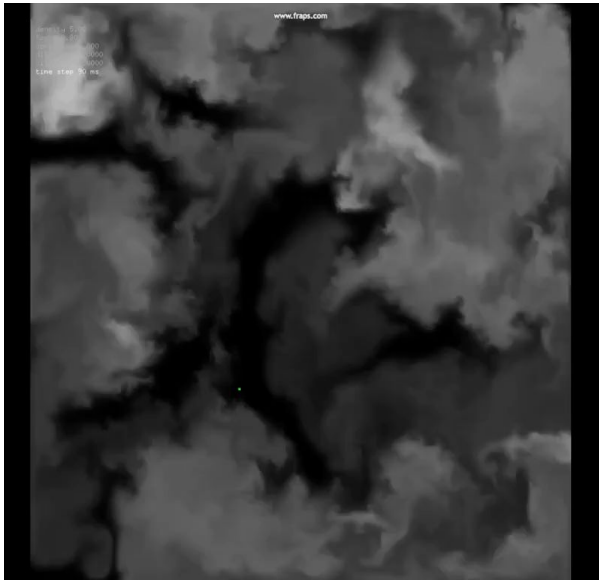
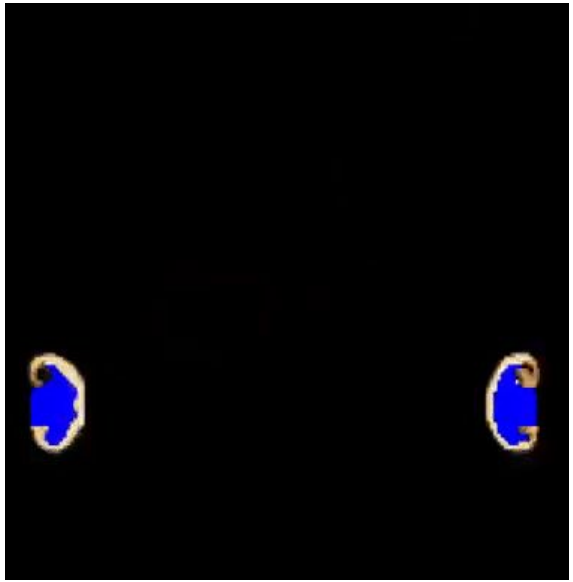
# Smoke and Fire Simulation

Presented by your TAs Josh Wolper and Ziyin Qu



# Background: Film Applications

# Background: Real time Applications

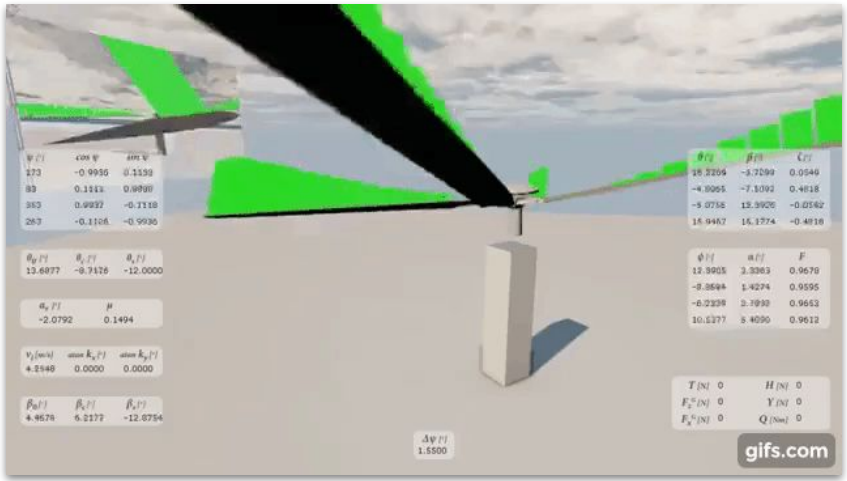
- Engines like PhysX come packed with FX SDKs like FlameWorks (uses real time Eulerian fluid simulation)
- Real time simulations are becoming more common but many video games still optimize by using 2D sprites and application specific tricks

*The Legend of Zelda: Breath of the Wild (2017)*

*Dark Souls 3 (2016)*

# Background: Scientific Simulation

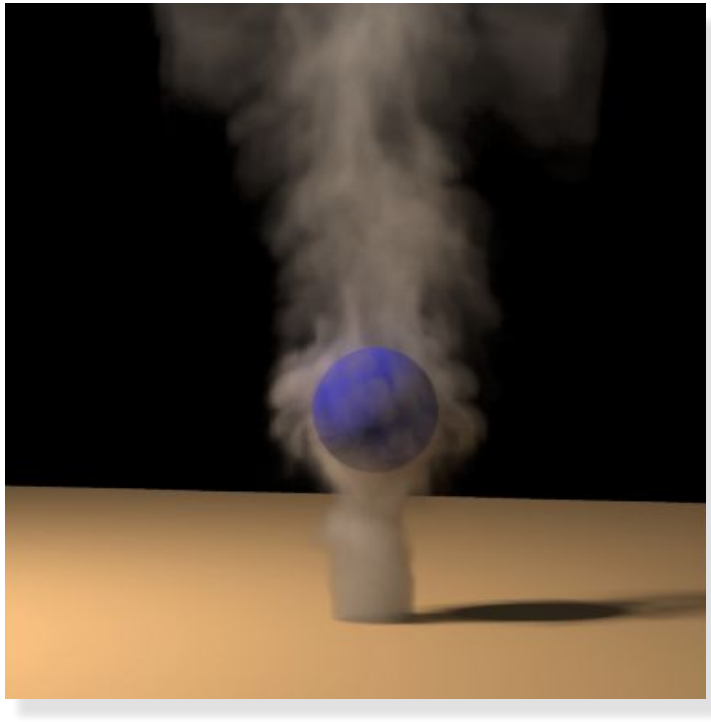
- Variety of real world applications for these simulations
  - Helicopter rotor modeling (general fluids)
  - “EXIT” sign placements (smoke)
  - Firefighter training simulation (smoke and fire)
  - Structure response (explosions)



# Smoke

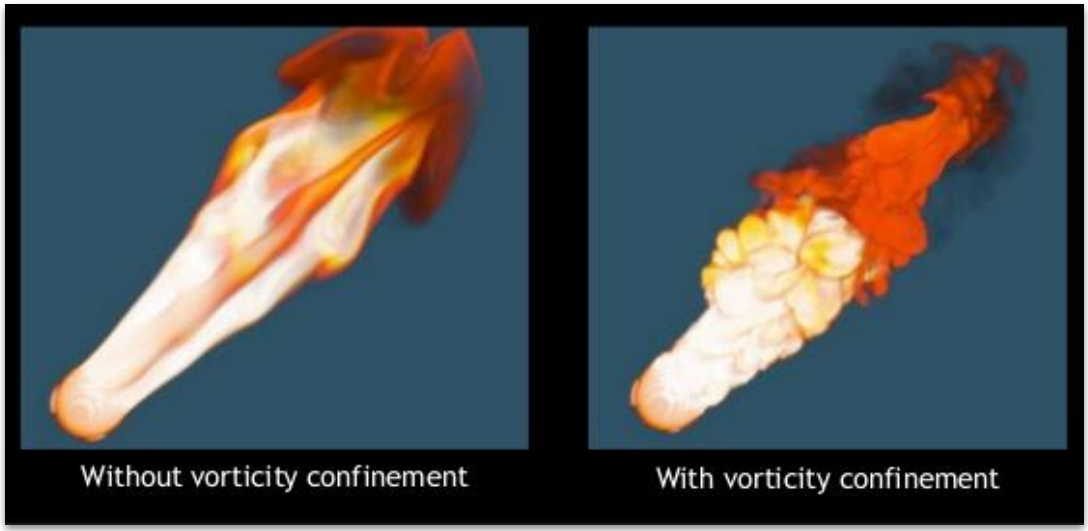
# Smoke Overview

- Challenges
- Physically Based Smoke Model
  - Fedkiw et al. 2001
- Implementation
- Rendering Smoke
- Smoke Demos



# Smoke Challenges

- Solution Stability
- Time Step Size
- Grid Granularity
- Numerical Dissipation
- Adding “knobs” for artists to control




*Numerical dissipation leads to loss of turbulent effects*

# Physically Based Smoke Model

- Want to model inviscid fluid Euler equations
  - Smoke velocity should conserve mass
  - Smoke velocity should conserve momentum
- Must keep track of temperature and density
  - Important for modeling buoyancy and for rendering
  - Temp. and density are advected along smoke velocity

$$\nabla \cdot \mathbf{u} = 0$$

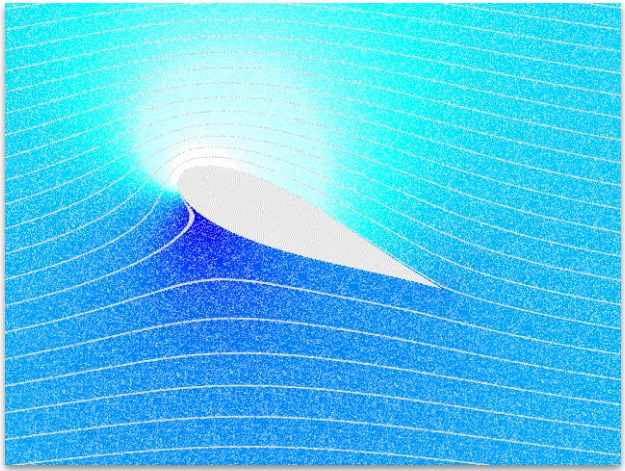
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p + \mathbf{f}.$$

**u** = smoke velocity
p = pressure of gas
**f** = external forces

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

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

**u** = smoke vel.
T = temp.
ρ = density



*Flow around a plane wing*

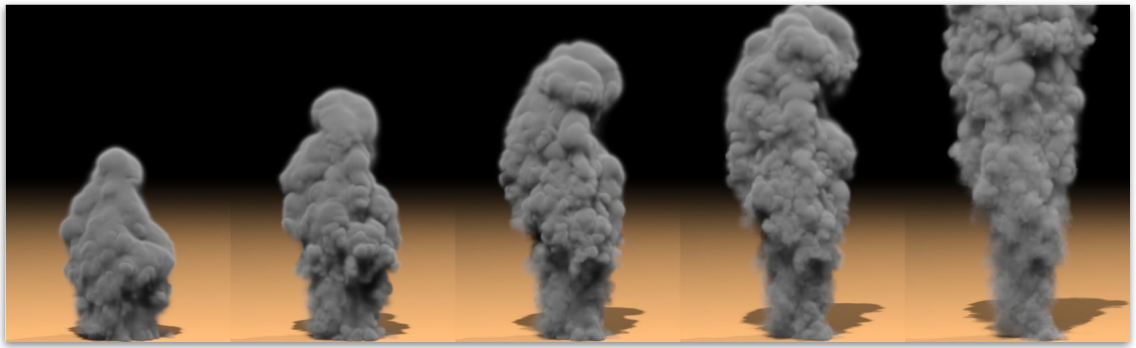
# Physically Based Model cont.

- Want to model a buoyant force
  - Incorporates both temperature and density
  - Tunable parameters give artists freedom

$$\mathbf{f}_{\text{buoy}} = -\alpha\rho\mathbf{z} + \beta(T - T_{\text{amb}})\mathbf{z}$$

| | |
|---|---|
| $\mathbf{f}_{\text{buoy}}$ = buoyant force<br>$T_{\text{amb}}$ = ambient temp<br>$\mathbf{z}$ = (0,0,1) | $\alpha$ = positive constant<br>$\beta$ = positive constant |

- This method has been expanded to a particle based Lagrangian method
  - Main difference: advect over particles



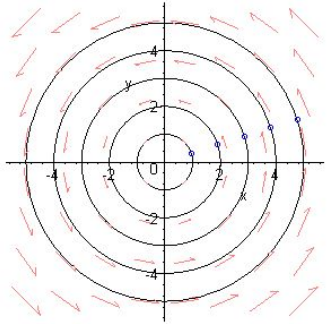
# Vorticity Confinement

- Goal: preserve turbulence within smoke (prevent numerical dissipation)
- Idea:
  - Determine where the system is losing energy (through numerical dissipation)

$$\boldsymbol{\omega} = \nabla \times \mathbf{u} \quad \longrightarrow \quad \mathbf{N} = \frac{\boldsymbol{\eta}}{|\boldsymbol{\eta}|} \qquad (\boldsymbol{\eta} = \nabla |\boldsymbol{\omega}|)$$

  - Inject that energy back into the system

$$\mathbf{f}_{\text{conf}} = \epsilon \, h \, (\mathbf{N} \times \boldsymbol{\omega})$$




*Simple vortex*

- Limitation: can only inject lost energy the system “knows” about
  - Too low grid granularity → fine vortices are lost, cannot be recovered

# Implementation: Semi-Lagrangian method

- **Eulerian Method:**
  - conditionally stable(timestep restriction)
  - stability criteria: Courant-Friedrichs-Lewy(CFL) condition

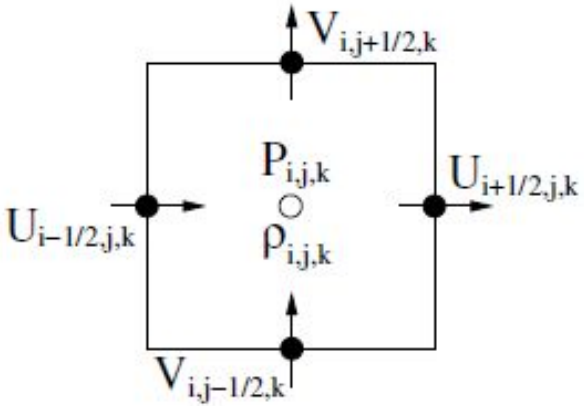
$$C = \frac{u\,\Delta t}{\Delta x} \leq C_{\max}$$

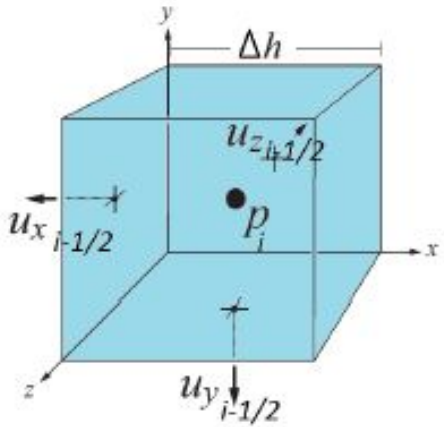
u is the magnitude of velocity
Δt is the timestep, Δx is gridsize, Cmax = 1 for explicit scheme

- **Semi-Lagrangian Method**
  - unconditionally stable(allow really large timestep)
  - still use grid
  - advect velocity, temperature, density in Semi-Lagrangian way
  - resource might be helpful(Chapter 2)

# Implementation: Semi-Lagrangian method

- The computational grid(MAC Grid)
  - Center of cell face = velocity(u, v, w in 3D)
  - Center of cell = pressure(p), density(ρ), temperature(T)
  - Grid is staggered(to avoid checkerboard pattern)



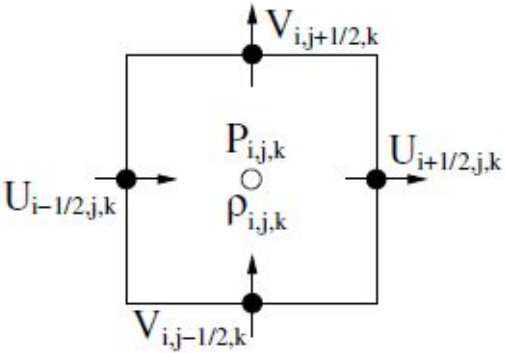



# Implementation: Semi-Lagrangian method

**Notations:**

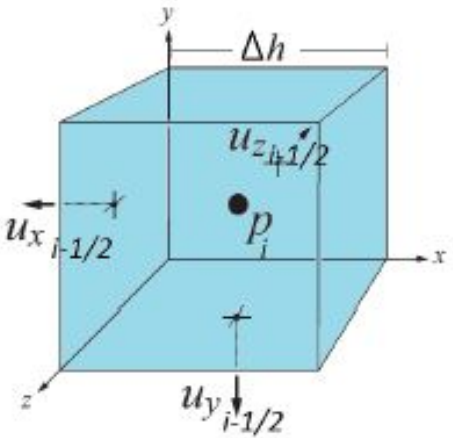
- Velocity:

$$
\begin{aligned}
&u_{i+1/2,j,k}, \quad i = 0, \cdots, N, \quad j,k = 1, \cdots, N,\\
&v_{i,j+1/2,k}, \quad j = 0, \cdots, N, \quad i,k = 1, \cdots, N,\\
&w_{i,j,k+1/2}, \quad k = 0, \cdots, N, \quad i,j = 1, \cdots, N.
\end{aligned}
$$




- Divergence:

$$
\begin{aligned}
(\nabla \cdot \mathbf{u})_{i,j,k} \quad = \quad & (u_{i+1/2,j,k} - u_{i-1/2,j,k} + \\
& v_{i,j+1/2,k} - v_{i,j-1/2,k} + \\
& w_{i,j,k+1/2} - w_{i,j,k-1/2})/h
\end{aligned}
$$

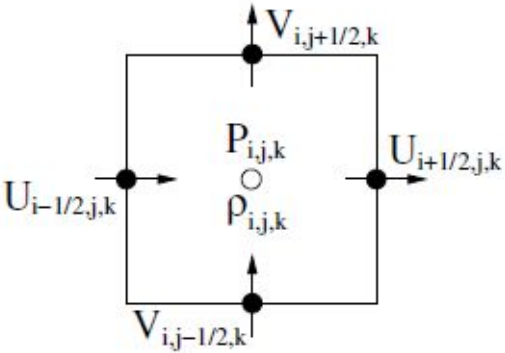


# Implementation: Semi-Lagrangian method

**Notations:**

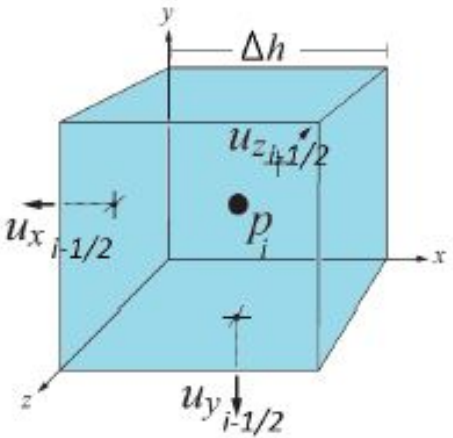
- Velocity:

$$
\begin{aligned}
&u_{i+1/2,j,k}, \quad i = 0, \cdots, N, \quad j,k = 1, \cdots, N,\\
&v_{i,j+1/2,k}, \quad j = 0, \cdots, N, \quad i,k = 1, \cdots, N,\\
&w_{i,j,k+1/2}, \quad k = 0, \cdots, N, \quad i,j = 1, \cdots, N.
\end{aligned}
$$




- Divergence:

$$
\begin{aligned}
(\nabla \cdot \mathbf{u})_{i,j,k} \quad = \quad & (u_{i+1/2,j,k} - u_{i-1/2,j,k} + \\
& v_{i,j+1/2,k} - v_{i,j-1/2,k} + \\
& w_{i,j,k+1/2} - w_{i,j,k-1/2})/h
\end{aligned}
$$

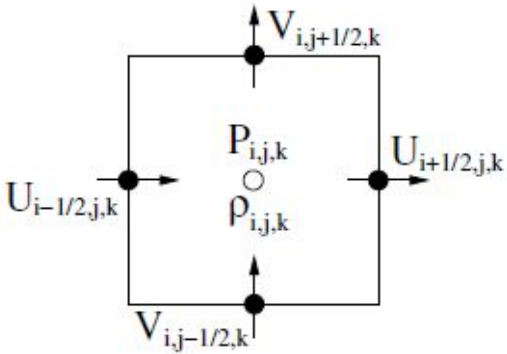


# Implementation: Semi-Lagrangian method

**Notations:**

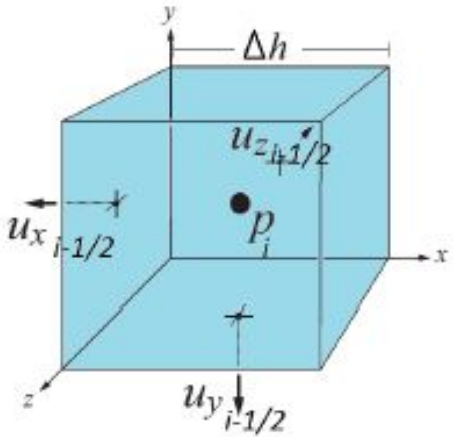
- Pressure Gradient($\nabla p=(px, py, pz)$):

$$
\begin{aligned}
(p_x)_{i+1/2,j,k} &= (p_{i+1,j,k} - p_{i,j,k})/h, \\
(p_y)_{i,j+1/2,k} &= (p_{i,j+1,k} - p_{i,j,k})/h, \\
(p_z)_{i,j,k+1/2} &= (p_{i,j,k+1} - p_{i,j,k})/h.
\end{aligned}
$$




- Force and velocity update:

$$
\begin{aligned}
u_{i+1/2,j,k} &+= \Delta t\ (f^1_{i,j,k} + f^1_{i+1,j,k})/2, \\
v_{i,j+1/2,k} &+= \Delta t\ (f^2_{i,j,k} + f^2_{i,j+1,k})/2, \\
w_{i,j,k+1/2} &+= \Delta t\ (f^3_{i,j,k} + f^3_{i,j,k+1})/2.
\end{aligned}
$$

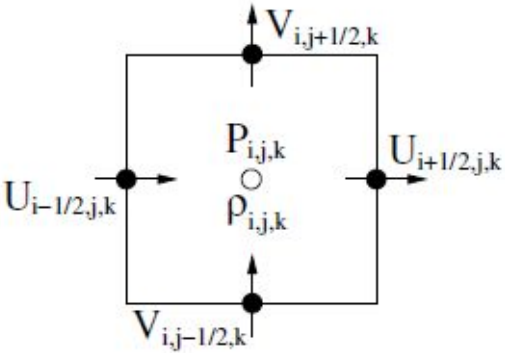


# Implementation: Semi-Lagrangian method

**Notations:**

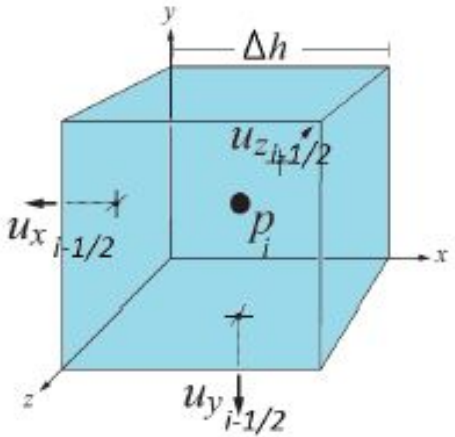
- Vorticity, first calculate velocity at cell center:

$$\begin{aligned}\bar{u}_{i,j,k} &= (u_{i-1/2,j,k} + u_{i+1/2,j,k})/2,\\ \bar{v}_{i,j,k} &= (v_{i,j-1/2,j} + v_{i,j+1/2,j})/2,\\ \bar{w}_{i,j,k} &= (w_{i,j,k-1/2} + w_{i,j,k+1/2})/2.\end{aligned}$$




Then vorticity ω=(ω1,ω2,ω3):

$$\begin{aligned}\omega^1_{i,j,k} &= (\bar{w}_{i,j+1,k} - \bar{w}_{i,j-1,k} - \bar{v}_{i,j,k+1} + \bar{v}_{i,j,k-1})/2h,\\ \omega^2_{i,j,k} &= (\bar{u}_{i,j,k+1} - \bar{u}_{i,j,k-1} - \bar{w}_{i+1,j,k} + \bar{w}_{i-1,j,k})/2h,\\ \omega^3_{i,j,k} &= (\bar{v}_{i+1,j,k} - \bar{v}_{i-1,j,k} - \bar{u}_{i,j+1,k} + \bar{u}_{i,j-1,k})/2h.\end{aligned}$$



# Implementation: Semi-Lagrangian method

Loop:

1) Update source: set initial density, temperature velocity for corresponding source grids.(Sample rendering particles here if applicable)
2) Advect velocity using Semi-Lagrangian method(linear here but you can apply more accurate scheme like Runge-Kutta):
    a) Obtain velocity *u* at position *x_n* at time *t_n*
    b) Calculate *x_star = x_n - u*Δt*
    c) Interpolate to get new velocity *u_star* as velocity at position *x_n* at time *t_(n+1)* (can be linear interpolation or cubic as described in the paper)
3) Apply buoyancy and vorticity confinement force and update *u_star*
4) Solve Poisson's equation and enforce incompressibility:

$$\nabla^2 p = \frac{1}{\Delta t}\nabla \cdot \mathbf{u}^* \qquad\qquad \mathbf{u} = \mathbf{u}^* - \Delta t \nabla p.$$

5) Advect temperature and density(like how you advect velocity in Semi-Lagrangian way)
6) Advect rendering particles if applicable(obtain velocity for particles and update their position)

# Visualization of Smoke

**Two ways of rendering**:

- **Particle rendering:**Adding particles to your simulation, letting them advect along the velocity field, and then rendering particle frames. The basecode we are releasing has this capability! (Example: link)
  - Need sample particles during simulation
  - Need interpolate velocity field to particle and do advection
  - Recommend .bgeo file(can color attributes like velocity)
- **Volume rendering:**There is no need for OpenVDB to do volume rendering anymore! All you need to do is to output the density value at each cell's center as a .bgeo file, and we will provide corresponding Houdini file to help you render it.
  - Need dump out density value at each cell center
  - Require a .bgeo file(otherwise won't be able to do volume render)

# Smoke Demo

*Implementation combines methods from Stable Fluids (Stam et al. 1999) and Visual Simulation of Smoke (Fedkiw et al. 2001)*

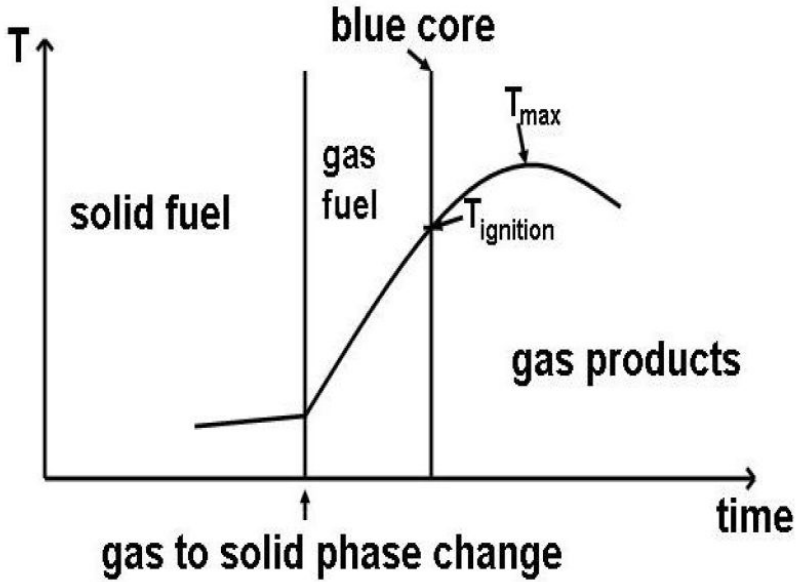
# Fire

# Fire Overview

- Fire Challenges
- Physically Based Fire Model
  - Nguyen et al. 2002
- Implementing the Model
- Rendering Flames
- Fire Demos

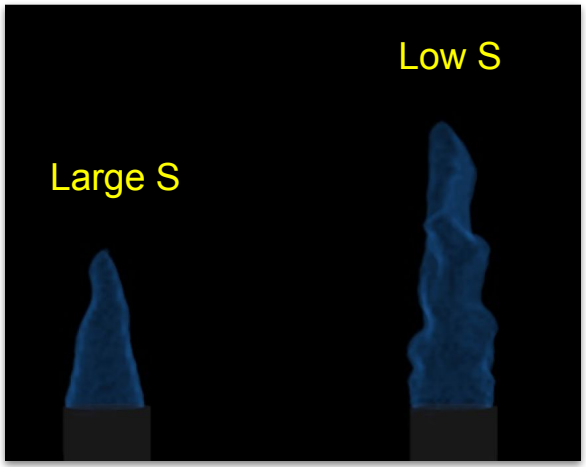
# Fire Challenges

- Fire is complex, has many visual components
  - Blue Core
  - Blackbody Radiation (from product)
  - Smoke and Soot
- Fire is a reaction, has phase changes
  - Gaseous fuel → hot gas product
  - Solid fuel → hot gas product
- Physically correct color is hard



# Physically Based Fire Model: Blue Core

- Need to model the blue core
  - Barrier between the fuel and hot gas product:
    - Modeled as an implicit surface (level set) adjacent to the blue core
  - Implicit surface moves with velocity equal to that of the unreacted fuel + a flame speed, S
    - Can model the rate at which fuel is consumed at this barrier:



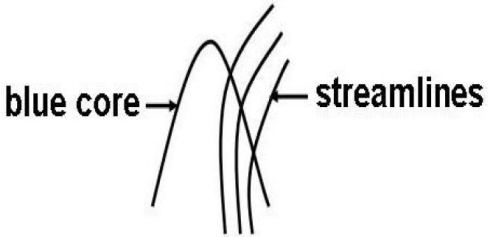

*Size of S affects size of blue core*

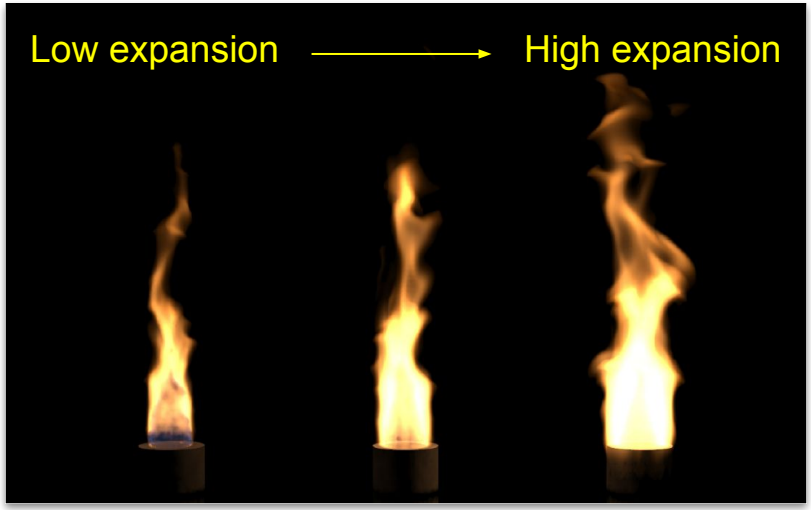
$$v_f A_f = S A_S$$

$v_f$ = velocity of fuel
S = rate of fuel consumption
$A_f$ = cross section of fuel injection site
$A_S$ = blue core surface area

# Physically Based Fire Model: Hot Gas Product

- Need to model the fuel and hot gaseous product interacting
  - Hot gas product expands and forms curved flow lines
  - To model expansion use densities and conservation of mass:
    - Expansion parameter given as ratio of densities ($\rho_f/\rho_h$)
    - Need equation to interpolate fuel and product velocities across the fuel barrier



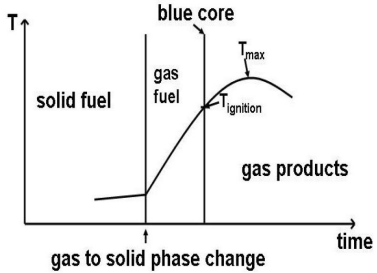



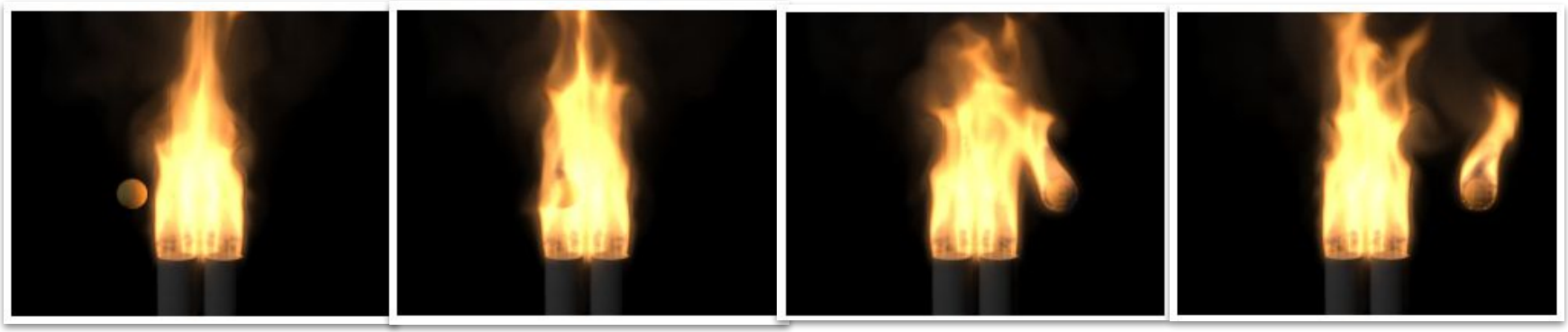

$$\rho_h(V_h - D) = \rho_f(V_f - D)$$

$\rho_h$ = density of hot gas product
$V_h$ = velocity of hot gas product
$D = V_f - S$
$\rho_f$ = density of fuel
$V_f$ = velocity of fuel

# Implementing the Physical Model

- Voxel based storage (velocities on faces still)
- Use separate set of Euler equations for gas fuel and hot gaseous products
  - Interpolate between these velocities when crossing the implicit surface
- Keeping track of temperature and density
  - Model by keeping track of how long since passed the blue core surface
- Flammable objects:
  - Voxels have temperature thresholds
  - When object filled voxel reaches its threshold, change voxel to be filled with fuel



# Rendering Fire

- Fire can be rendered largely the same as smoke
- Two new challenges to rendering fire
  - Fire emits light (radiance) from a complex shape
  - Fire is bright enough that our eyes adapt to its color
- Recursive ray marching used to compute radiance
  - Break ray into segments and evaluate recursively towards the origin
  - Use radiative transport equation to compute
- Von Kries transformation maps b/w color spaces
  - Assume human eye is adapted to color associated with highest temperature in the fire
  - Map this spectrum to other color spaces to get RGB

*Hot gaseous product and soot emit radiance that illuminates the smoke*

# Fire Demos

Student implementation of *Physically based modeling and animation of fire* (Nguyen et al. 2002)

# Fire Demos


*NVIDIA FlameWorks: combines a grid-based fluid simulator with volumetric rendering engine*

# Annotated References


- Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. 2001. Visual simulation of smoke. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH '01). ACM, New York, NY, USA, 15-22. DOI: https://doi.org/10.1145/383259.383260
    - This work introduces a grid based Eulerian smoke model that addresses common issues of stability and numerical dissipation by introducing “vorticity confinement” that injects lost energy at physically correct locations. Additionally, the authors address the interaction between smoke and moving objects.
- Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. 2002. Physically based modeling and animation of fire. ACM Trans. Graph. 21, 3 (July 2002), 721-728. DOI=http://dx.doi.org/10.1145/566654.566643
    - This work introduces a physically based model for simulating and animating realistic flame effects using a grid-based Eulerian model as a base. Additionally, the authors propose stable methods to realistically maintain turbulence within the flames, model the different phases and components of a flame, and to accurately render both the intricacies of the flame itself as well as the accurate coloring of the flame.