(-(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + \mathbf{F}\right)$$

Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - Advect:
  - Diffuse:
  - Solve for pressure:
  - Subtract pressure gradient:

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Algorithm

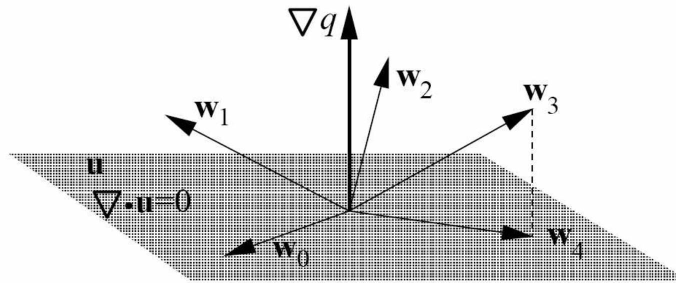


Figure 1: One simulation step of our solver is composed of steps. The first three steps may take the field out of the space of divergent free fields. The last projection step ensures that the field is divergent free after the entire simulation step.

From: Stam J. Stable Fluids. Siggraph 1999.

Algorithm

- Break it down [Stam 2000]:
  - Add forces:

Source: Mark Harris / GPGPU tutorial@Siggraph 04

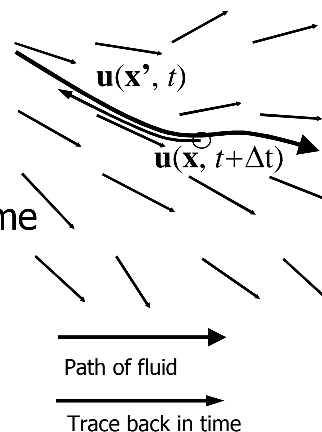
Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - Advect:

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Advection

- Advection: quantities in a fluid are carried along by its velocity
- Want velocity at position x at new time t + Δt
- Follow velocity field back in time from x : (x - $w_1\Delta$t)
  - Like tracing particles!
  - Simple in a fragment program



Source: Mark Harris / GPGPU tutorial@Siggraph 04

Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - Advect:
  - Diffuse:

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Implicit integration

$$\frac{dw_2}{dt} = \frac{w_2(x, t+h) - w_2(x, t)}{h} = \nu\nabla^2 w_2(x, t+h) \Rightarrow$$

$$w_2(x, t+h) = w_2(x, t) + h\nu\nabla^2 w_2(x, t+h) \Rightarrow$$

$$(I - h\nu\nabla^2)\underbrace{w_2(x, t+h)}_{\mathbf{w}_3(\mathbf{x})} = w_2(x, t)$$

Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - Advect:
  - Diffuse:
  - Solve for pressure:

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - Advect:
  - Diffuse:
  - Solve for pressure:
  - Subtract pressure gradient:

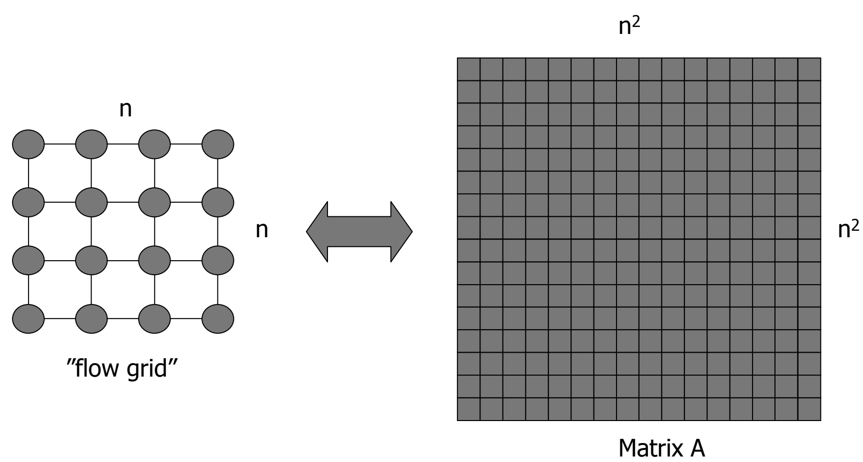
Source: Mark Harris / GPGPU tutorial@Siggraph 04

Jacobi iteration

- To solve linear system A**x**=**b**, or
  -

$$\nabla^2 p = \nabla \cdot \mathbf{w}_3$$

Grid vs 'Matrix A' layout



$$\nabla^2 p = \nabla \cdot \mathbf{w}_3$$

Laplacian	$\nabla^2 p = \dfrac{\partial^2 p}{\partial x^2} + \dfrac{\partial^2 p}{\partial y^2}$	$\dfrac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{(\delta x)^2} + \dfrac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{(\delta y)^2}$

With grid spacing $\delta x = \delta y = 1$:

$A = \nabla^2 =$

-4	1			1											
1	-4	1			1										
	1	-4	1			1									
		1	-4	0			1								
1			0	-4	1			1							
	1			1	-4	1			1						
		1			1	-4	1			1					
			1			1	-4	0			1				
				1			0	-4	1			1			
					1			1	-4	1			1		
						1			1	-4	1			1	
							1			1	-4	0			1
								1			0	-4	1		
									1			1	-4	1	
										1			1	-4	1
											1			1	-4

$\mathbf{b} = \nabla \cdot \mathbf{w}_3$
(evaluated at each grid point)

With $\mathbf{x}=\mathbf{p}$ now solve $A\mathbf{x}=\mathbf{b}$ using Jacobi iteration

Jacobi iteration

At the $(i+1)$st iteration, the vector $x^{(i+1)}$ is calculated by

$$x_j^{(i+1)} = \frac{1}{a_{jj}} \left(- \sum_{\substack{k=1 \\ k \neq j}}^{n} a_{jk} x_k^{(i)} + b_j \right) \qquad j = 1, \ldots, n \tag{3}$$

```
void jacobi( half2      coords : WPOS,
                        out half4       xNew   : COLOR,
                        uniform half        alpha,
                        uniform half        rBeta, // reciprocal beta
                        uniform samplerRECT x,     // x vector (Ax = b)
                        uniform samplerRECT b)     // b vector (Ax = b)
{

half4 xL, xR, xB, xT;
h4texRECTneighbors(x, coords, xL, xR, xB, xT);

half4 bC = h4texRECT(b, coords);

xNew = (xL + xR + xB + xT + alpha * bC) * rBeta;
}
```

Boundary Conditions

- Various types of BCs
  - No-slip, free-slip, outflow, inflow, periodic, etc.
- Demo uses "no-slip" velocity BCs
  - V = 0 at boundaries.
- Pressure: pure Neumann BCs
  - Have to set them each iteration of solver
  - p(boundary) = p(nearest non-boundary neighbor)

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Slab computation (geeb) - 1

```
template
<
  class RenderTargetPolicy,
  class GLStatePolicy,
  class VertexPipePolicy,
  class FragmentPipePolicy,
  class ComputePolicy,
  class UpdatePolicy
>
class SlabOp : public RenderTargetPolicy,
               public GLStatePolicy,
               public VertexPipePolicy,
               public FragmentPipePolicy,
               public ComputePolicy,
               public UpdatePolicy
{
public:
    SlabOp()  {}
    ~SlabOp() {}

    void Compute();
};
```

Slab computation (geeb) - 2

```
void  SlabOp::Compute()
{
   // Activate the output slab, if necessary
   ActivateRenderTarget(); // RenderTargetPolicy

   // Set the necessary state for the slab operation
   GLStatePolicy::SetState();
   VertexPipePolicy::SetState();
   FragmentPipePolicy::SetState();

   SetViewport(); // UpdatePolicy

   // Perform the slab operation
   ComputePolicy::Compute();

   // Put the results of the operation into the output slab.
   UpdateOutputSlab(); // UpdatePolicy

   // Reset state
   FragmentPipePolicy::ResetState();
   VertexPipePolicy::ResetState();
   GLStatePolicy::ResetState();

   // Deactivate the output slab, if necessary
   DeactivateRenderTarget(); // RenderTargetPolicy
 }
};
```

CopyTexGLUpdatePoliy

```
class CopyTexGLUpdatePolicy
{
public:

void SetOutputTexture(GLuint id, int width, int height) { ... }
void SetOutputRectangle(int xOffset, int yOffset, int x, int y, int width, int height) { ... }
void SetViewport() { ... }

void UpdateOutputSlab()
 {
   glBindTexture(GL_TEXTURE_RECTANGLE_NV, _iOutputTexture);
   glCopyTexSubImage2D(GL_TEXTURE_RECTANGLE_NV,
                0, _iXOffset, _iYOffset, _iX, _iY, _iWidth, _iHeight);
   glViewport(_iOldVP[0], _iOldVP[1], _iOldVP[2], _iOldVP[3]);
 }
}
```

Update function(1)

```
//---------------
//   Advect
//---------------

// Advect velocity.
_advect.SetFragmentParameter1f( "dissipation", 1 );
_advect.SetOutputTexture( _iTextures[TEXTURE_VELOCITY], _iWidth, _iHeight );
_advect.SetTextureParameter( "x", _iTextures[TEXTURE_VELOCITY] );
_advect.Compute();

// Advect "ink", a passive scalar carried by the flow.
_advect.SetFragmentParameter1f( "dissipation", _rInkLongevity );
_advect.SetOutputTexture( _iTextures[TEXTURE_DENSITY], _iWidth, _iHeight);
_advect.SetTextureParameter("x", _iTextures[TEXTURE_DENSITY]);
_advect.Compute();
```

Update function(2)

```
//---------------
//   Diffuse
//---------------

float centerFactor = _dx * _dx / (_rViscosity * _rTimestep);
float stencilFactor = 1.0f / (4.0f + centerFactor);

_poissonSolver.SetTextureParameter("x", _iTextures[TEXTURE_VELOCITY]);
_poissonSolver.SetTextureParameter("b", _iTextures[TEXTURE_VELOCITY]);
_poissonSolver.SetFragmentParameter1f("alpha", centerFactor);
_poissonSolver.SetFragmentParameter1f("rBeta", stencilFactor);
_poissonSolver.SetTexCoordRect(0, 0, 0, _iWidth, _iHeight);
_poissonSolver.SetOutputTexture(_iTextures[TEXTURE_VELOCITY], _iWidth, _iHeight);

for (i = 0; i < _iNumPoissonSteps; ++i)
    _poissonSolver.Compute();
```

Fragment programs

- How are they implemented?
  - Take a look yourself!
  - The code is very well documented.