Fluid Simulation on the GPU

GPGP Course Presentation

Huai-Ping Lee

Outline

- Navier-Stokes based methods
- Lattice Boltzmann method
- Summary and Comparison

Navier-Stokes Equations for Fluid Simulation on the GPU

Navier-Stokes Equations

- Macroscopic behaviors of incompressible fluids

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 & (1) \\ \dfrac{\partial \mathbf{u}}{\partial t} = \underbrace{-(\mathbf{u} \cdot \nabla)\mathbf{u}}_{\text{advection}} - \underbrace{\dfrac{1}{\rho}\nabla p}_{\text{pressure}} + \underbrace{\nu \nabla^2 \mathbf{u}}_{\text{diffuse}} + \underbrace{\mathbf{f}}_{\text{external force}} & (2) \end{cases}$$

$\mathbf{u}(\mathbf{x}, t)$: velocity of fluid (vector field);

ρ : density of fluid (constant); $p(\mathbf{x}, t)$: pressure (scalar field);

$\mathbf{f}(\mathbf{x}, t)$: external force (vector field)

$$\mathbf{u} = \begin{bmatrix} u(\mathbf{x}, t) \\ v(\mathbf{x}, t) \end{bmatrix}, \text{ where } \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} \text{ for 2D cases;}$$

Notation—Vector Calculus

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

Derivation of Navier-Stokes Equations

- Eq. 1: conserve mass
  - The integral over the mass of the fluid = constant, and the density is constant
  - So the amount of flux = 0, therefore the flux in each small area = 0
  - By divergence theorem, flux density is div(**u**)

$$\int_{\partial\Omega_t} \mathbf{u}(\mathbf{x},t)\cdot\mathbf{n}\,ds = \int_{\Omega_t} div(\mathbf{u}(\mathbf{x},t))\,d\mathbf{x}$$

Derivation of Navier-Stokes Equations

- Eq. 2: conserve momentum

$$\mathbf{m}(t) = \int_{\Omega_t} \rho(\mathbf{x},t) \cdot \mathbf{u}(\mathbf{x},t) d\mathbf{x}$$

  - Newton's second law:

$$\frac{d}{dt}\mathbf{m}(t) = \sum acting\ \ forces$$

Derivation of Navier-Stokes Equations

- There are two kinds of acting forces
  - Body force: given by the force density per unit volume $\mathbf{f}(\mathbf{x}, t)$ $\mathbf{F}_b = \int_{\Omega_t} \rho(\mathbf{x},t)\cdot\mathbf{f}(\mathbf{x},t)d\mathbf{x}$
  - Surface force (e.g. pressure): represented by stress tensor σ

$$\mathbf{F}_s = \int_{\partial\Omega_t} \boldsymbol{\sigma}(\mathbf{x},t)\cdot\mathbf{n}ds = \int_{\Omega_t} div(\boldsymbol{\sigma})d\mathbf{x},$$

$$\text{where } \mathbf{n} : \text{surface normal}; \boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix} \text{ for 3D cases.}$$

Derivation of Navier-Stokes Equations

- Transport theorem

For a differentiable scalar field $f : \Omega_t \times [0, t_{end}] \to \Re$,

$$\frac{d}{dt}\int_{\Omega_t} f(\mathbf{x},t)d\mathbf{x} = \int_{\Omega_t}\left(\frac{\partial}{\partial t} f(\mathbf{x},t) + div(f(\mathbf{x},t)\cdot\mathbf{u})\right)d\mathbf{x}$$

- So Newton's second law says (here $f = \rho\,\mathbf{u}$)

$$\frac{\partial}{\partial t}(\rho\mathbf{u}) + \mathbf{u}\cdot(\nabla\rho\mathbf{u}) + \rho\mathbf{u}(\underbrace{\nabla\cdot\mathbf{u}}_{0}) - \rho\mathbf{f} - div(\boldsymbol{\sigma}) = 0$$

$$\therefore \frac{\partial}{\partial t}\mathbf{u} = \underbrace{-\mathbf{u}\cdot\nabla(\mathbf{u})}_{\text{advection}} + \underbrace{\frac{1}{\rho}div(\boldsymbol{\sigma})}_{\text{pressure, diffuse}} + \underbrace{\mathbf{f}}_{\text{external force}}$$

Derivation of Navier-Stokes Equations

- So the equation depends on the stress tensor
- For viscous fluids, σ depends on pressure and internal friction
  - Some applications also include boyancy in σ
  - For more detail, see [Griebel et al. 98]
  - Finally we have

$$\frac{\partial \mathbf{u}}{\partial t} = \underbrace{-(\mathbf{u}\cdot\nabla)\mathbf{u}}_{\text{advection}} - \underbrace{\frac{1}{\rho}\nabla p}_{\text{pressure}} + \underbrace{\nu\nabla^2\mathbf{u}}_{\text{diffuse}} + \underbrace{\mathbf{f}}_{\text{external force}}$$

The velocity of the fluid carries itself along

Pressure in the fluid leads to acceleration

Internal friction results in diffusion in the momentum

Helmholtz-Hodge Decomposition

$$\mathbf{w} = \mathbf{u} + \nabla q, \text{ where } \nabla \cdot \mathbf{u} = 0$$

$$\nabla \cdot \mathbf{w} = \nabla^2 q$$

- Decomposes a vector field **w** into a divergence-free vector field **u** and another gradient field
- Define an operator P such that P(**w**) = **u**
  - Project any vector field to its divergence-free part
  - P(gradient field) = 0

$$\mathbf{u} = \mathrm{P}(\mathbf{w}) = \mathbf{w} - \nabla q \qquad (3)$$

Helmholtz-Hodge Decomposition

- Apply P() to both sides of (2), we get

$$\frac{\partial \mathbf{u}}{\partial t} = \mathrm{P}(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}) \qquad (4)$$

  - Since P(**u**) = **u** and P(del(*p*)) = 0

Outline of Solution

- Start from the solution of previous time step (*t*) and add each term on the right hand side of Eq.4, and them perform the projection to satisfy Eq.1

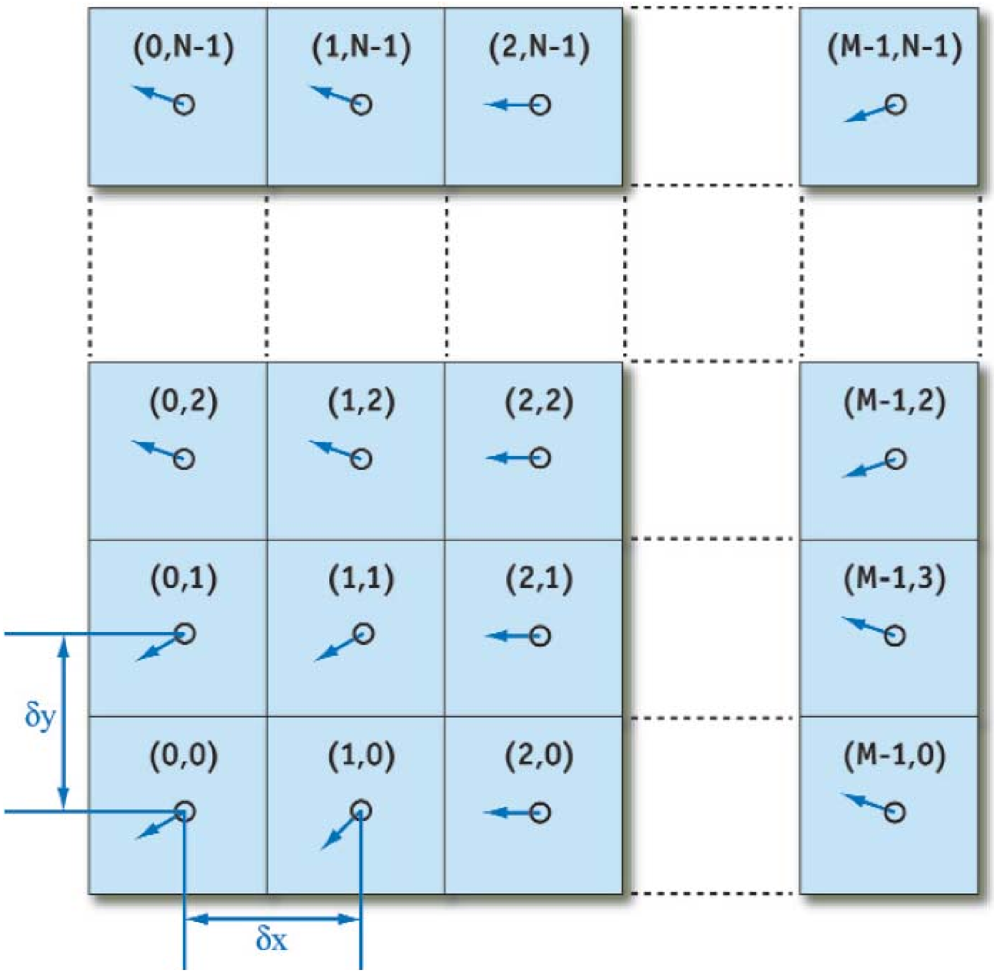
$$\mathbf{w}_0(\mathbf{x}) = \mathbf{u}(\mathbf{x},t) = \begin{bmatrix} u(\mathbf{x},t) \\ v(\mathbf{x},t) \end{bmatrix}, \text{ where } \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\mathbf{w}_0(\mathbf{x}) \xrightarrow{\text{add force}} \mathbf{w}_1(\mathbf{x}) \xrightarrow{\text{advect}} \mathbf{w}_2(\mathbf{x}) \xrightarrow{\text{diffuse}} \mathbf{w}_3(\mathbf{x}) \xrightarrow{\text{project}} \mathbf{w}_4(\mathbf{x})$$

- **w** can be stored in one RGBA texture
  - 2D case: 2D texture using 2 channels
  - 3D case: 3D texture using 3 channels

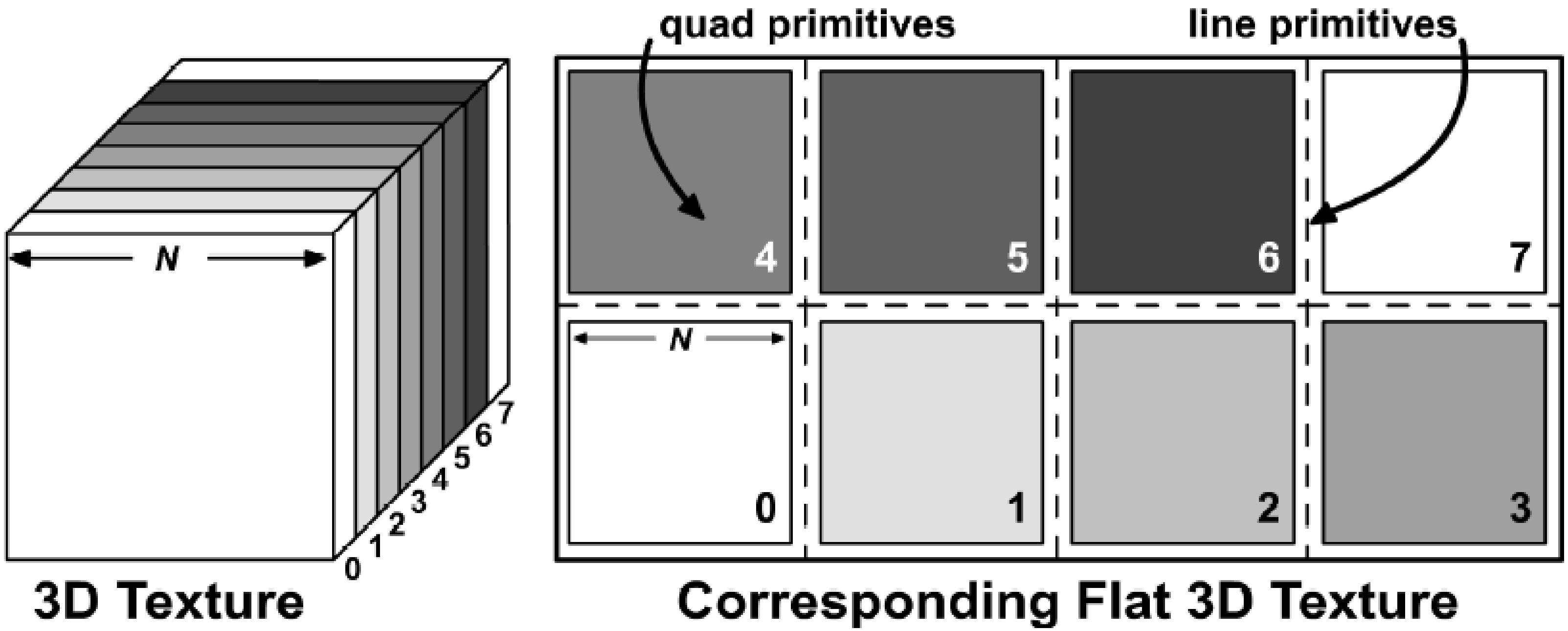
Storage

- 2D example:



3D Textures vs. Flat 3D Textures

- According to [Harris 03], flat 3D textures have performance advantage over true 3D textures on current graphics hardware



External Force

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{f}$$

$$\mathbf{w}_1(\mathbf{x}) = \mathbf{w}_0(\mathbf{x}) + \Delta t\mathbf{f}(\mathbf{x}, t)$$

- An approximation over the time step Δt
- Easy to implement on GPU once we have $\mathbf{w}_0$ and **f** as input texture
  - For each cell (fragment), lookup textures $\mathbf{w}_0$ and **f** and add them.

Advection [Stam 99]

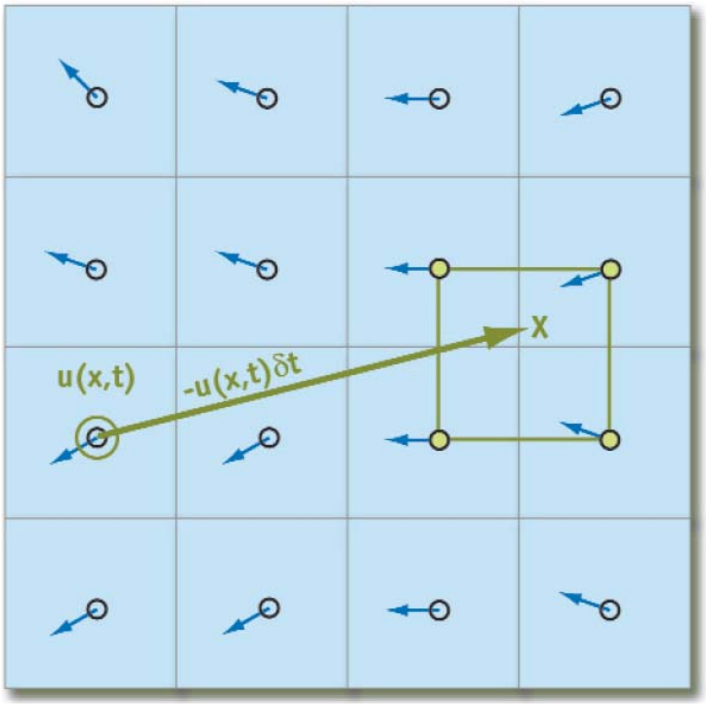
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u}$$

- Solve the PDE by method of characteristics, we can find that the value of **u** does not change along the "streamlines" of the velocity field, therefore

$$\mathbf{w}_2(\mathbf{x}) = \mathbf{w}_1(\mathbf{p}(\mathbf{x}, -\Delta t))$$

$\mathbf{p}(\mathbf{x}, -\Delta t)$: the location of $\mathbf{x}$ a time Δt ago, according to the velocity field

Advection



- When **p**(**x**, Δ*t*) is between the grids, interpolate it
- Can also be easily done on GPU, for each cell,
  - $\mathbf{w}_1$ as input texture
  - Compute **p**(**x**, Δ*t*) in fragment shader
  - Perform 4 texture look-ups on $\mathbf{w}_1$ and interpolate
    - Use built-in function in Cg, **`f4texRECTbilerp()`**

Diffusion [Stam 99]

$$\frac{\partial \mathbf{w}_3}{\partial t} = \nu \nabla^2 \mathbf{w}_3$$

$$\frac{\mathbf{w}_3 - \mathbf{w}_2}{\Delta t} \approx \nu \nabla^2 \mathbf{w}_3$$

$$(\mathbf{I} - \nu \Delta t \nabla^2)\mathbf{w}_3(\mathbf{x}) = \mathbf{w}_2(\mathbf{x})$$

- It involves solving a Poisson equation (details later)

$$-\nabla^2 v = f(x),$$

$$\text{where } \nabla^2 = \nabla \cdot \nabla$$

Projection to Divergence-Free Vectors

- Solve for *q* and subtract it from $\mathbf{w}_3$

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

$$\mathbf{w}_4 = \mathbf{w}_3 - \nabla q$$

- Also a Poisson equation

Poisson Equation as Linear System

- So the key to solving N-S equation is solving the Poisson equations
- For example, one-dimensional version:
  - Discretize the space into N+1 grids

v_a	v_1	…	v_N	v_b
0	1/(N+1)	…	N/(N+1)	1

$$-\frac{\partial^2 v(x)}{\partial x^2} = f(x),\ 0 \le x \le 1,\ v(0) = v_a, v(1) = v_b$$

$$h = \frac{1}{N+1}, \text{let } v_n = v(nh).\ \frac{\partial}{\partial x} v_i \approx \frac{v_i - v_{i-1}}{h}; \frac{\partial}{\partial x} v_{i+1} \approx \frac{v_{i+1} - v_i}{h}.$$

$$\text{Therefore } \frac{\partial^2}{\partial x^2} v_i \approx \frac{-2v_i + v_{i-1} + v_{i+1}}{h^2}, \text{ and } \underbrace{\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}}_{\mathbf{T}_N} \underbrace{\begin{bmatrix} v_1 \\ \\ \vdots \\ \\ v_N \end{bmatrix}}_{\mathbf{v}} - \begin{bmatrix} v_a \\ 0 \\ \vdots \\ 0 \\ v_b \end{bmatrix} \approx h^2 \begin{bmatrix} f_1 \\ \\ \vdots \\ \\ f_N \end{bmatrix}.$$

Poisson Equation Solvers

- It can be extended to 2D or 3D
  - T_N, T_{NxN}, T_{NxNxN} are symmetric banded matrices
  - Direct methods to solve linear systems: $O(N^3)$ time
    - impossible for 2D or 3D cases
- Need iterative methods
  - Please refer to previous lectures on linear algebra and banded matrices [Sashi, Suddha]
  - Conjugate gradient [Krüger and Westermann 03], [Boltz et al. 03]
  - Multigrid [Boltz et al. 03]: O(*N*) time for *N* samples

Poisson Equation as Linear System

- It can be shown that [Demmel 97]

$$\left\|\mathbf{v}-\hat{\mathbf{v}}\right\|_2 \leq O(h^2 \left\| \frac{d^4 v}{dx^4} \right\|_\infty)$$

  - Truncation error approaches zero proportional to h^2
  - But the condition number of $\mathbf{T}_N$ is [Demmel 97]

$$\kappa(T_N) \approx \frac{4(N+1)^2}{\pi^2}$$

    - Larger *N* makes the system more sensitive to FP errors
    - Remember: only 32-bit floating point numbers on GPU
    - N should be large enough, but not too large

Boundary Conditions

- To solve the Poisson equation, we still need boundary values that satisfy boundary conditions
  - No-slip condition: velocity goes to zero at the boundaries
- Resolution of boundary is limited by the size of grids

Boundary Conditions

- The boundary lies on the edge between the boundary cell and its nearest interior cell
  - Assign imaginary velocity value to boundary cells so that the average of itself and its nearest interior cell should satisfy the condition
  - For example, on the left side,

$$\frac{\mathbf{u}_{0,j} + \mathbf{u}_{1,j}}{2} = 0$$

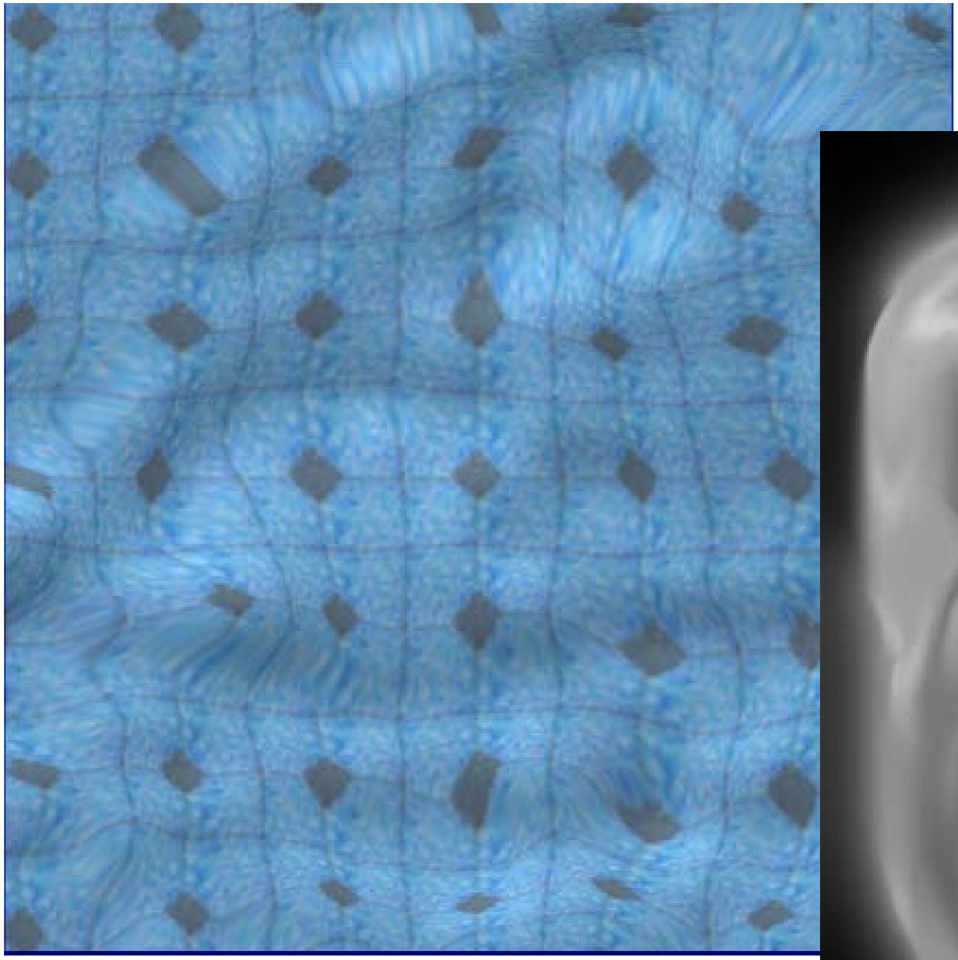
$$\mathbf{u}_{0,j} = -\mathbf{u}_{1,j}$$

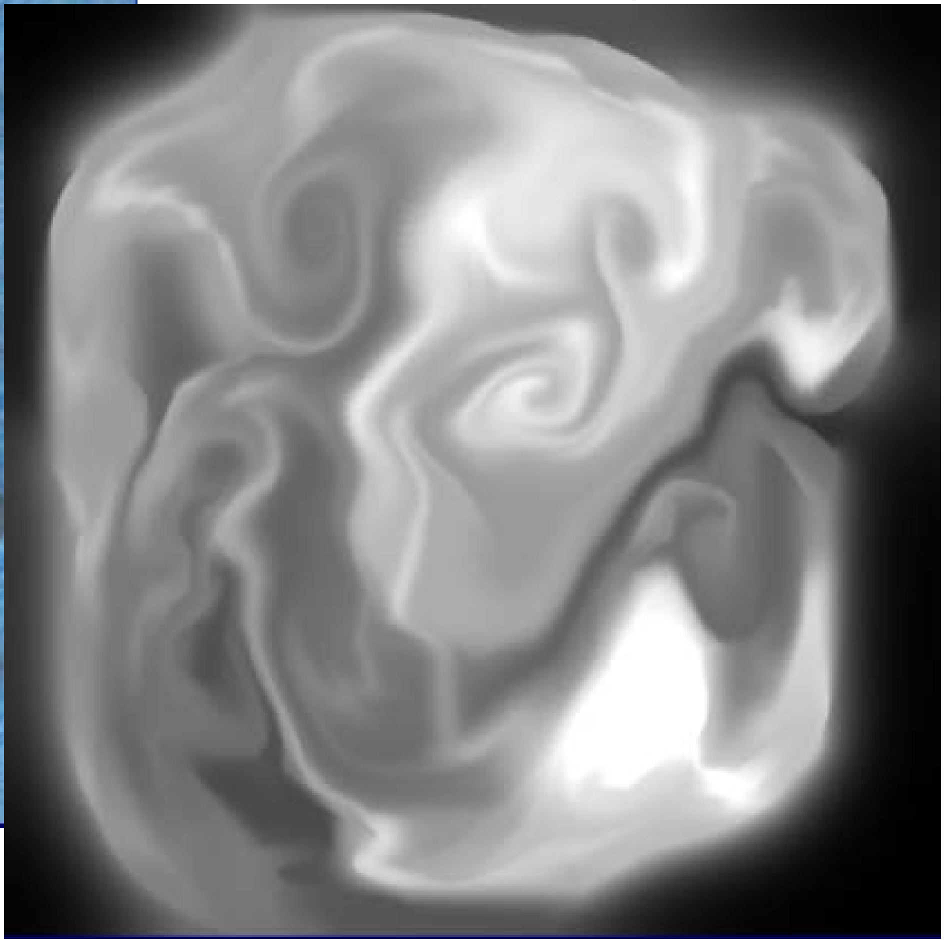
Boundary Conditions

- To update the boundary cells after solved the velocity field:
  - Draw lines on the boundary
  - In the shader: lookup texture **u** at the coordinate of nearest interior cell and return the negative of the value.
- Arbitrary boundaries is complicated
  - For each boundary cell, need to determine the direction of the face
  - More computation in the shader, more lines

Results [Krüger and Westermann 03]

1024x1024, 9 fps



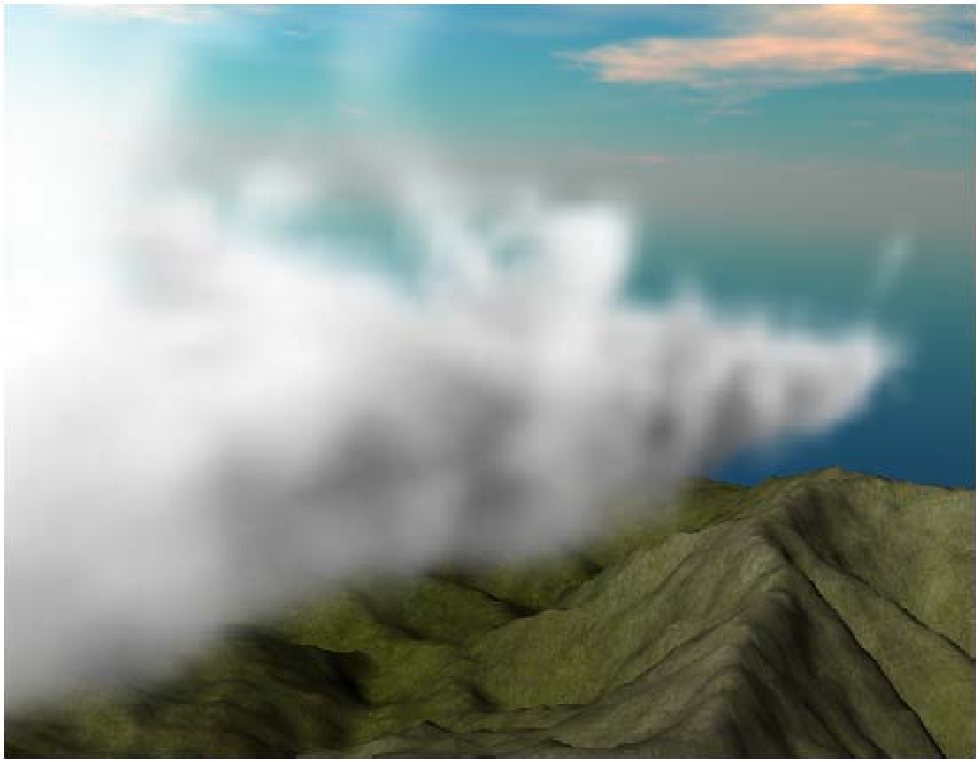


1024x1024, 13 fps

Performance

- The performance should be governed by the Poisson solver since other parts require little effort
- [Krüger and Westermann 03] reported a 2D N-S equation solver has 9 fps on a 1024^2 grid
  - using P4 2.8GHz with ATI 9800 graphics card
  - but did not compare with performance on CPU

Results [Harris et al. 2003]



128x128 grid, 30 fps

Performance [Harris et al. 2003]

- [Harris et al. 2003] reported 3D cloud simulation results on Geforce FX Ultra
  - 32x32x32: 27 iterations per second
  - 64x64x64: 3.6 iterations per second
    - (I'm not sure if they include rendering time)
  - Not compared to CPU

Reference—Navier-Stokes Equations

- Stam, J. Stable Fluids. In *Proceedings of SIGGRAPH 1999*.
- Griebel, M., Dornseifer, T., Neunhoeffer, T. *Numerical Simulation in Fluid Dynamics*. Society for Industrial and Applied Mathematics. 1998.
- Demmel, J. W. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics. 1997.
- Harris, M. Fast Fluid Dynamics Simulation on the GPU. In *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. 2004.
- Krüger, J. and Westermann, R. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *SIGGRAPH 2003.*
- Bolz, Farmer, Grinspun and Schröder, Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *SIGGRAPH 2003*.
- Harris, M., Baxter, W. V., Scheuermann, T., and Lastra, A. Simulation of Cloud Dynamics on Graphics Hardware. *Graphics Hardware 2003*.

Lattice Boltzmann Method for Fluid Simulation on the GPU

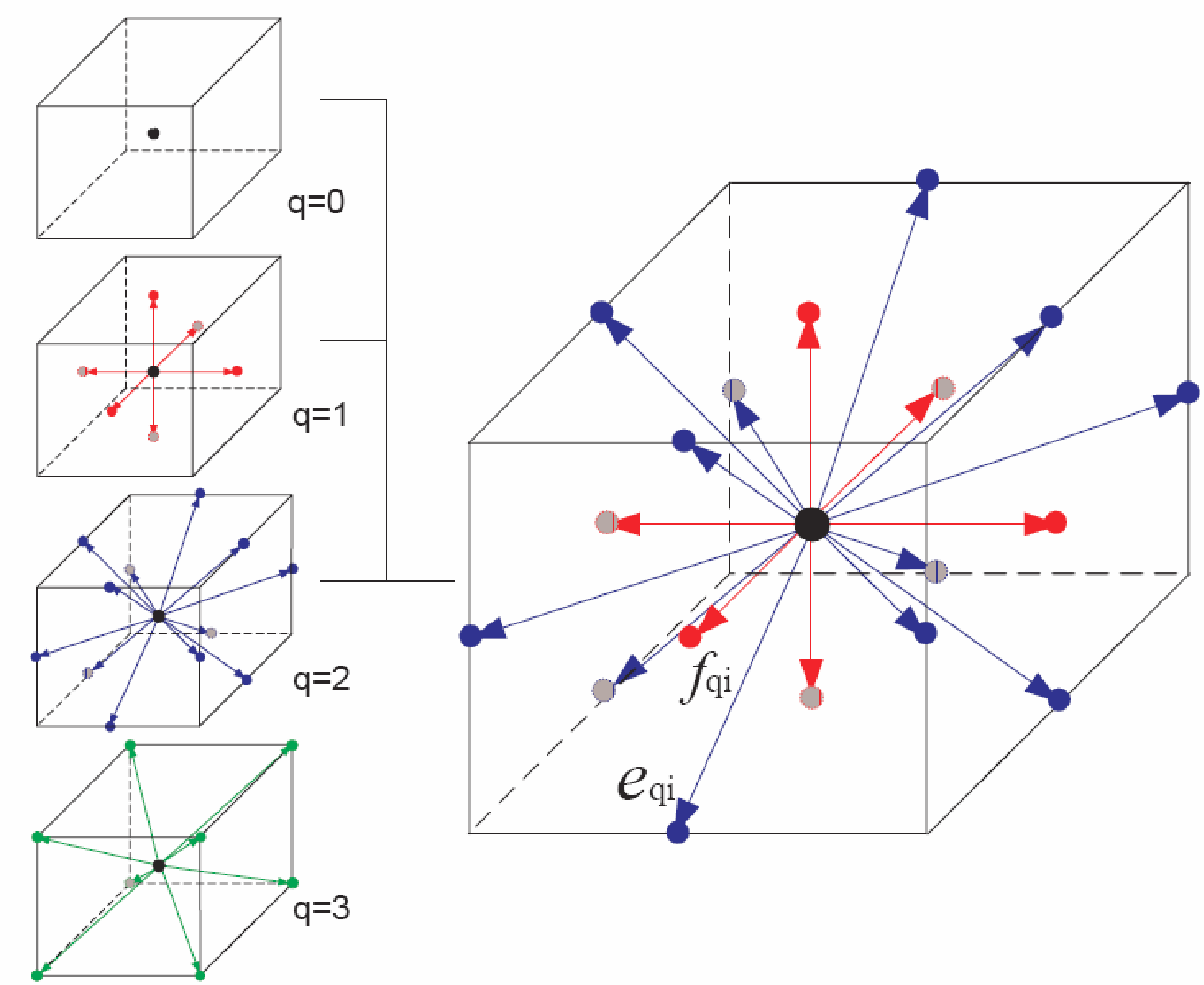
Two Different Strategies

- Top-down: solving differential equations by discretizing the space
  - Be aware of truncation error when using finite difference!
  - Navier-Stokes equations
- Bottom-up: start from a discretized microscopic model that conserves desired quantities
  - Lattice Gas Automata, Lattice Boltzmann Model

Lattice Boltzmann Model

- Simulate microscopic behaviors of particles
  - Streaming: each particle moves to the nearest node in the direction of its velocity
  - Collision: particles arriving at a node interact and change their velocity directions
- Averaged microscopic properties obey the desired macroscopic properties (conservation of mass and momentum)

Lattice Geometry—D3Q19



Lattice Gas Automata

- The space is divided into a lattice of nodes with particles resides on them
- Each node has a set of directions of velocity
  - $\mathbf{e}_i$, $i = 0, 1, \ldots, M$
- Each velocity vector is coupled with a boolean variable
  - $n_i(\mathbf{x}, t)$, $i = 0, 1, \ldots, M$
  - $\mathbf{x}$: location of the node; t: time
  - true iff there is a particle moving in this direction

Lattice Gas Automata

- At each time step, evolve each node with

$$n_i(\mathbf{x}+\mathbf{e}_i,t) = \underbrace{n_i(\mathbf{x},t)}_{\text{streaming}} + \underbrace{\Omega_i(n(\mathbf{x},t))}_{\text{collision}}$$

  - Streaming: each particle moves to the nearest node in the direction of its velocity
  - Collision: particles arriving at a node interact and change their velocity directions
    - No more than one particle is allowed in a node with a given velocity

Lattice Boltzmann Method (LBM)

- Now replace the particle occupation variables n_i with single-particle distribution functions
  - $f_i = \langle n_i \rangle$
  - The density of particles that have a given velocity

Lattice Boltzmann Equations (LBE)

$$f_i(\mathbf{x}+\mathbf{e}_i\Delta x, t+\Delta t) = \underbrace{f_i(\mathbf{x},t)}_{\text{streaming}} + \underbrace{\Omega_i(f(\mathbf{x},t))}_{\text{collision}}, \qquad i = 0, 1, \ldots, M$$

f_i : particle velocity distribution function along the ith direction

$\Omega_i = \Omega_i(f(\mathbf{x},t))$: collision operator which represents the rate of change of f_i resulting from collision

Δt and Δx : time and space increments

- Discretized space is consistent with the equation
  - The nearest neighbors of $\mathbf{x}$ are $\mathbf{x} + \mathbf{e}_i$, $i = 0, 1, \ldots, M$

Lattice Boltzmann Equations (LBE)

- The density and momentum density of a node are

$$\rho = \sum_{i=1}^{M} f_i, \qquad \rho\mathbf{u} = \sum_{i=1}^{M} f_i \mathbf{e}_i$$

  - So we can compute velocity field **u**

- Ω_i is required to satisfy conservation of total mass and total momentum at each node

$$\sum_{i=1}^{M} \Omega_i = 0, \qquad \sum_{i=1}^{M} \Omega_i \mathbf{e}_i = 0$$

Two-Step Update of LBE

$$\text{collision}: f_i^{new}(\mathbf{x},t) = f_i(\mathbf{x},t) + \Omega_i$$

$$\text{streaming}: f_i(\mathbf{x}+\mathbf{e}_i, t+1) = f_i^{new}(\mathbf{x},t)$$

- How to compute the collision term?

Collision

- The distribution function f_i can be expanded about the local equilibrium distribution function f_i^{eq}, which satisfies

$$f_i = \underbrace{f_i^{eq}}_{\text{equilibrium}} + \underbrace{f_i^{neg}}_{\text{nonequilibrium}}$$

$$\rho = \sum_{i=1}^{M} f_i^{eq}, \qquad \rho \mathbf{u} = \sum_{i=1}^{M} f_i^{eq} \mathbf{e}_i$$

$$0 = \sum_{i=1}^{M} f_i^{neq}, \qquad 0 = \sum_{i=1}^{M} f_i^{neq} \mathbf{e}_i$$

  - f_i^{eq} only depend on ρ and $\mathbf{u}$
  - Equilibrium means that forces in all directions are balanced

Collision

- The nonequilibrium ("unbalanced") part is resulted from collision

$$\Omega_i = -\frac{1}{\tau}\left(f_i(\mathbf{x},t) - f_i^{eq}(\rho,\mathbf{u})\right)$$

$f_i^{eq}(\rho,\mathbf{u})$: local equilibrium distribution function

τ : constant that determines the viscosity

- How to find f_i^{eq}?

Equilibrium Distribution Function

- Bhatnager, Gross, Krook (BGK) model [Wolf-Gladrow 2000]

$$f_i^{eq}(\rho, \mathbf{u}) = \rho(A + B(\mathbf{e}_i \cdot \mathbf{u}) + C(\mathbf{e}_i \cdot \mathbf{u})^2 + D(\mathbf{u} \cdot \mathbf{u}))$$

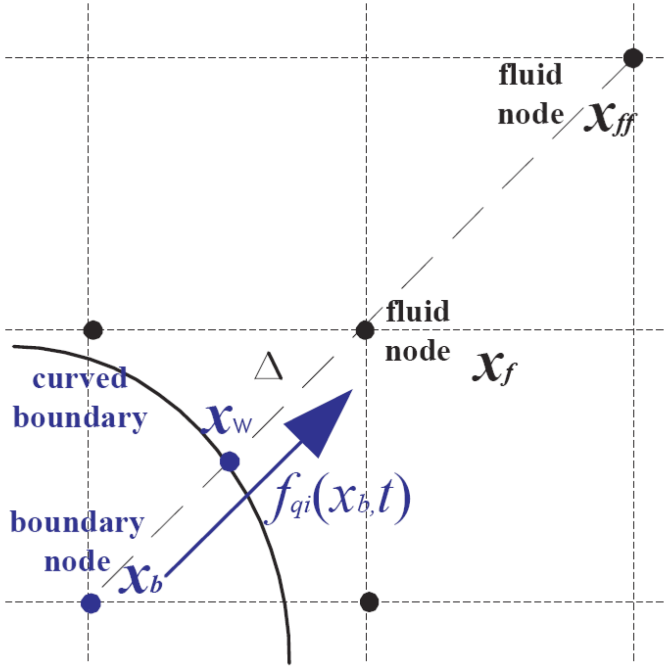
A, B, C, D : constant coefficients specific to the chosen lattice geometry

Boundary Condition

- For simple boundary (box aligned with axes), the “bounce-back” method we mentioned before is enough
- For arbitrary boundary, LBM becomes easier than N-S based methods since the vectors are fixed to a certain directions
  - *f* for boundary nodes can be interpolated

Arbitrary Boundary [Mei et al. 2000]

- Boundary nodes are given a imaginary *f* value so that the interpolated value at the boundary satisfies the no-slip condition



$$\Delta = \frac{|\mathbf{x}_f - \mathbf{x}_w|}{|\mathbf{x}_f - \mathbf{x}_b|}$$

The packet distribution at $\mathbf{x}_f$ is streamed from $\mathbf{x}_b$, so we need to define an imaginary distribution for $\mathbf{x}_b$

Arbitrary Boundary [Mei et al. 2000]

- Post-collision value of $f_i(\mathbf{x}_b, t)$ is

$$f_{qi}(\mathbf{x}_b, t) = (1-\chi) f_{qi}(\mathbf{x}_f, t) + \chi f^*_{qi}(\mathbf{x}_b) + 6 A_q \rho \mathbf{e}_{qi} \cdot \mathbf{u}_w$$

Velocity of the wall

where,

$$f^*_{qi}(\mathbf{x}_b) = \rho(A_q + B_q \mathbf{e}_{qi} \cdot \mathbf{u}_{bf} + C_q (\mathbf{e}_{qi} \cdot \mathbf{u}_f)^2 - D_q (\mathbf{u}_f)^2)$$

$\mathbf{u}_{bf}$ represents the virtual speed of the boundary node $\mathbf{x}_b$
and for $\Delta \geq 1/2$,

$$\mathbf{u}_{bf} = (1 - \frac{3}{2\Delta})\mathbf{u}_f + \frac{3}{2\Delta}\mathbf{u}_w \quad \text{and} \quad \chi = \frac{2\Delta - 1}{\tau + 1/2}$$

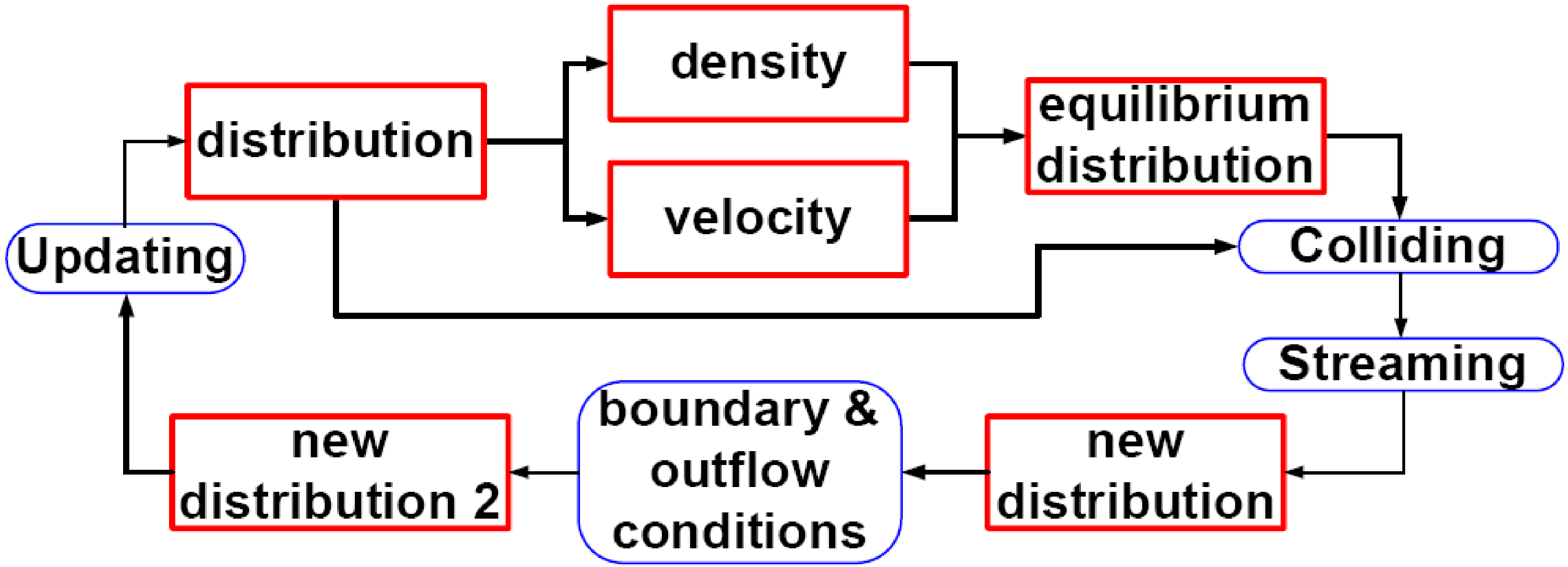
Constant determining viscosity

while for $\Delta < 1/2$,

$$\mathbf{u}_{bf} = \mathbf{u}_{ff} \quad \text{and} \quad \chi = \frac{2\Delta - 1}{\tau - 2}.$$

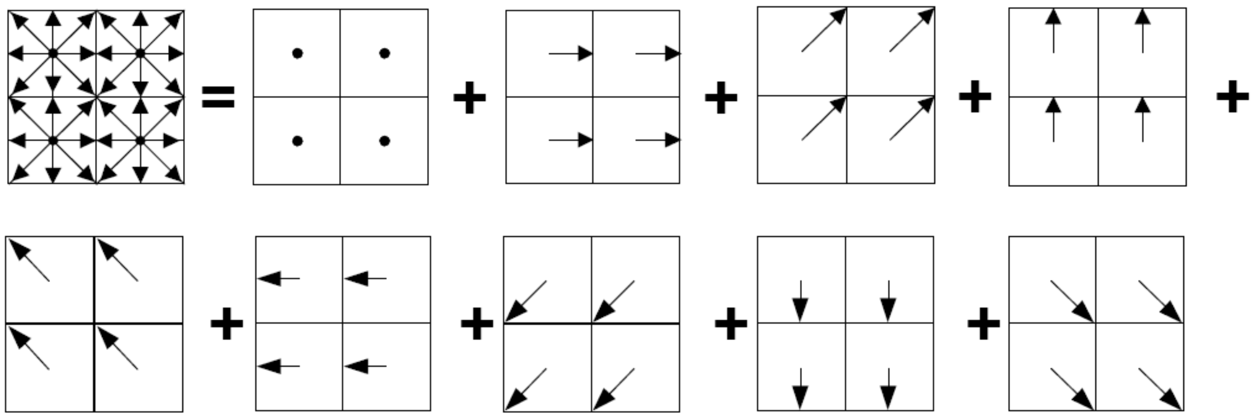
GPU Implementation [Li et al. 2003]

- Flow chart



GPU Implementation—Storage

- Group the distribution functions into arrays according to their velocity vectors



  - Also density, velocity, and equilibrium distribution

GPU Implementation—Storage

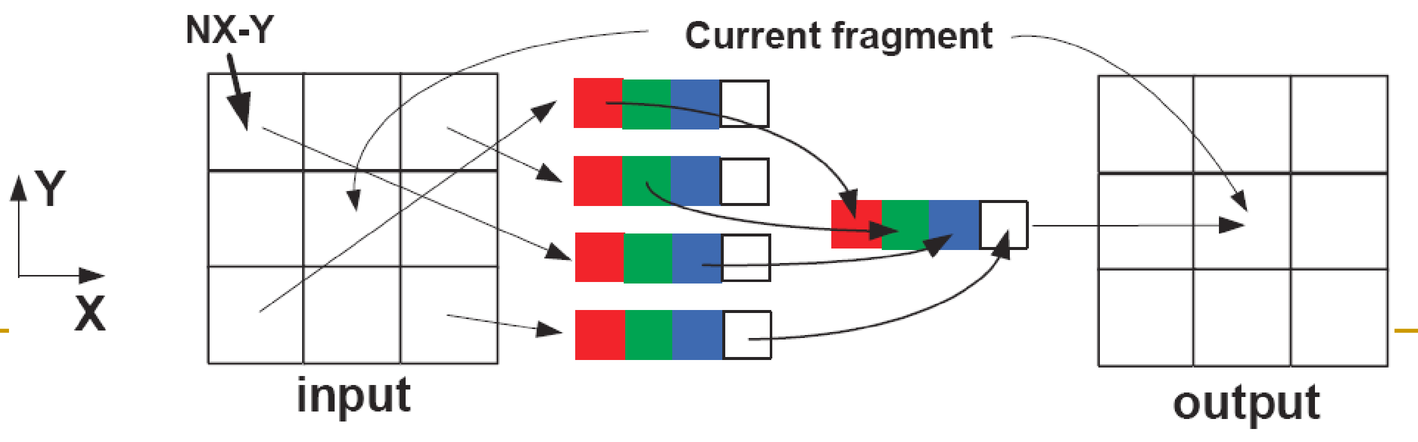
- To exploit 4 channels, pack four arrays into one texture
- For 3D case, the volume is treated as slices of 2D textures
  - Flat volume, [Harris et al. 2003]

Table 1: Packed LBM variables of the D3Q19 model

Texture	R	G	B	A
$\mathbf{u}\rho$	v_x	v_y	v_z	ρ
$f0$	$f_{(1,0,0)}$	$f_{(-1,0,0)}$	$f_{(0,1,0)}$	$f_{(0,-1,0)}$
$f1$	$f_{(1,1,0)}$	$f_{(-1,-1,0)}$	$f_{(1,-1,0)}$	$f_{(-1,1,0)}$
$f2$	$f_{(1,0,1)}$	$f_{(-1,0,-1)}$	$f_{(1,0,-1)}$	$f_{(-1,0,1)}$
$f3$	$f_{(0,1,1)}$	$f_{(0,-1,-1)}$	$f_{(0,1,-1)}$	$f_{(0,-1,1)}$
$f4$	$f_{(0,0,1)}$	$f_{(0,0,-1)}$	$f_{(0,0,0)}$	unused

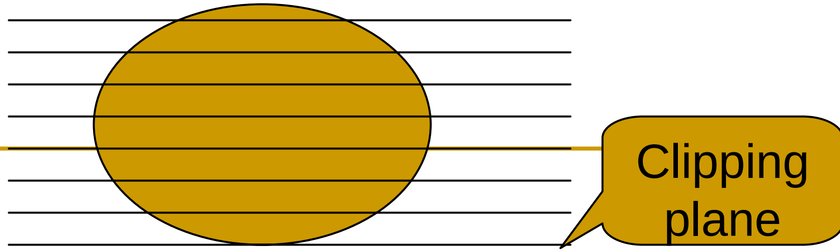
GPU Implementation—Collision & Streaming

- Collision term is computed from texture u ρ and added to textures f0-f4
- Streaming: fetch neighboring texels and copy the corresponding f
  - $f_i^{new}(\mathbf{x}) = f_i(\mathbf{x}-\mathbf{e}_i)$
  - For example, $f^{new}_{(1,-1,0)}(\mathbf{x}) = f_{(1,-1,0)}(\mathbf{x}-(1,-1,0))$



GPU Implementation—Boundary

- To handle the complex boundary, we need to compute the intersections of boundary surface with all the lattice links
  - For moving or deformable boundary, the intersection changes dynamically
- Create voxelization for boundaries by rendering the scene several times with different near and far clipping planes
  - Boundary is sparse in the entire scene, thus does not need too many passes



GPU Implementation—Boundary

- When rendering the boundary voxels, apply the fragment shader to compute boundary conditions
  - We still need $\Delta = \dfrac{|\mathbf{x}_f - \mathbf{x}_w|}{|\mathbf{x}_f - \mathbf{x}_b|}$
  - Each boundary distribution will have the velocity vector crossing the boundary surface

Boundary

- Suppose the boundary surface is defined by Ax + By + Cz + D = 0 [(A, B, C) is normalized]
  - Define

$$flag_1 = -(pos_x, pos_y, pos_z, 1) \cdot (A, B, C, D) \quad \text{(distance to the boundary surface)}$$

$$flag_2 = (A, B, C) \cdot \mathbf{e}_i \quad \text{(angle between normal and } \mathbf{e}_i\text{)}$$

(pos_x, pos_y, pos_z) is the 3D coordinate of the voxel

  - Each $\mathbf{e}_i$ has its own flags
- Need to make sure that each boundary node is covered by a fragment, so that boundary condition is computed for all boundary nodes

Boundary

- Three passes to cover boundary cells
  - First pass—just render the voxels
  - Second pass—only R and G channels are updated, $\mathbf{e}_i$ is the vector corresponding to R channel, translate all voxels and render, with translation offsets decided by the rule:

$$\begin{cases} \mathbf{e}_i : \text{if } \; flag_1 * flag_2 > 0 \\ -\mathbf{e}_i : \text{if } \; flag_1 * flag_2 < 0 \\ 0 : \text{if } \; flag_1 * flag_2 = 0 \end{cases}$$

Boundary

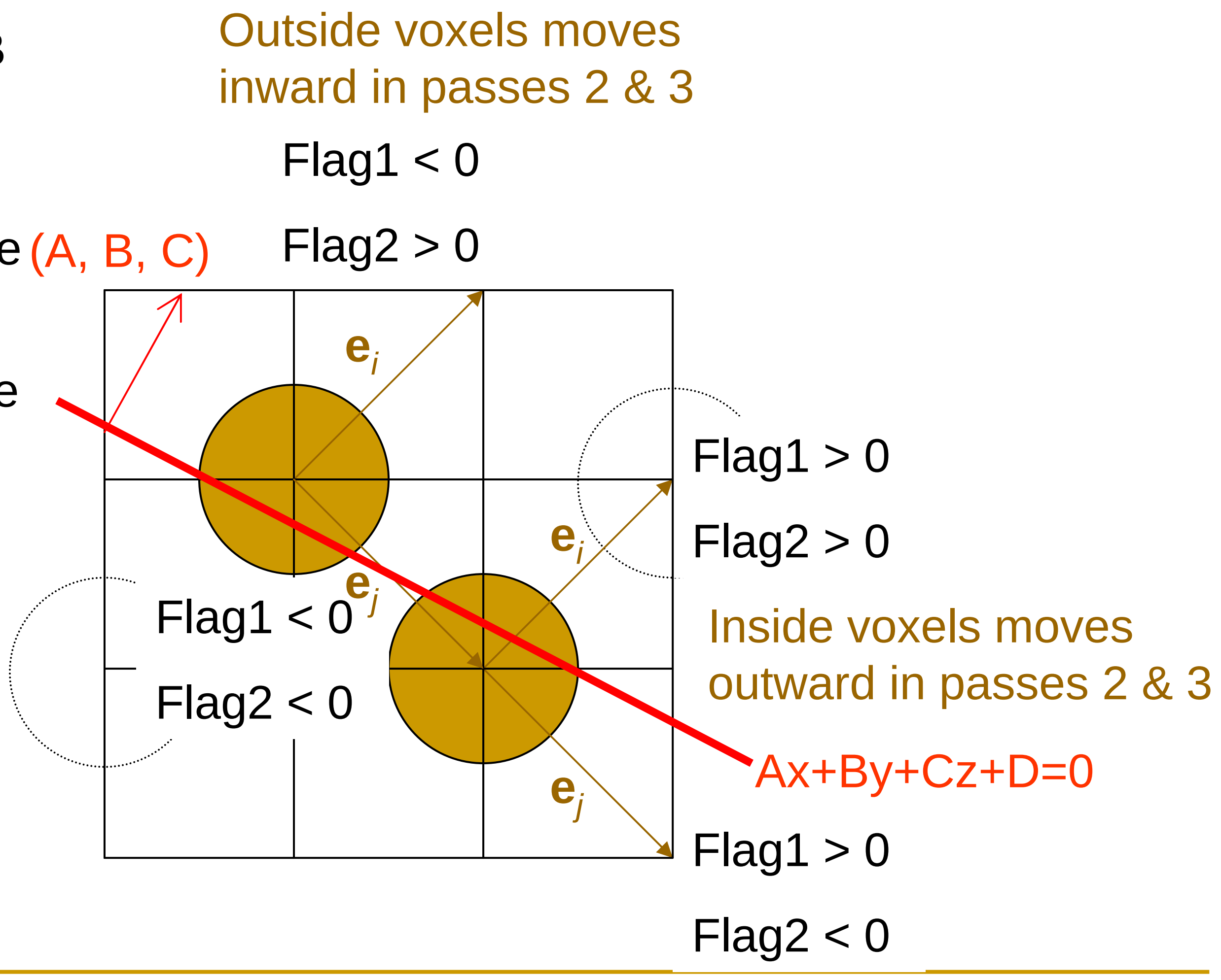
- Third pass—similar to second pass, but only B and A channels are updated, and $\mathbf{e}_i$ is the vector corresponding to the blue channel
- All boundary nodes will be covered by the voxels
  - Note that each pass will check for all four textures f0~f4
- During the passes, compute in the fragment shader

$$\Delta = 1 - \frac{flag_1}{flag_2}$$

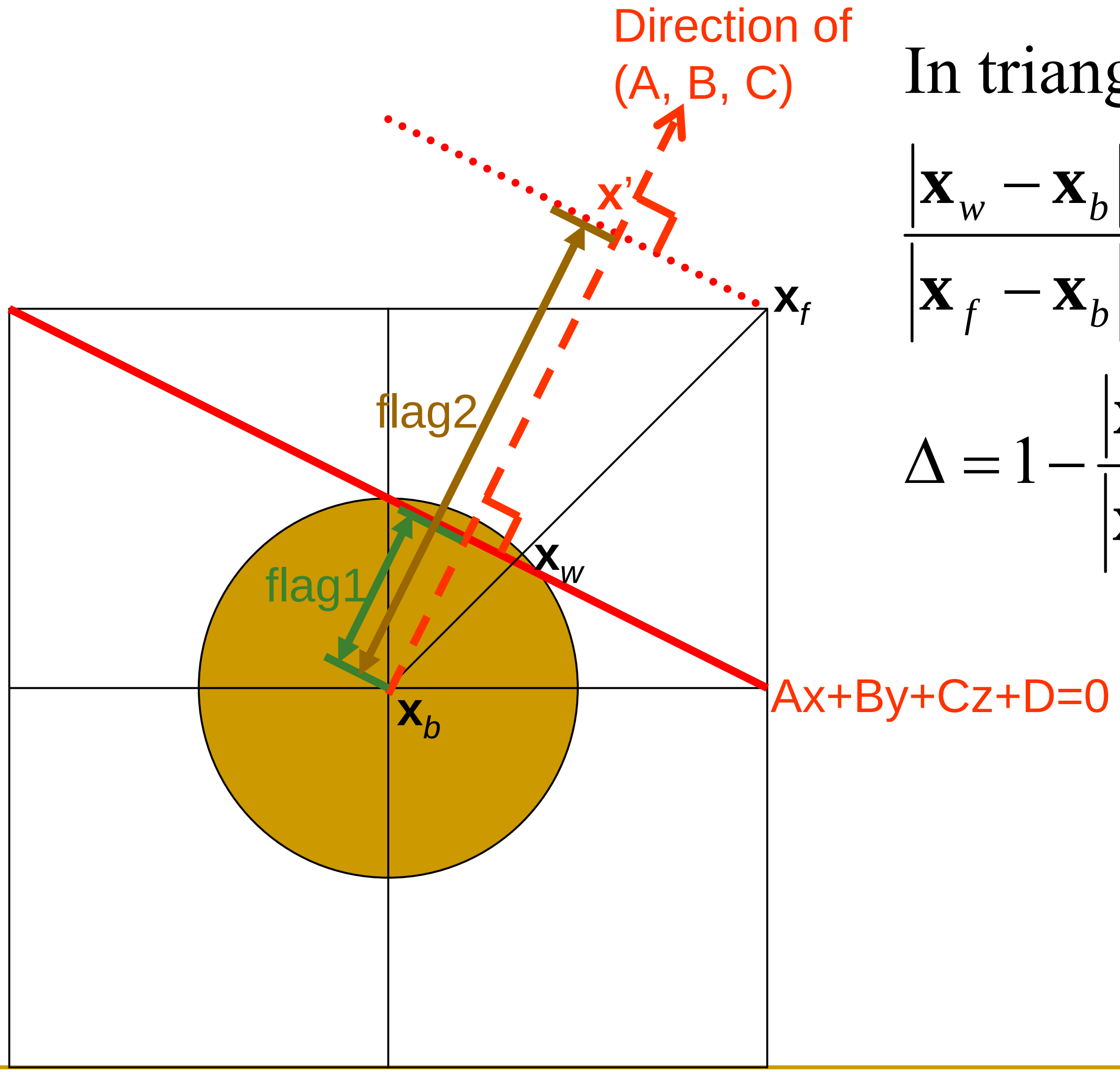
  - If $1 >= \Delta >= 0$, the voxel is a boundary node, and the boundary condition is computed for the voxel

Boundary

- The vectors in R and B channels are perpendicular
- Vectors in R is opposite to vectors in G;
- Vectors in B is opposite to vectors in A



Boundary

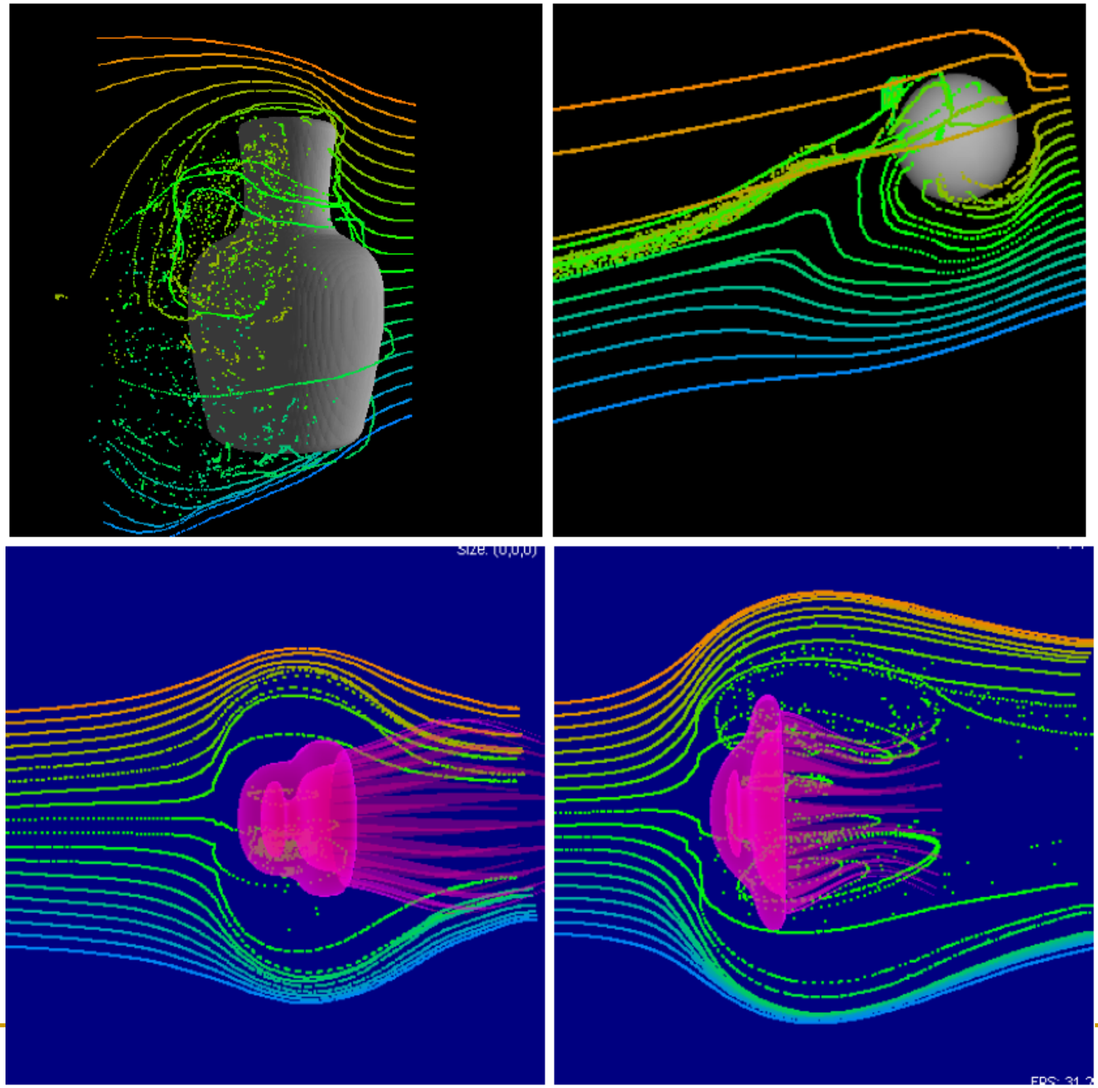


In triangle $\mathbf{x}_b\mathbf{x}_w\mathbf{x}'$,

$$\frac{|\mathbf{x}_w - \mathbf{x}_b|}{|\mathbf{x}_f - \mathbf{x}_b|} = \frac{flag_1}{flag_2}$$

$$\Delta = 1 - \frac{|\mathbf{x}_w - \mathbf{x}_b|}{|\mathbf{x}_f - \mathbf{x}_b|} = 1 - \frac{flag_1}{flag_2}$$

Results [Li et al. 2003]



Performance—2D [Li et al. 2003]

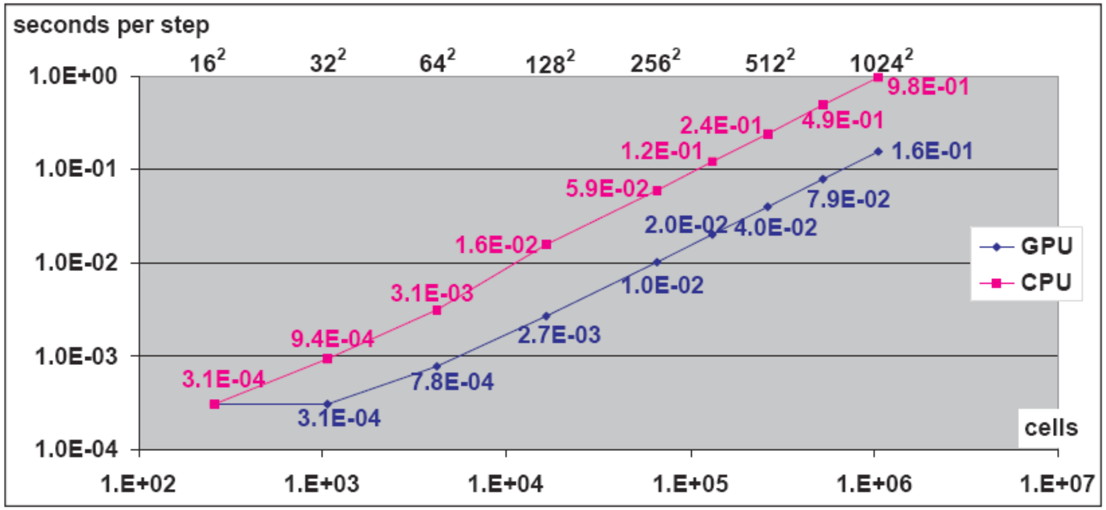
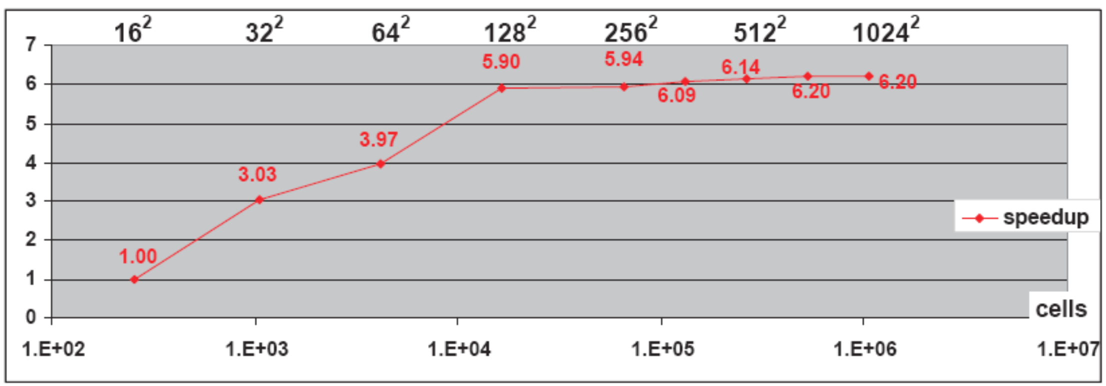


Figure 9: Time (milliseconds) per step of a D2Q9 LBM simulation.



- •Hardware used: P4 2.53 GHz, 1GB PC800 RDRAM with GeForce FX 5900 Ultra (256MB DDR RAM)
- •0.16 seconds per frame on 1024x1024 cell, including simulation and visualization
- •[Kruger and Westermann 03] claimed 0.11 seconds per frame, but they did not deal with complex boundary

Performance—3D [Li et al. 2003]

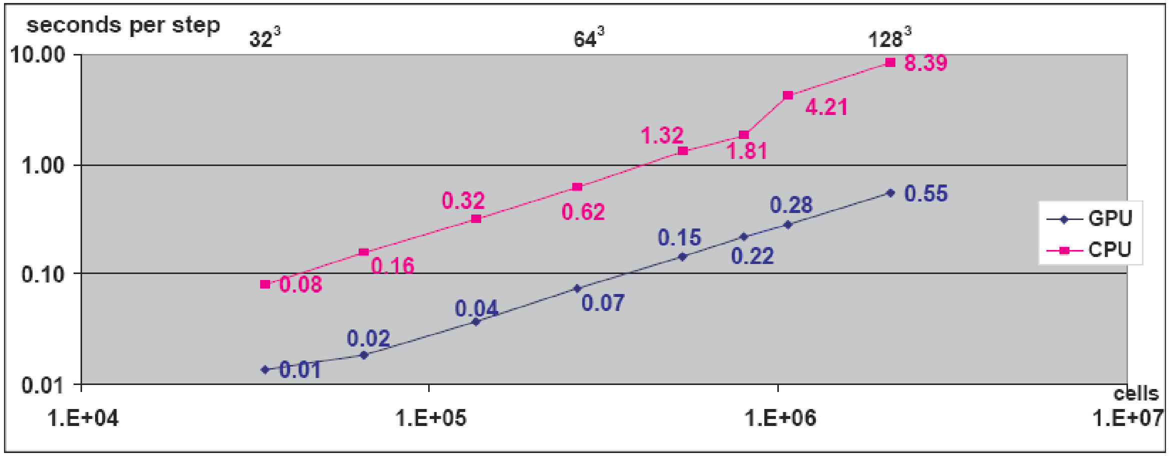
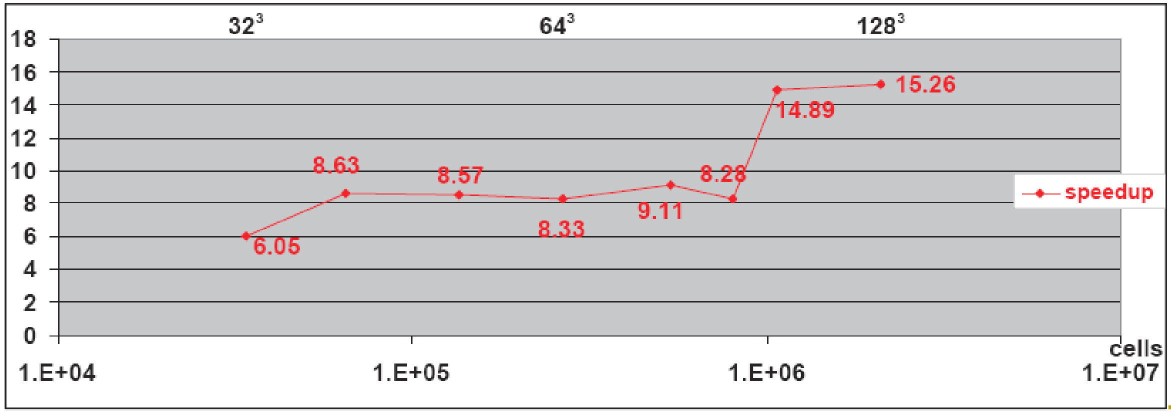


Figure 11: Time (seconds) per step of a D3Q19 LBM simulation.



- [Harris et al. 2003] reported 0.28 sec/iteration on 64x64x64 grids

Reference—Lattice Boltzmann Method

- Li, W., Fan, Z., Wei, X., and Kaufman, A. GPU-Based Flow Simulation with Complex Boundaries. *Technical Report 031105, Computer Science Department, SUNY at Stony Brook.* Nov 2003.
- Chen, S. and Doolean, G. D. Lattice Boltzmann Method for Fluid Flows. *Annu. Rev. Fluid Mech.* 30, 329-364. 1998.
- Wolf-Gladrow, D. A. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models: An Introduction*. Springer-Verlag. 2000.
- Mei, R., Shyy, W., Yu, D., and Luo, L.-S. Lattice Boltzmann Method for 3-D Flows with Curved Boundary. *Journal of Comp. Phys.* 161, 680-699. 2000.

Summary and Comparison

- Navier-Stokes and LBM can be used to simulate fluids, and they are both parallelizable
  - Solving Poisson equations can be a bottle neck for N-S based methods (need more passes for iterative refinement)
  - N-S based method relies on numerical accuracy more than bottom-up methods
    - Sensitivity of linear systems can be critical
    - No double precision on current GPUs may become a major problem for large scale simulation

Summary and Comparison

- Current work using N-S on GPUs only deal with simple boundary, while LBM on GPUs can deal with complex boundary
  - LBM is easier for this because each node only have a set of vector directions
- LBM has advantage of complex boundary and numerical sensitivity

Future Work

- Simulation and visualization of liquid surface are still not solved on GPUs
  - Can we solve for isocontour of liquid grids on the GPU??