GAMES 201

Advanced Physics Engines 2020: A Hands-on Tutorial

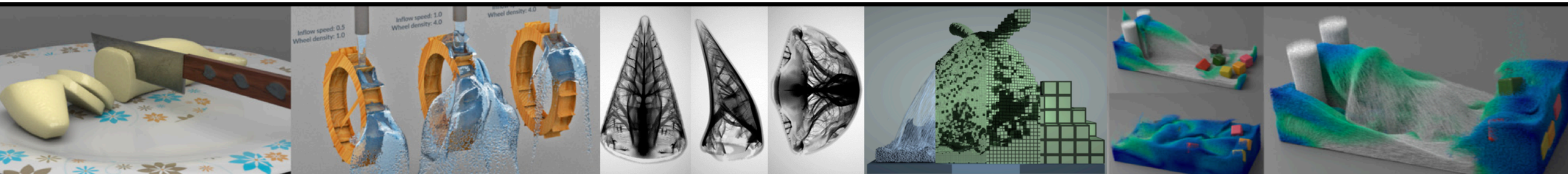
高级物理引擎实战2020 （基于太极编程语言）

第六讲：混合欧拉-拉格朗日视角(1)

Yuanming Hu 胡渊鸣 MIT CSAIL

麻省理工学院 计算机科学与人工智能实验室

Taichi Programming Language



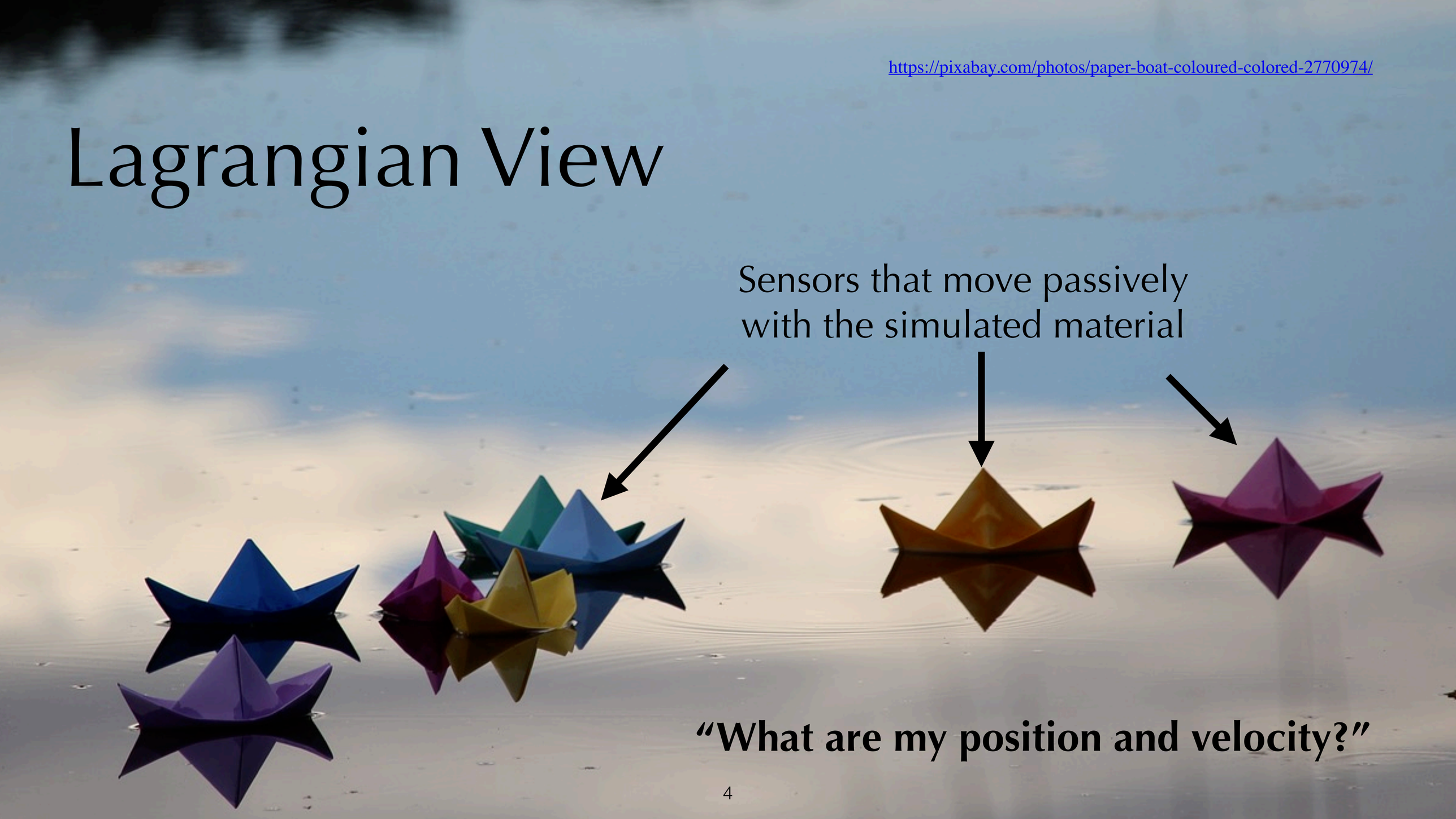
Schedule

- The following two lectures will be focused on **hybrid Lagrangian-Eulerian methods**
- **July 20:**
  - Overview
  - Particle advection schemes:
    - Particle-in-cell (**PIC**), Affine PIC (**APIC**), Polynomial PIC (**PolyPIC**)
    - Fluid Implicit Particles (**FLIP**)
  - Material Point Method basics
- **July 27:**
  - Moving Least Squares MPM (MLS-MPM), theory and implementation
  - Constitutive models in MPM
  - Lagrangian forces in MPM
  - Implicit MPM
  - Advanced Taichi features

Lagrangian v.s. Eulerian: Two Views of Continuums

https://pixabay.com/photos/paper-boat-coloured-colored-2770974/

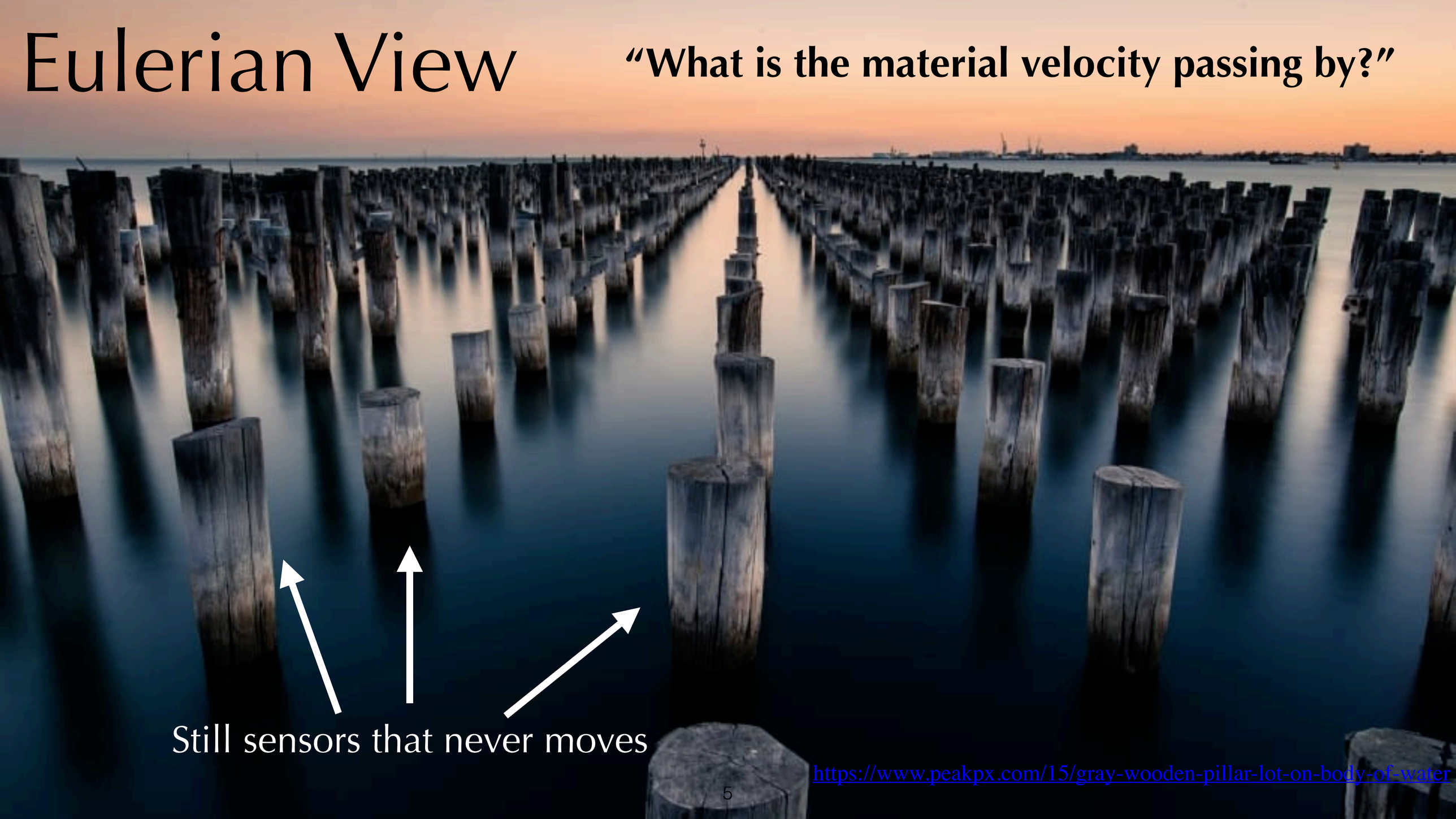
Lagrangian View



“What are my position and velocity?”

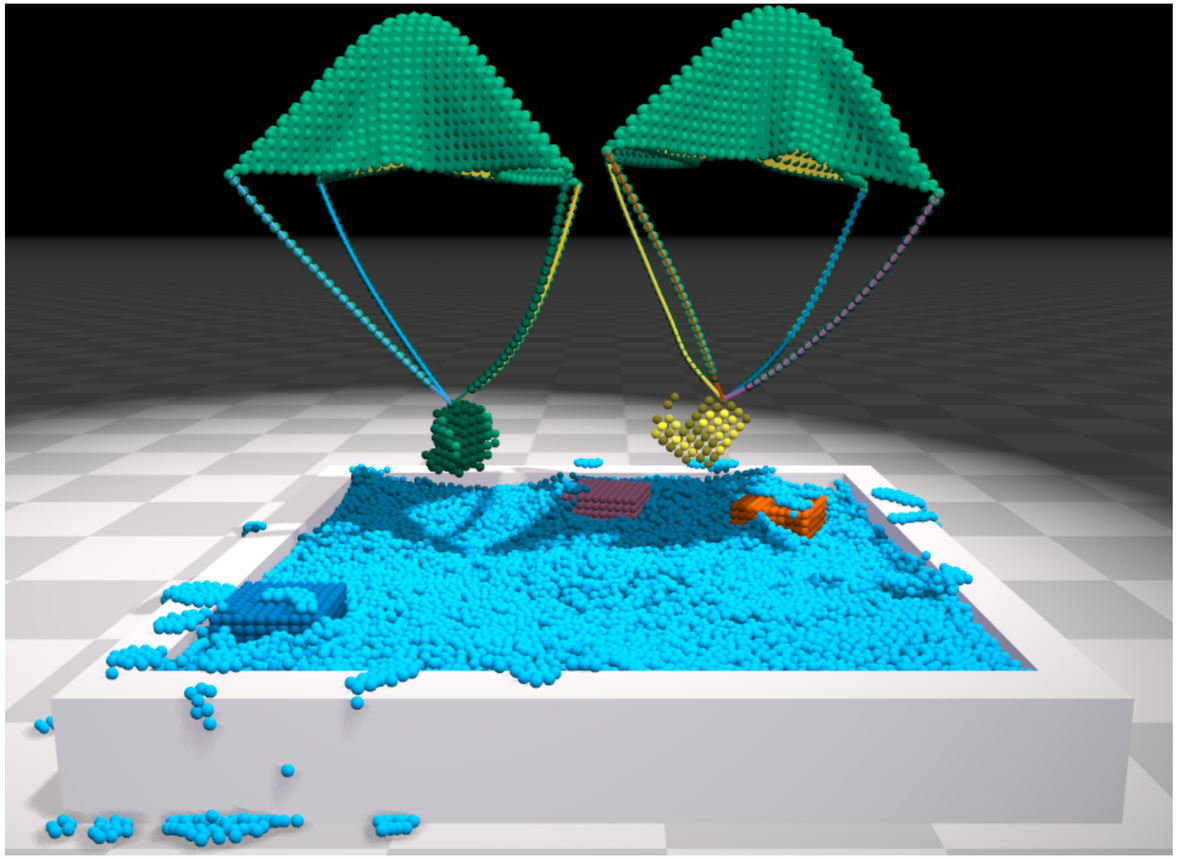
Eulerian View

“What is the material velocity passing by?”



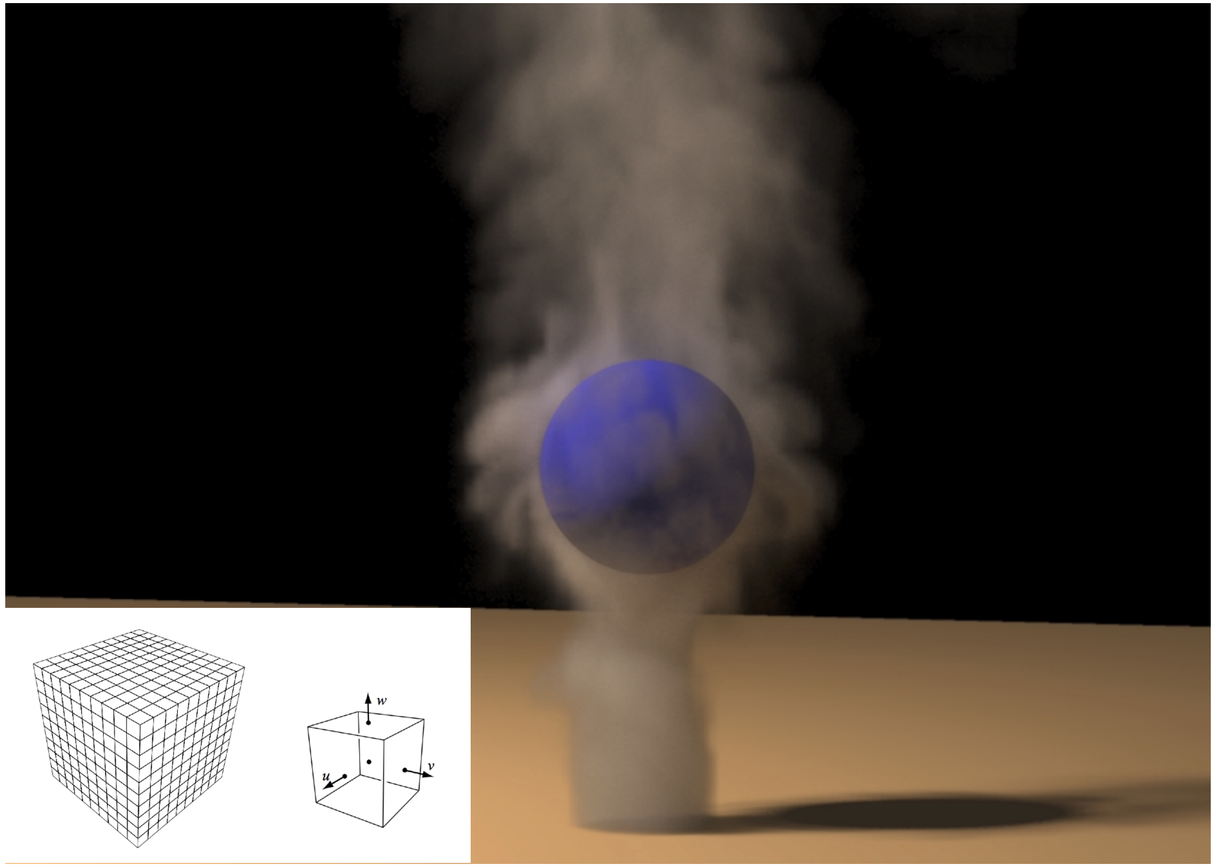
https://www.peakpx.com/15/gray-wooden-pillar-lot-on-body-of-water

Deformable Body Simulation



Lagrangian representation

[Macklin et al. 2014,
Unified Particle Physics for Real-Time Applications]



Eulerian representation

[Fedkiw 2001, Visual Simulation of Smoke]

Which is better?

Key factors to consider…

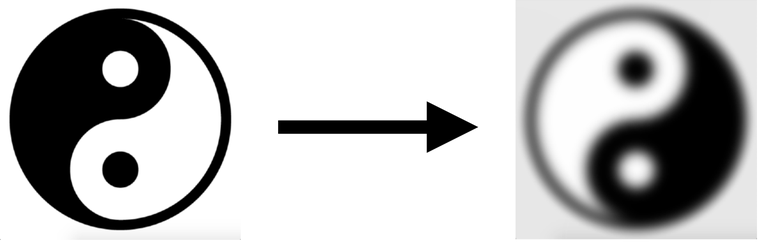
I.e., define "better":

- **Conservation of physical quantities**
  - Momentum
  - Angular momentum
  - Volume (incompressibility)
  - Energy (low dissipation)
  - …
- **Performance (parallelism & locality on modern hardware)**
- **Complexity**
- **…**

Hybrid Eulerian-Lagrangian Schemes

Motivation

- **Recall that a fluid solver usually has two components:**
  - **Advection** (evolving the fields)
  - **Projection** (enforcing incompressibility)
- **Eulerian grids are really good at projection:**
  - Easy to discretize
  - Efficient neighbor look-up
  - Easy to precondition (geometric multigrid)
- **But Eulerian grids are bad at advection…**
  - Dissipative: loss of energy and geometry



Motivation

- **Lagrangian particles are good at advection**
  - Simply move their coordinates :-)
  - More conservative (lower dissipation)
- **But projection on particles can be tricky:**
  - Tricky to discretize
  - Need complex data structures for neighbor look-up

- **Can we somehow smartly combine Lagrangian particles and Eulerian grids?**

(1)
Particle to Grid transfer
P2G

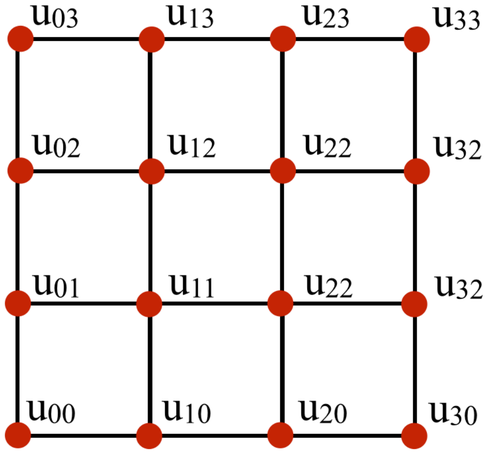
(2) Grid operations:

- Pressure projection
- Boundary conditions
- …

(3)
Grid to Particle transfer
G2P

(4) Particle operations:

- Move particles
- Update material
- …

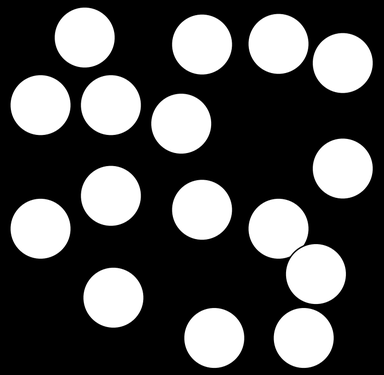


Eulerian Grids

(often auxiliary)

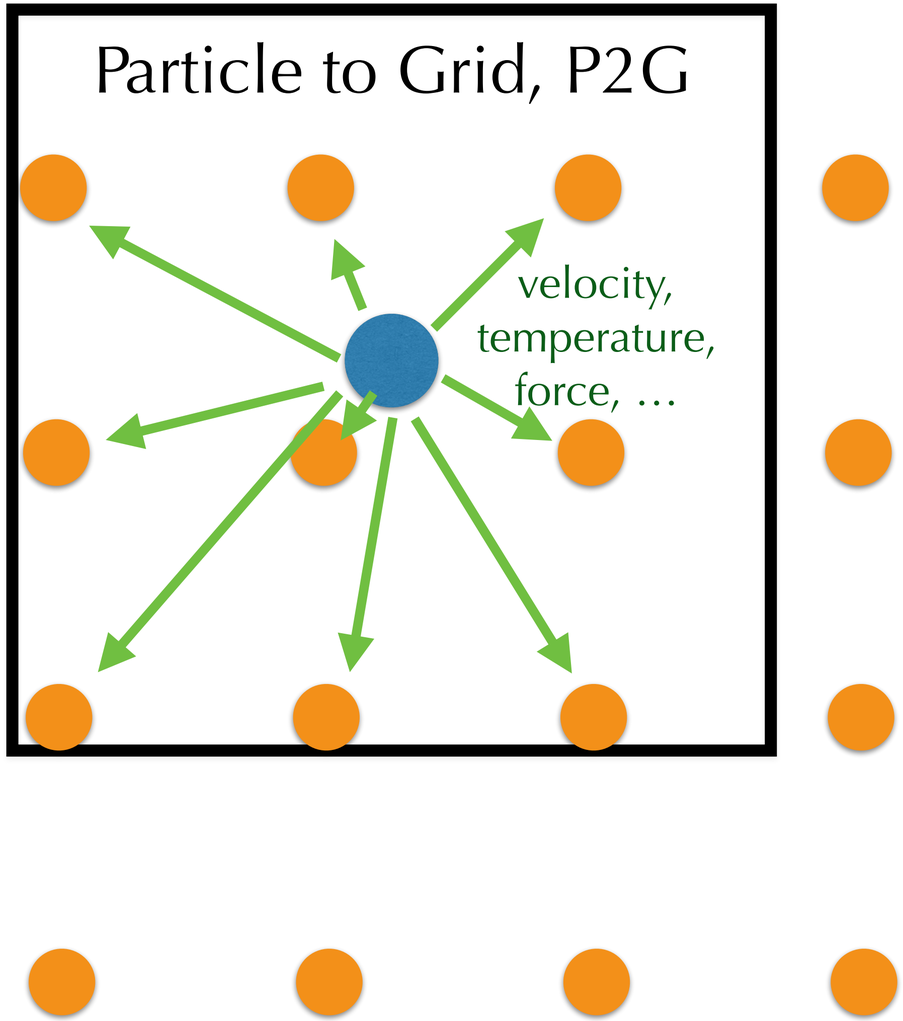
Lagrangian Particles

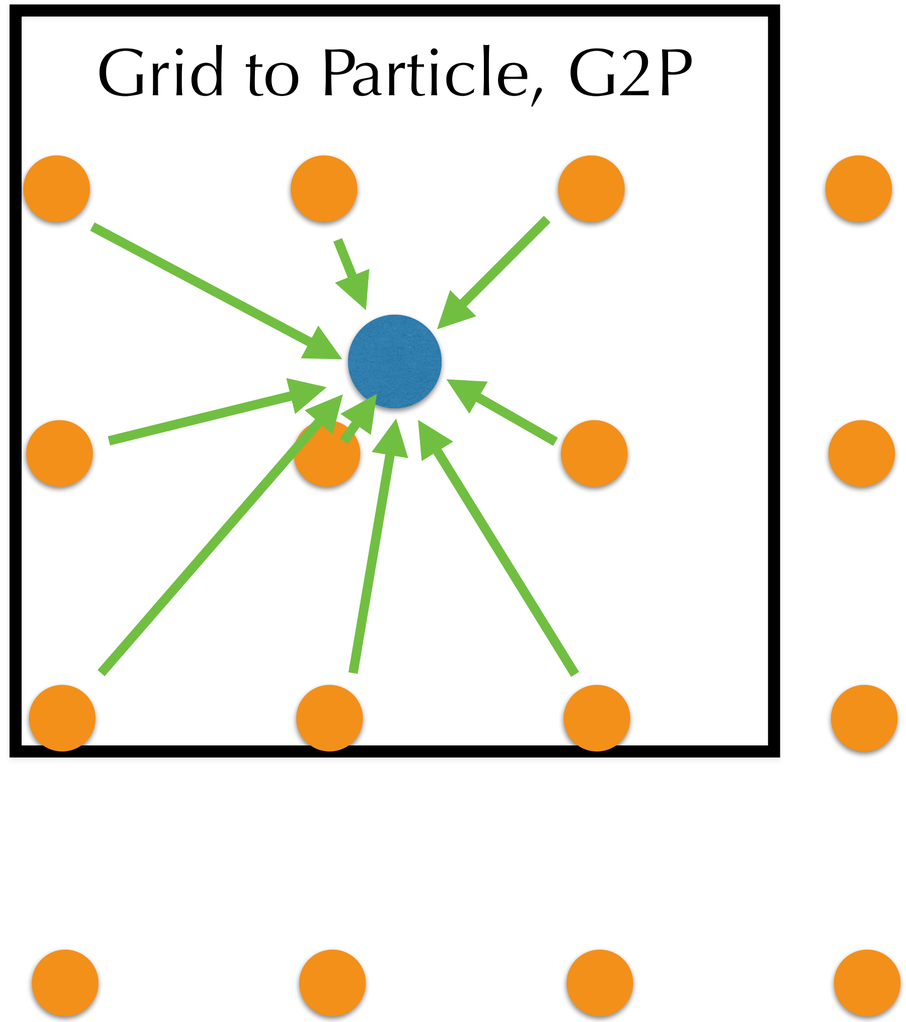
(which stores most of the information)



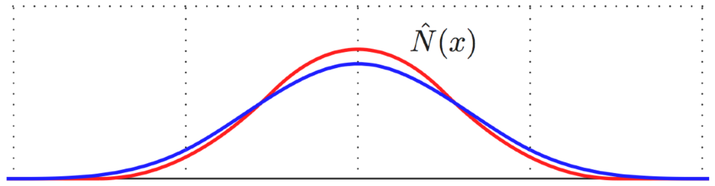
Particle-in-cell

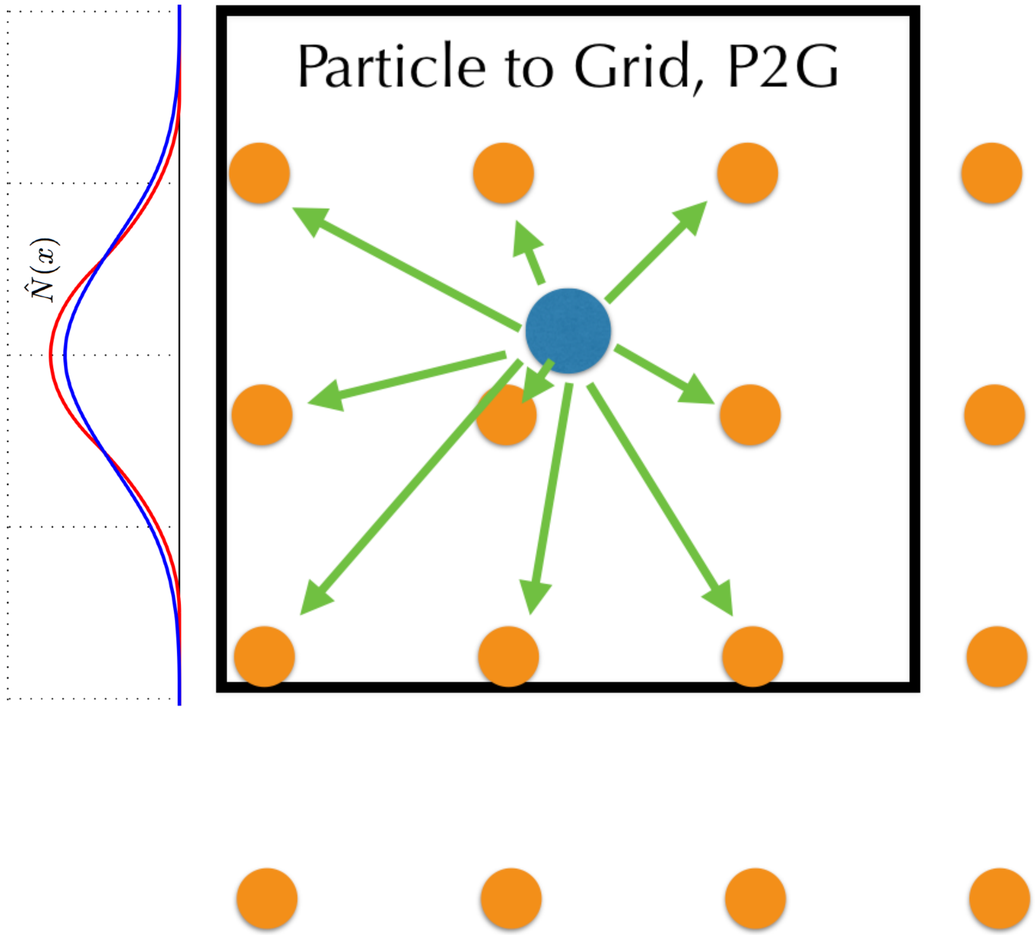
Harlow, F.H. (1964) The Particle-in-Cell Computing Method for Fluid Dynamics. Methods in Computational Physics, 3, 319-343.





The particle does **not** treat neighbors equally





Closer = more importance

B-Spline Kernels $\mathbf{N(x)}$

Linear

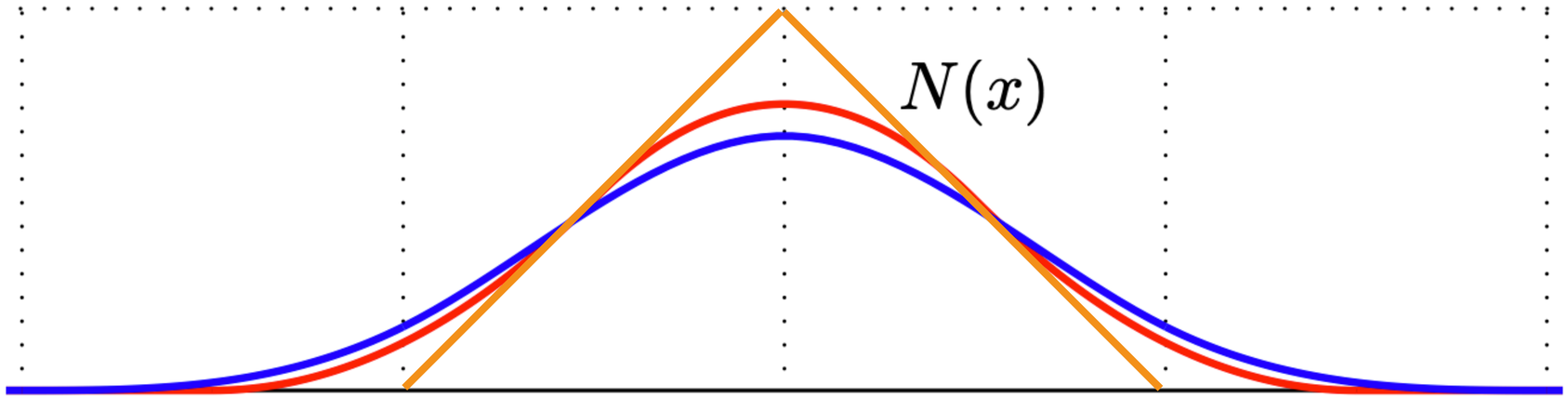
$$N(x) = \begin{cases} 1 - |x| & 0 \leqslant |x| < 1 \\ 0 & 1 \leqslant |x| \end{cases}$$

Quadratic

$$N(x) = \begin{cases} \frac{3}{4} - |x|^2 & 0 \leqslant |x| < \frac{1}{2} \\ \frac{1}{2}\left(\frac{3}{2} - |x|\right)^2 & \frac{1}{2} \leqslant |x| < \frac{3}{2} \\ 0 & \frac{3}{2} \leqslant |x| \end{cases}$$

Cubic

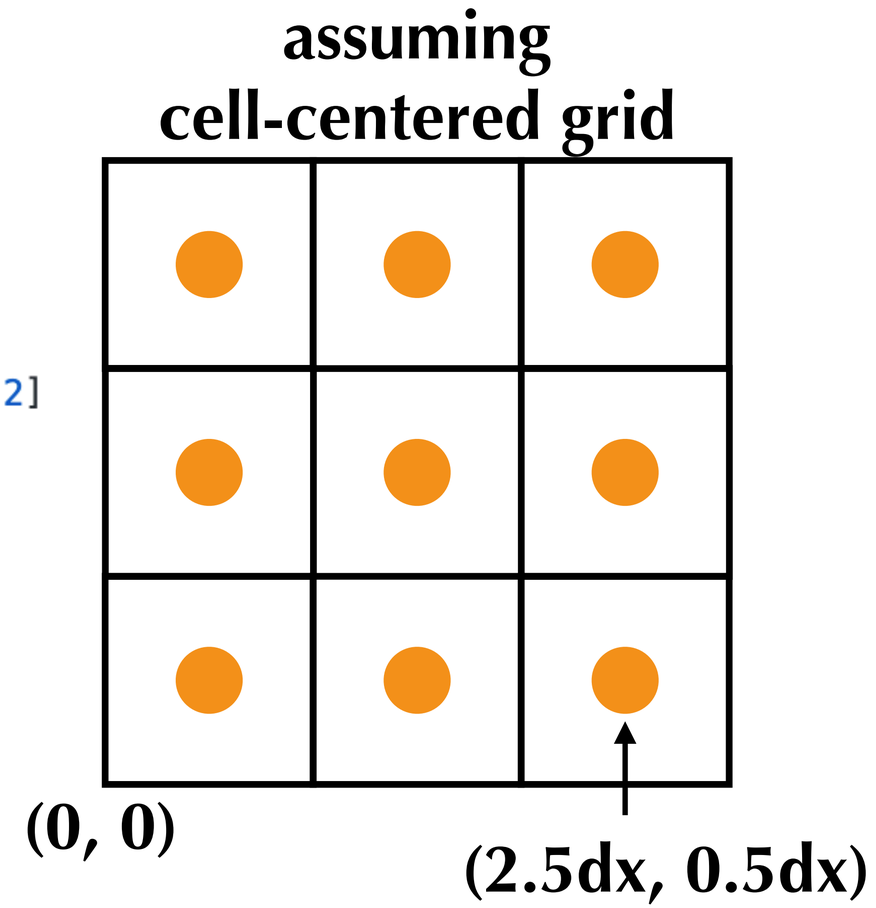
$$N(x) = \begin{cases} \frac{1}{2}|x|^3 - |x|^2 + \frac{2}{3} & 0 \leqslant |x| < 1 \\ \frac{1}{6}(2 - |x|)^3 & 1 \leqslant |x| < 2 \\ 0 & 2 \leqslant |x| \end{cases}$$



https://www.seas.upenn.edu/~cffjiang/research/mpmcourse/mpmcourse.pdf

PIC P2G code (transfer velocity)

```
for p in x:
    base = (x[p] * inv_dx - 0.5).cast(int)
    fx = x[p] * inv_dx - base.cast(float)
    # Quadratic B-spline
    w = [0.5 * (1.5 - fx) ** 2, 0.75 - (fx - 1) ** 2, 0.5 * (fx - 0.5) ** 2]
    for i in ti.static(range(3)):
        for j in ti.static(range(3)):
            offset = ti.Vector([i, j])
            weight = w[i][0] * w[j][1]
            grid_v[base + offset] += weight * v[p]
            grid_m[base + offset] += weight
```



PIC grid normalization code

```
54          for i, j in grid_m:
55              if grid_m[i, j] > 0:
56                  inv_m = 1 / grid_m[i, j]
57                  grid_v[i, j] = inv_m * grid_v[i, j]
```

PIC G2P code (gather velocity)

```
59      for p in x:
60          base = (x[p] * inv_dx - 0.5).cast(int)
61          fx = x[p] * inv_dx - base.cast(float)
62          # Quadratic B-spline
63          w = [
64              0.5 * (1.5 - fx) ** 2, 0.75 - (fx - 1.0) ** 2, 0.5 * (fx - 0.5) ** 2
65          ]
66          new_v = ti.Vector.zero(ti.f32, 2)
67          for i in ti.static(range(3)):
68              for j in ti.static(range(3)):
69                  weight = w[i][0] * w[j][1]
70                  new_v += weight * grid_v[base + ti.Vector([i, j])]
71
72          x[p] = clamp_pos(x[p] + v[p] * dt)
73          v[p] = new_v
```

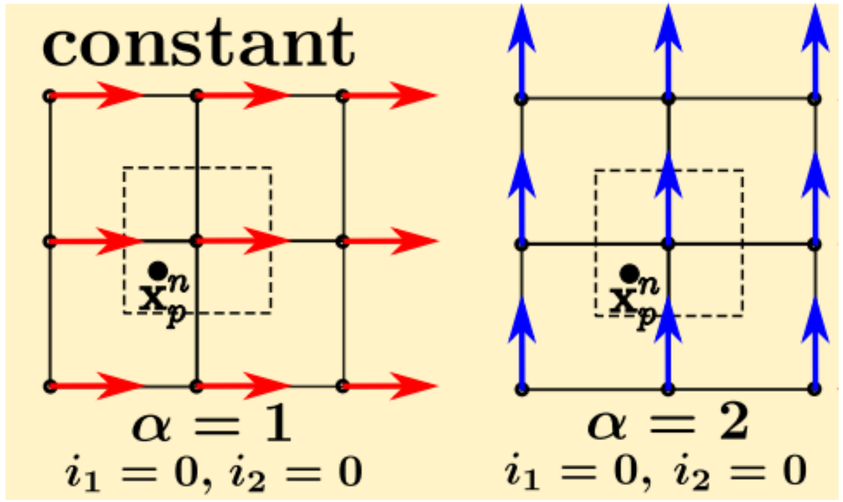
Combing PIC and grid-based Poisson solver

- **Simulation cycle**
  - P2G: scatter **velocity** from particles to grid
  - normalize velocity
  - Pressure projection
  - G2P: gather *divergence-free* **velocity** from grid to particles
    - Move particles according to the divergence-free velocity field
    - Use RK2/3/4 if necessary
- **Demo: http://yuanming-hu.github.io/fluid/ (FLIP blending=0)**
- **Does it look good?**

Let's run a very simple PIC simulation…

PIC G2P: information loss

(Assume we have only 1 particle)



Problem: <u>18</u> DoFs on grid, <u>2</u> DoFs on particle

Figure from *Fu et al 2017, A Polynomial Particle-In-Cell Method (SIGGRAPH Asia 2017)*

Reducing particle-in-cell dissipation

- **Two solutions:**

1. **Transfer more information: APIC, PolyPIC**
2. **Transfer the delta: FLIP (later in this lecture)**

Affine Particle-in-cell (APIC)

Jiang et al., SIGGRAPH & JCP 2016

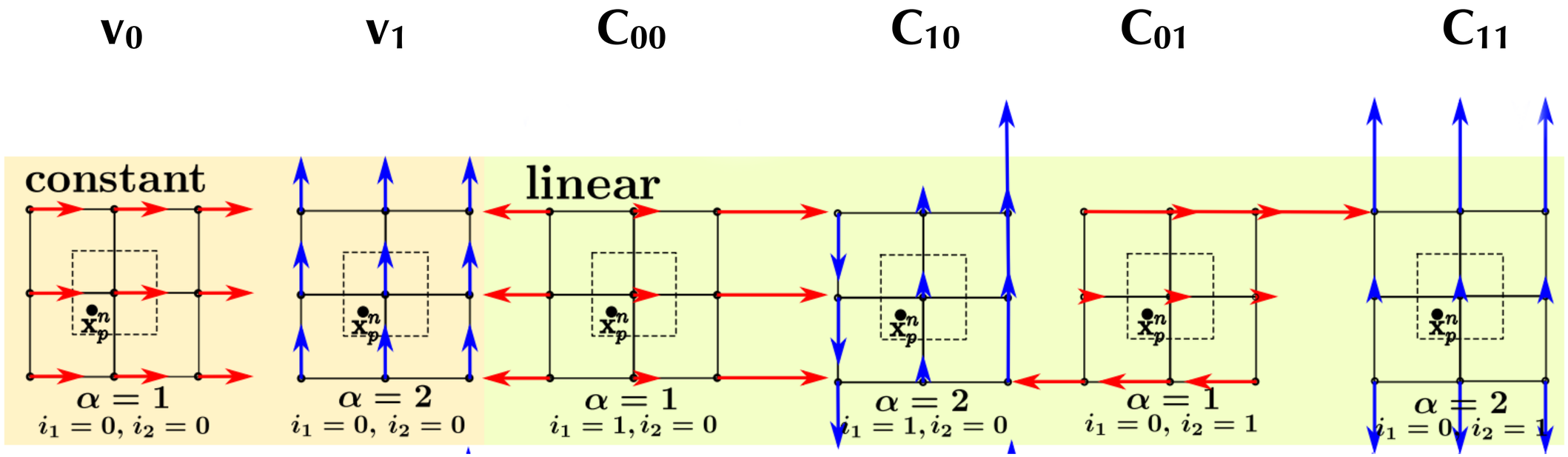
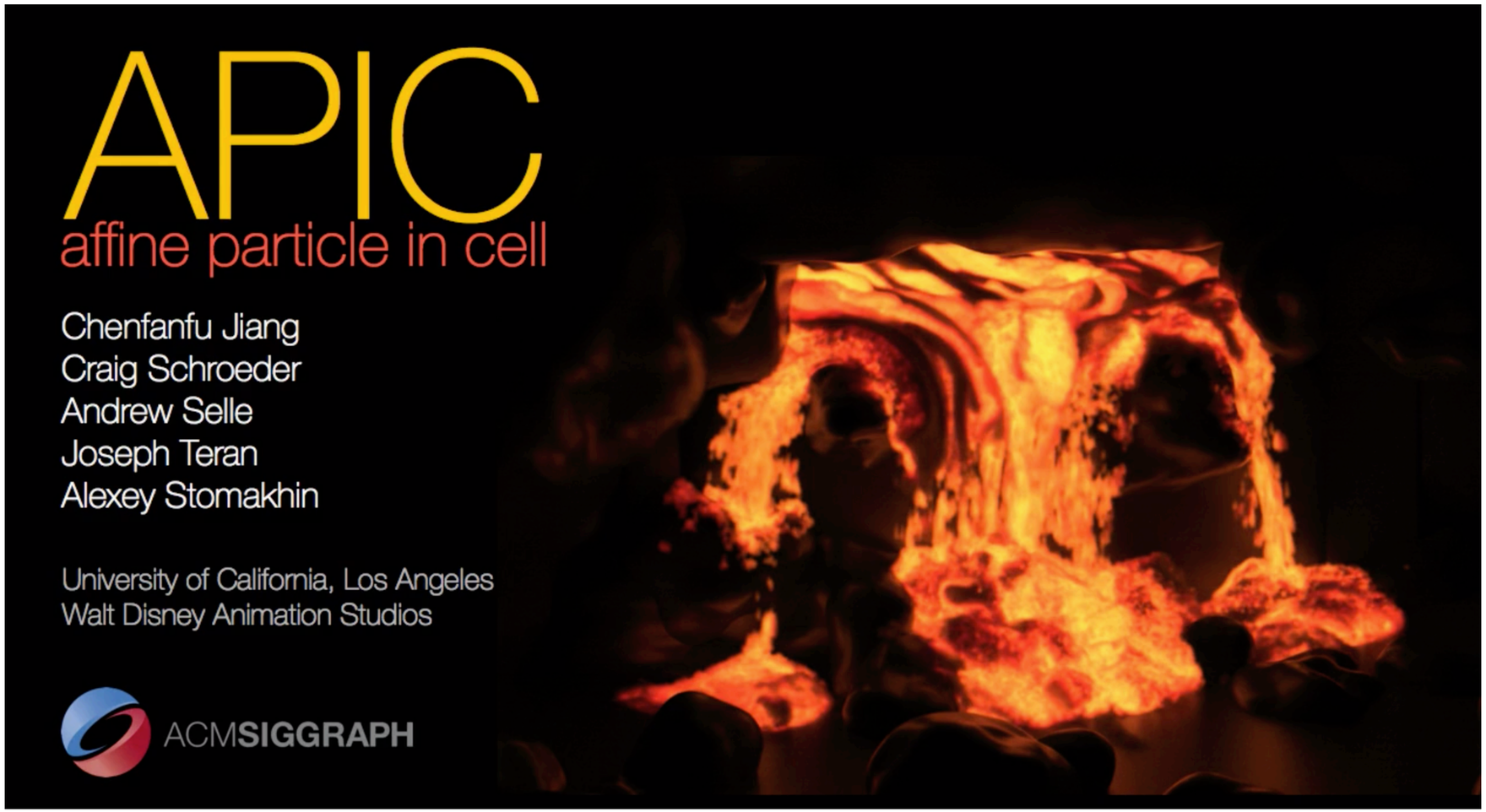


Figure from *Fu et al 2017, A Polynomial Particle-In-Cell Method (SIGGRAPH Asia 2017)*

Homework (highly recommended!)



✦ **Watch Bilibili: https://www.bilibili.com/video/BV1St411G7nm/**

APIC conserves angular momentum!

5.3.1 Particle to grid

Proposition 5.4. *Angular momentum is conserved during the APIC transfer from particles to the grid.* $\boldsymbol{L}_{tot}^{G,n} = \boldsymbol{L}_{tot}^{P,n}$ *under transfer 3.*

Proof. The angular momentum on the grid after transferring from particles is

$$
\begin{aligned}
\boldsymbol{L}_{tot}^{G,n} &= \sum_i \boldsymbol{x}_i^n \times m_i^n \boldsymbol{v}_i^n \\
&= \sum_p \sum_i \boldsymbol{x}_i^n \times m_p w_{ip}^n (\boldsymbol{v}_p^n + \boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} (\boldsymbol{x}_i^n - \boldsymbol{x}_p^n)) \\
&= \sum_p \sum_i \boldsymbol{x}_i^n \times m_p w_{ip}^n \boldsymbol{v}_p^n + \sum_p \sum_i \boldsymbol{x}_i^n \times m_p w_{ip}^n \boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{x}_i^n - \sum_p \left(\sum_i w_{ip}^n \boldsymbol{x}_i^n \right) \times m_p \boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{x}_p^n \\
&= \sum_p \boldsymbol{x}_p^n \times m_p \boldsymbol{v}_p^n + \sum_p m_p \sum_i w_{ip}^n \boldsymbol{x}_i^n \times (\boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{x}_i^n) - \sum_p m_p \boldsymbol{x}_p^n \times (\boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{x}_p^n) \\
&= \sum_p \boldsymbol{x}_p^n \times m_p \boldsymbol{v}_p^n + \sum_p m_p \left(\sum_i w_{ip}^n \boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{x}_i^n (\boldsymbol{x}_i^n)^T \right)^T : \boldsymbol{\epsilon} - \sum_p m_p \left(\boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{x}_p^n (\boldsymbol{x}_p^n)^T \right)^T : \boldsymbol{\epsilon} \\
&= \sum_p \boldsymbol{x}_p^n \times m_p \boldsymbol{v}_p^n + \sum_p m_p \left(\sum_i w_{ip}^n \boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{x}_i^n (\boldsymbol{x}_i^n)^T - \boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{x}_p^n (\boldsymbol{x}_p^n)^T \right)^T : \boldsymbol{\epsilon} \\
&= \sum_p \boldsymbol{x}_p^n \times m_p \boldsymbol{v}_p^n + \sum_p m_p \left(\boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \left(\sum_i w_{ip}^n \boldsymbol{x}_i^n (\boldsymbol{x}_i^n)^T - \boldsymbol{x}_p^n (\boldsymbol{x}_p^n)^T \right) \right)^T : \boldsymbol{\epsilon} \\
&= \sum_p \boldsymbol{x}_p^n \times m_p \boldsymbol{v}_p^n + \sum_p m_p \left(\boldsymbol{B}_p^n (\boldsymbol{D}_p^n)^{-1} \boldsymbol{D}_p^n \right)^T : \boldsymbol{\epsilon} \\
&= \sum_p \boldsymbol{x}_p^n \times m_p \boldsymbol{v}_p^n + \sum_p m_p (\boldsymbol{B}_p^n)^T : \boldsymbol{\epsilon} \\
&= \boldsymbol{L}_{tot}^{P,n}
\end{aligned}
$$

5.3.2 Grid to particle

Proposition 5.5. *Angular momentum is conserved during the APIC transfer from the grid to particles.* $\boldsymbol{L}_{tot}^{P,n+1} = \boldsymbol{L}_{tot}^{G,n+1}$ *und*

Proof. As before, the manipulation $(\boldsymbol{v}\boldsymbol{u}^T)^T : \boldsymbol{\epsilon} = \boldsymbol{u} \times \boldsymbol{v}$ is used to convert the permutation tensor into a cross product. Th
on the particles after transferring from the grid is

$$
\begin{aligned}
\boldsymbol{L}_{tot}^{P,n+1} &= \sum_p \boldsymbol{x}_p^{n+1} \times m_p \boldsymbol{v}_p^{n+1} + \sum_p m_p (\boldsymbol{B}_p^{n+1})^T : \boldsymbol{\epsilon} \\
&= \sum_p \boldsymbol{x}_p^{n+1} \times m_p \boldsymbol{v}_p^{n+1} + \sum_p m_p \left(\sum_i w_{ip}^n \tilde{\boldsymbol{v}}_i^{n+1} (\boldsymbol{x}_i^n - \boldsymbol{x}_p^n)^T \right)^T : \boldsymbol{\epsilon} \\
&= \sum_p \boldsymbol{x}_p^{n+1} \times m_p \boldsymbol{v}_p^{n+1} + \sum_p m_p \sum_i w_{ip}^n (\boldsymbol{x}_i^n - \boldsymbol{x}_p^n) \times \tilde{\boldsymbol{v}}_i^{n+1} \\
&= \sum_p \boldsymbol{x}_p^{n+1} \times m_p \boldsymbol{v}_p^{n+1} - \sum_p m_p \sum_i w_{ip}^n \boldsymbol{x}_p^n \times \tilde{\boldsymbol{v}}_i^{n+1} + \sum_p m_p \sum_i w_{ip}^n \boldsymbol{x}_i^n \times \tilde{\boldsymbol{v}}_i^{n+1} \\
&= \sum_p \boldsymbol{x}_p^{n+1} \times m_p \boldsymbol{v}_p^{n+1} - \sum_p m_p \boldsymbol{x}_p^n \times \sum_i w_{ip}^n \tilde{\boldsymbol{v}}_i^{n+1} + \sum_i \left(\sum_p m_p w_{ip}^n \right) \boldsymbol{x}_i^n \times \tilde{\boldsymbol{v}}_i^{n+1} \\
&= \sum_p \boldsymbol{x}_p^{n+1} \times m_p \boldsymbol{v}_p^{n+1} - \sum_p m_p \boldsymbol{x}_p^n \times \boldsymbol{v}_p^{n+1} + \sum_i m_i^n \boldsymbol{x}_i^n \times \tilde{\boldsymbol{v}}_i^{n+1} \\
&= \sum_p (\boldsymbol{x}_p^{n+1} - \boldsymbol{x}_p^n) \times m_p \boldsymbol{v}_p^{n+1} + \sum_i \boldsymbol{x}_i^n \times m_i^n \tilde{\boldsymbol{v}}_i^{n+1} \\
&= \sum_p \Delta t \boldsymbol{v}_p^{n+1} \times m_p \boldsymbol{v}_p^{n+1} + \boldsymbol{L}_{tot}^{G,n+1} \\
&= \boldsymbol{L}_{tot}^{G,n+1}
\end{aligned}
$$

https://www.seas.upenn.edu/~cffjiang/research/apic/tech-doc.pdf

An angular momentum conserving affine-particle-in-cell method, Jiang et al., JCP 2017

APIC P2G, G2P

```python
for p in x:
    base = (x[p] * inv_dx - 0.5).cast(int)
    fx = x[p] * inv_dx - base.cast(float)
    # Quadratic B-spline
    w = [0.5 * (1.5 - fx) ** 2, 0.75 - (fx - 1) ** 2, 0.5 * (fx - 0.5) ** 2]
    affine = C[p]
    for i in ti.static(range(3)):
        for j in ti.static(range(3)):
            offset = ti.Vector([i, j])
            dpos = (offset.cast(float) - fx) * dx
            weight = w[i][0] * w[j][1]
            grid_v[base + offset] += weight * (v[p] + affine @ dpos)
            grid_m[base + offset] += weight
```

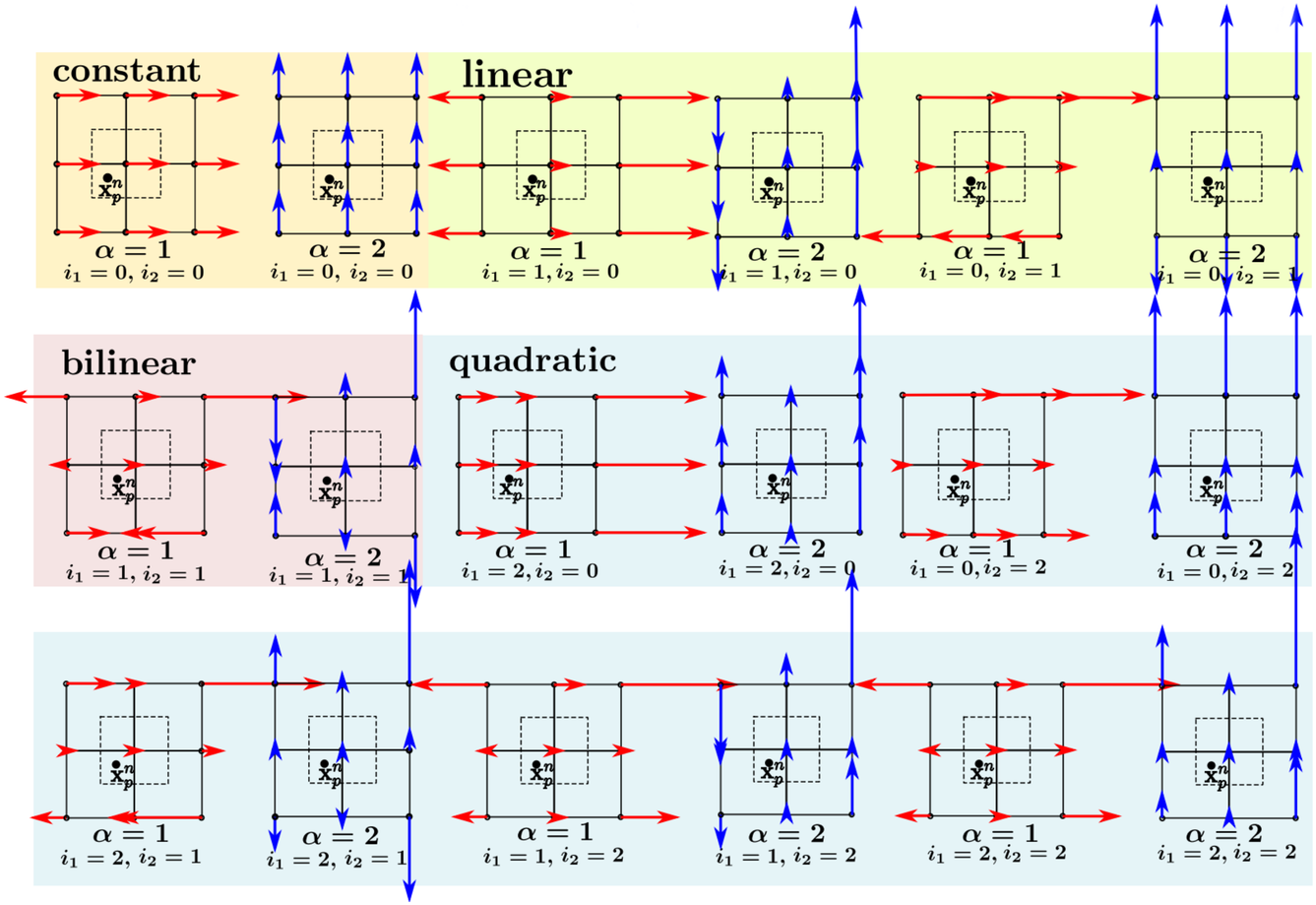
```python
for p in x:
    base = (x[p] * inv_dx - 0.5).cast(int)
    fx = x[p] * inv_dx - base.cast(float)
    # Quadratic B-spline
    w = [
        0.5 * (1.5 - fx) ** 2, 0.75 - (fx - 1.0) ** 2, 0.5 * (fx - 0.5) ** 2
    ]
    new_v = ti.Vector.zero(ti.f32, 2)
    new_C = ti.Matrix.zero(ti.f32, 2, 2)
    for i in ti.static(range(3)):
        for j in ti.static(range(3)):
            dpos = ti.Vector([i, j]).cast(float) - fx
            g_v = grid_v[base + ti.Vector([i, j])]
            weight = w[i][0] * w[j][1]
            new_v += weight * g_v
            new_C += 4 * weight * g_v.outer_product(dpos) * inv_dx

    x[p] = clamp_pos(x[p] + new_v * dt)
    v[p] = new_v
    C[p] = new_C
```

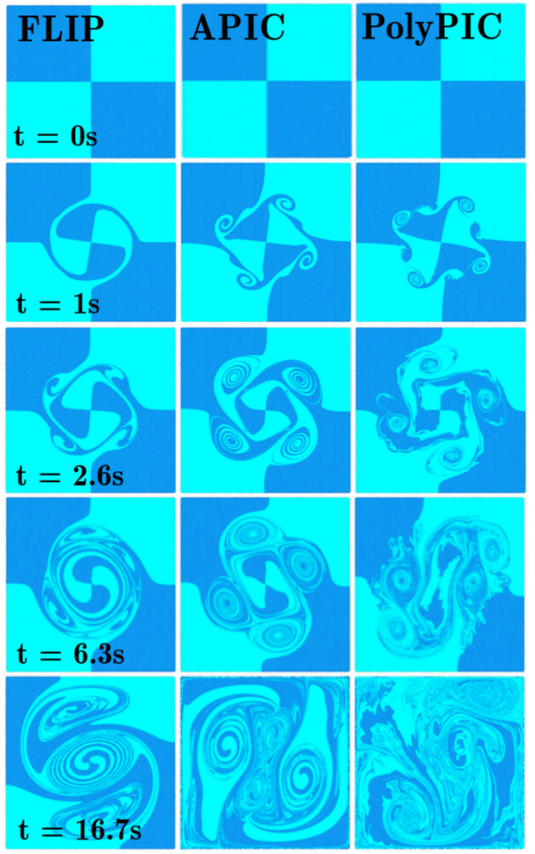
Let's run it…

PolyPIC

A Polynomial Particle-In-Cell Method, Fu et al. 2017

18 modes=9 nodes X 2 DoFs per node: Lossless transfer!





Figures from *Fu et al 2017, A Polynomial Particle-In-Cell Method (SIGGRAPH Asia 2017)*

Fluid implicit particles (FLIP)

- **Idea: don't gather the physical quantity. Gather the delta of the physical quantities before/after grid operation.**
  - grid op = pressure projection in incompressible fluid simulation
  - grid op = internal force computation in solid simulation (MPM)
- Note: some VFX people may use "FLIP" for "a fluid solver using Chorin-Style pressure projection with FLIP advection", but FLIP itself is just an advection scheme.

BRACKBILL, J. U., AND RUPPEL, H. M. 1986. FLIP: a method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. **JCP**

Yongning Zhu and Robert Bridson. 2005. Animating Sand As a Fluid. **SIGGRAPH 2005**

Combining PIC and FLIP

$$v_p^{t+1} = \mathbf{gather}(v_i^{t+1}) \quad v_p^{t+1} = v_p^t + \mathbf{gather}(v_i^{t+1} - v_i^t)$$

- **PIC is too dissipative, yet FLIP is too noisy: can we interpolate between the two methods?**
- **FLIP0.99 = 0.99 * FLIP + 0.01 * PIC**
- **Demo: http://yuanming-hu.github.io/fluid/ (adjust FLIP blending)**

PIC/FLIP/APIC/PolyPIC: which one to use?

- **Suggestion: start with APIC. PIC is almost never used in graphics**
  - APIC is easy to implement
  - APIC is storage friendly (no backup velocity field compared with FLIP)
  - APIC preserves angular momentum
  - APIC is stable
  - APIC leads to nice visual results
- APIC is the basis of MLS-MPM!

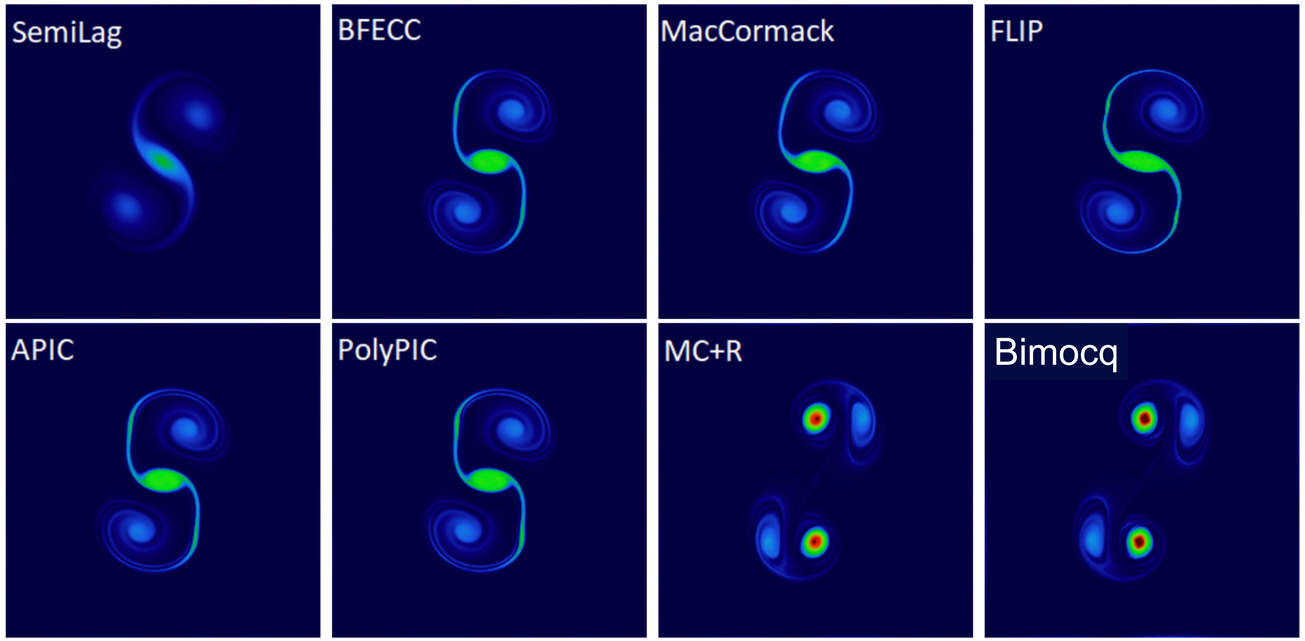


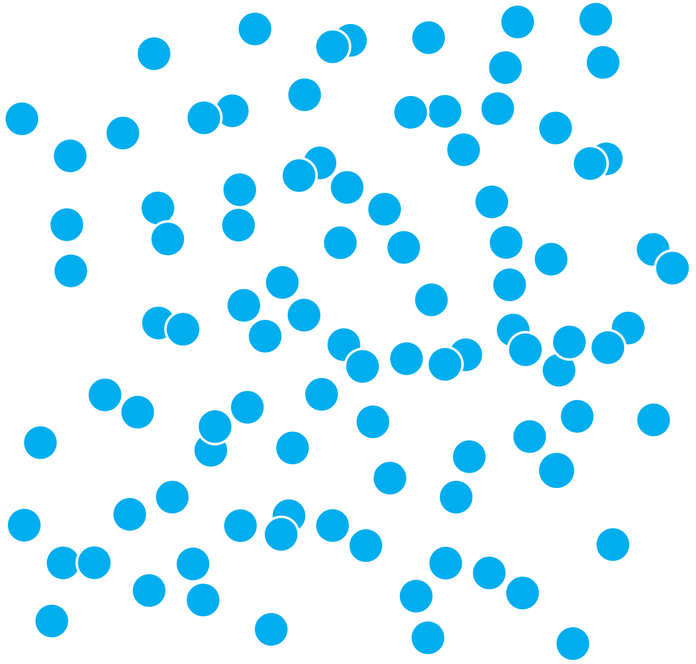
Figure from Qu et al., Efficient and Conservative Fluids Using Bidirectional Mapping, SIGGRAPH 2019

Material Point Method (MPM)

Material Point Method (MPM)

- **Hybrid Lagrangian-Eulerian simulation scheme**
  - Not just “advection”
  - Particle carries a lot more than velocity
- **Very hot research topic: at least 5 papers at SIGGRAPH 2020**
- **Invented by Sulsky & Schreyer in 1996**
- **First introduced to graphics in 2013**
  - A material point method for snow simulation (Stomakhin et al.)

The Material Point Method as of 2018

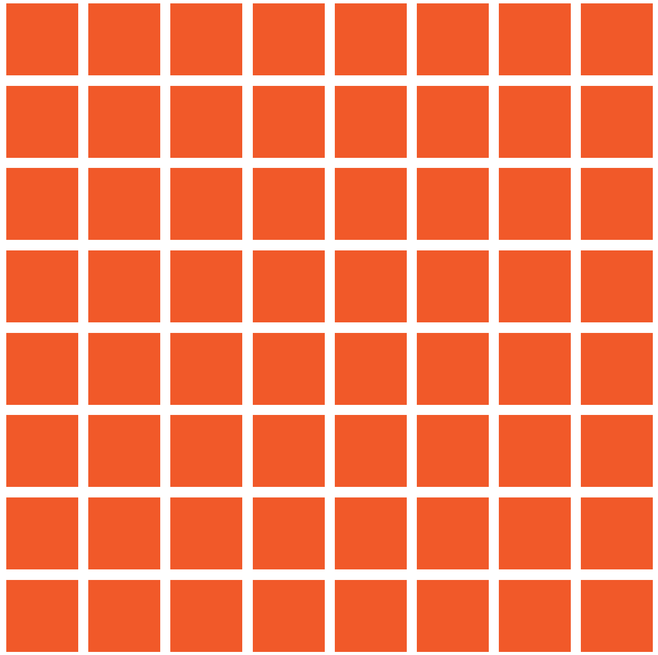


Particle to Grid (P2G)

Grid to Particle (G2P)

Transfer (Particle-in-Cell, PIC)

Affine PIC, APIC [Jiang et al. 2016]
Polynomial PIC, PolyPIC [Fu et al. 2017]
High-performance GIMP [Gao et al. 2017]
Moving Least Squares [Hu et al. 2018]
Compatible PIC [Hu et al. 2018]
...



Particles (Constitutive models)

Snow [Stomakhin et al. 2013],
Foam [Ram et al. 2015, Yue et al. 2015]
Sand [Klar et al. 2015, Pradhana et al 2017]

Grid

SPGrid [Setaluri et al. 2014],
OpenVDB [Museth 2013]
Multiple Grids [Pradhana et al. 2017]

The Material Point Method (MPM)

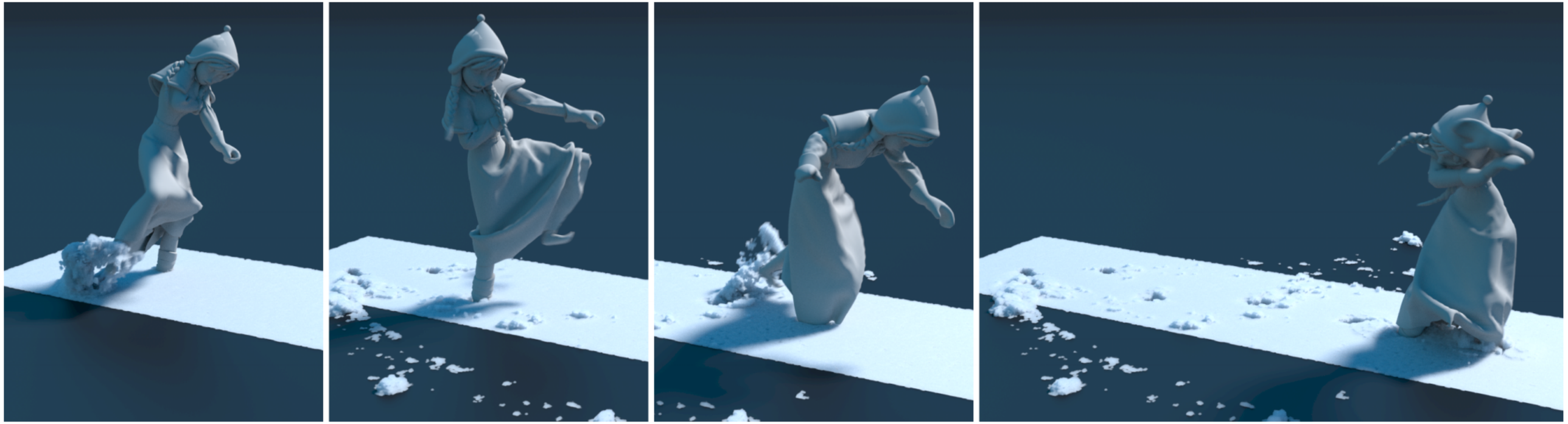


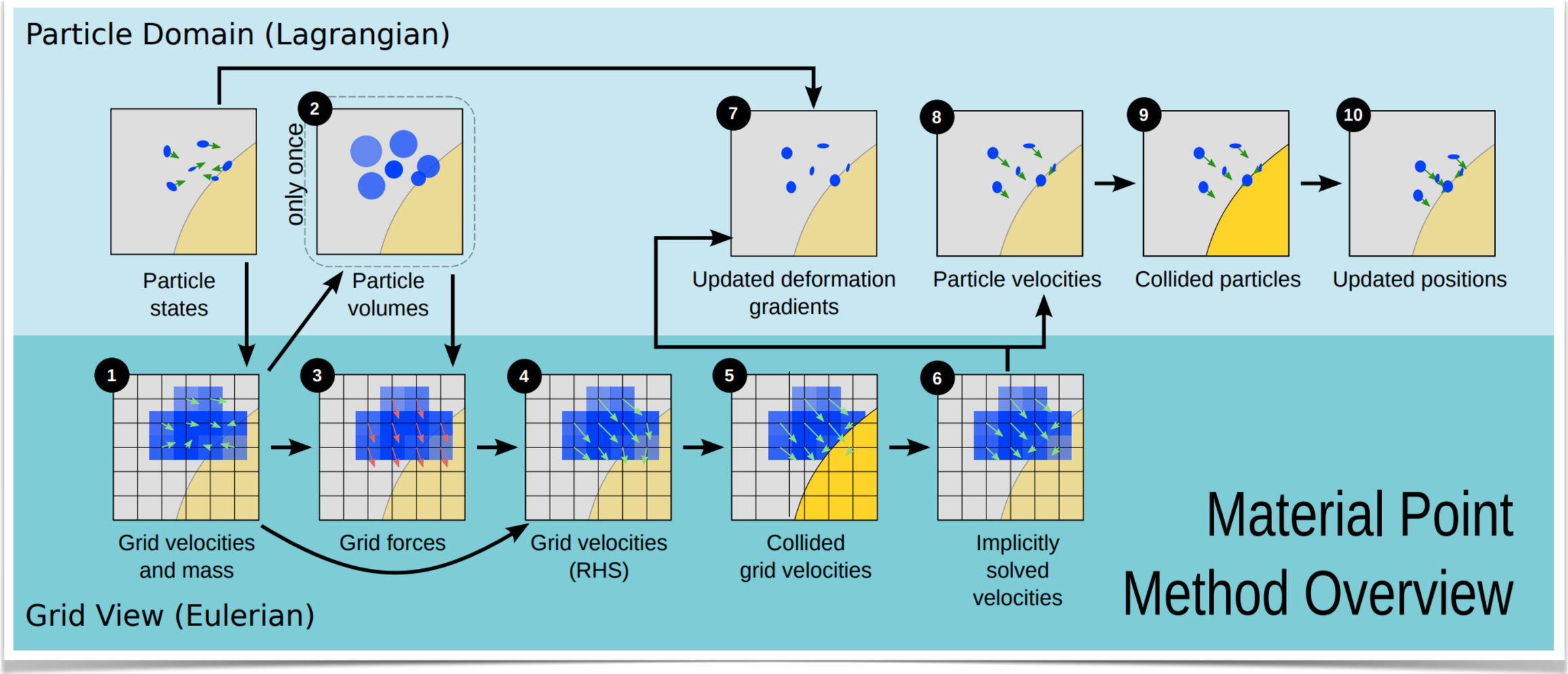
Figure 9: Walking character. *A character stepping through snow produces an interesting structure.* ©*Disney.*

[Stomakhin et al. 2013, A material point method for snow simulation]

The Material Point Method (MPM)

- **MPM is popular because of …**
  - Automatic coupling of different materials (liquids, solids, granular materials etc.)
  - Automatic (self-)collision handling
  - Automatic fracture
  - Capable of simulating large deformations
- **Hybrid Lagrangian-Eulerian: both a grid and particles are used**
  - An Eulerian grid is used for collision handling and momentum update
  - Lagrangian particles are used for state tracking such as advection and deformation

The Material Point Method (MPM)



Stomaching et al., A material point method for snow simulation, SIGGRAPH 2013

Classical MPM in graphics

1. **Rasterize particle data to the grid.** The first step is to transfer mass from particles to the grid. The mass is transferred using the weighting functions $m_i^n = \sum_p m_p w_{ip}^n$. Velocity also should be transferred to the grid, but weighting with w_{ip}^n does not result in momentum conservation. Instead, we use normalized weights for velocity $\boldsymbol{v}_i^n = \sum_p \boldsymbol{v}_p^n m_p w_{ip}^n / m_i^n$. This contrasts with most incompressible FLIP implementations.
2. **Compute particle volumes and densities.** *First timestep only.* Our force discretization requires a notion of a particle's volume in the initial configuration. We can estimate a cell's density as m_i^0/h^3, which we can weight back to the particle as $\rho_p^0 = \sum_i m_i^0 w_{ip}^0 / h^3$. We can now estimate a particle's volume as $V_p^0 = m_p/\rho_p^0$.
3. **Compute grid forces** using equation (6) with $\hat{\boldsymbol{x}}_i = \boldsymbol{x}_i$.
4. **Update velocities on grid** to $\boldsymbol{v}_i^\star$ using equation (10).
5. **Grid-based body collisions** on $\boldsymbol{v}_i^\star$ as described in Section 8.
6. **Solve the linear system** in equation (9) for semi-implicit integration. For explicit time integration, simply let $\boldsymbol{v}_i^{n+1} = \boldsymbol{v}_i^\star$.
7. **Update deformation gradient.** The deformation gradient for each particle is updated as $\boldsymbol{F}_p^{n+1} = (\boldsymbol{I} + \Delta t \nabla \boldsymbol{v}_p^{n+1}) \boldsymbol{F}_p^n$, where we have computed $\nabla \boldsymbol{v}_p^{n+1} = \sum_i \boldsymbol{v}_i^{n+1} (\nabla w_{ip}^n)^T$. Section 7 gives a detailed description of the update rule for elastic and plastic parts of $\boldsymbol{F}$.
8. **Update particle velocities.** Our new particle velocities are $\boldsymbol{v}_p^{n+1} = (1-\alpha)\boldsymbol{v}_{\text{PIC}p}^{n+1} + \alpha \boldsymbol{v}_{\text{FLIP}p}^{n+1}$, where the PIC part is $\boldsymbol{v}_{\text{PIC}p}^{n+1} = \sum_i \boldsymbol{v}_i^{n+1} w_{ip}^n$ and the FLIP part is $\boldsymbol{v}_{\text{FLIP}p}^{n+1} = \boldsymbol{v}_p^n + \sum_i (\boldsymbol{v}_i^{n+1} - \boldsymbol{v}_i^n) w_{ip}^n$. We typically used $\alpha = 0.95$.
9. **Particle-based body collisions** on $\boldsymbol{v}_p^{n+1}$ as detailed in Section 8.
10. **Update particle positions** using $\boldsymbol{x}_p^{n+1} = \boldsymbol{x}_p^n + \Delta t \boldsymbol{v}_p^{n+1}$.

Stomaching et al., A material point method for snow simulation, SIGGRAPH 2013
https://www.math.ucla.edu/~jteran/papers/SSCTS13.pdf

Moving Least Squares MPM

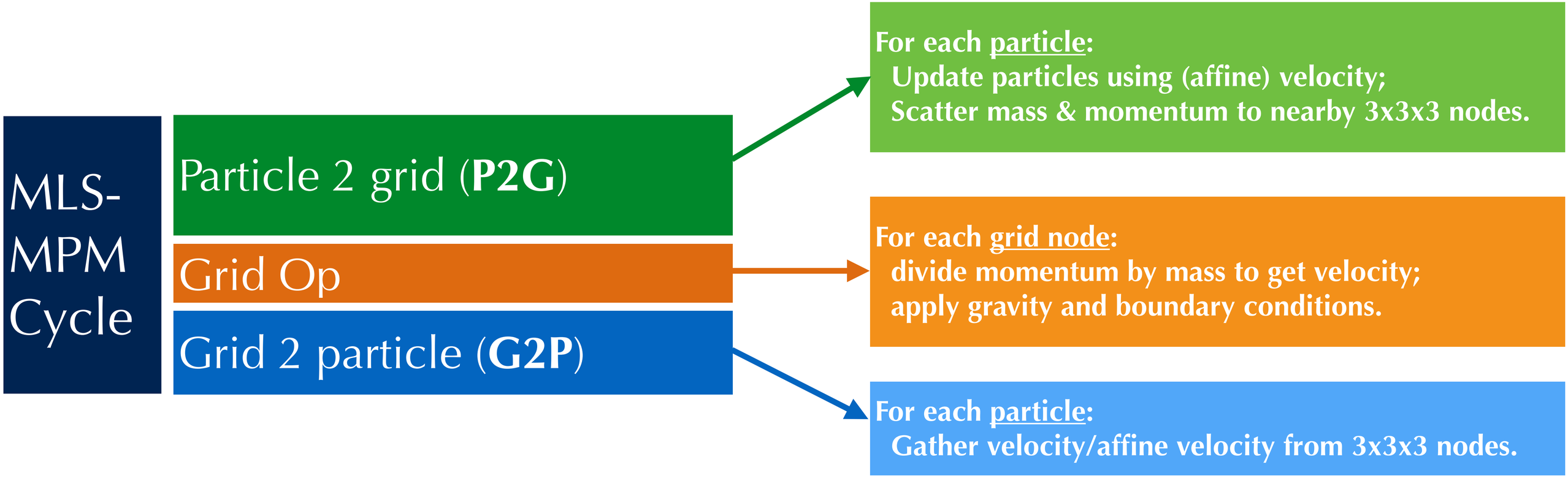
A Moving Least Squares Material Point Method with Displacement Discontinuity and Two-Way Rigid Body Coupling
Hu et al, SIGGRAPH 2018 http://taichi.graphics/wp-content/uploads/2019/03/mls-mpm-cpic.pdf

- **Based on APIC**
- **Halves the required FLOPs (2x faster!)**
- **Much easier to implement than traditional MPM**
  - 88 lines of code using Taichi
- **Demos:**
  - ti example mpm88
  - ti example mpm99
  - ti example mpm128

Implementing MLS-MPM

The MLS-MPM Simulation Cycle

Bandwidth-saving version in Hu2019Taichi



MLS-MPM in 88 Lines of Taichi Code

```
1   import taichi as ti
2   import random
3   ti.init(arch=ti.gpu)
4
5   dim = 2
6   n_particles = 8192
7   n_grid = 128
8   dx = 1 / n_grid
9   inv_dx = 1 / dx
10  dt = 2.0e-4
11  p_vol = (dx * 0.5)**2
12  p_rho = 1
13  p_mass = p_vol * p_rho
14  E = 400
15
16  x = ti.Vector(dim, dt=ti.f32, shape=n_particles)
17  v = ti.Vector(dim, dt=ti.f32, shape=n_particles)
18  C = ti.Matrix(dim, dim, dt=ti.f32, shape=n_particles)
19  J = ti.var(dt=ti.f32, shape=n_particles)
20  grid_v = ti.Vector(dim, dt=ti.f32, shape=(n_grid, n_grid))
21  grid_m = ti.var(dt=ti.f32, shape=(n_grid, n_grid))
```

MLS-MPM in 88 Lines of Taichi Code

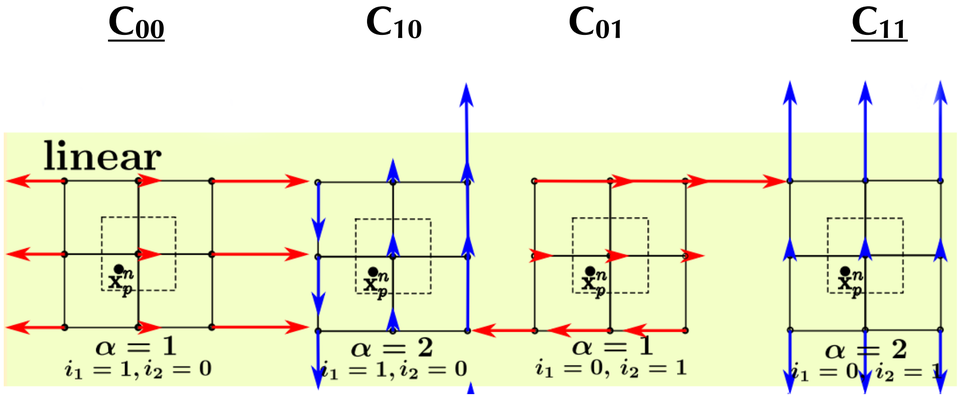
```
23  @ti.kernel
24  def substep():
25    for p in x:
26      base = (x[p] * inv_dx - 0.5).cast(int)
27      fx = x[p] * inv_dx - base.cast(float)
28      w = [0.5 * (1.5 - fx) ** 2, 0.75 - (fx - 1) ** 2, 0.5 * (fx - 0.5) ** 2]
29      stress = -dt * p_vol * (J[p] - 1) * 4 * inv_dx * inv_dx * E
30      affine = ti.Matrix([[stress, 0], [0, stress]]) + p_mass * C[p]
31      for i in ti.static(range(3)):
32        for j in ti.static(range(3)):
33          offset = ti.Vector([i, j])
34          dpos = (offset.cast(float) - fx) * dx
35          weight = w[i][0] * w[j][1]
36          grid_v[base + offset] += weight * (p_mass * v[p] + affine @ dpos)
37          grid_m[base + offset] += weight * p_mass
```

MLS-MPM in 88 Lines of Taichi Code

```
39      for i, j in grid_m:
40        if grid_m[i, j] > 0:
41          bound = 3
42          inv_m = 1 / grid_m[i, j]
43          grid_v[i, j] = inv_m * grid_v[i, j]
44          grid_v[i, j][1] -= dt * 9.8
45          if i < bound and grid_v[i, j][0] < 0:
46            grid_v[i, j][0] = 0
47          if i > n_grid - bound and grid_v[i, j][0] > 0:
48            grid_v[i, j][0] = 0
49          if j < bound and grid_v[i, j][1] < 0:
50            grid_v[i, j][1] = 0
51          if j > n_grid - bound and grid_v[i, j][1] > 0:
52            grid_v[i, j][1] = 0
```

MLS-MPM in 88 Lines of Taichi Code

```
54    for p in x:
55      base = (x[p] * inv_dx - 0.5).cast(int)
56      fx = x[p] * inv_dx - base.cast(float)
57      w = [
58          0.5 * (1.5 - fx) ** 2, 0.75 - (fx - 1.0) ** 2, 0.5 * (fx - 0.5) ** 2
59      ]
60      new_v = ti.Vector.zero(ti.f32, 2)
61      new_C = ti.Matrix.zero(ti.f32, 2, 2)
62      for i in ti.static(range(3)):
63        for j in ti.static(range(3)):
64          dpos = ti.Vector([i, j]).cast(float) - fx
65          g_v = grid_v[base + ti.Vector([i, j])]
66          weight = w[i][0] * w[j][1]
67          new_v += weight * g_v
68          new_C += 4 * weight * g_v.outer_product(dpos) * inv_dx
69      v[p] = new_v
70      x[p] += dt * v[p]
71      J[p] *= 1 + dt * new_C.trace()
72      C[p] = new_C
```



MLS-MPM in 88 Lines of Taichi Code

```
74  for i in range(n_particles):
75    x[i] = [random.random() * 0.4 + 0.2, random.random() * 0.4 + 0.2]
76    v[i] = [0, -1]
77    J[i] = 1
78
79  gui = ti.GUI("MPM88", (512, 512))
80  for frame in range(20000):
81    for s in range(50):
82      grid_v.fill([0, 0])
83      grid_m.fill(0)
84      substep()
85    gui.clear(0x112F41)
86    gui.circles(x.to_numpy(), radius=1.5, color=0x068587)
87    gui.show()
```

Recap

- **Hybrid Eulerian-Lagrangian schemes**
  - Use particles to track material (position, velocity, deformation)
  - Use grids to compute force fields (Chorin projection/Cauchy stress)
- **Reducing dissipation of PIC:**
  - APIC/PolyPIC (more modes)
  - FLIP (gather delta)
- **Material Point Method**
  - Use particles to store deformation information

MPM Courses/Paper list

- **The material point method for simulating continuum materials**
  - Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, and Andrew Selle
  - In ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)
- **On hybrid Lagrangian-Eulerian simulation methods: practical notes and high-performance aspects**
  - Yuanming Hu, Xinxin Zhang, Ming Gao, and Chenfanfu Jiang
  - In ACM SIGGRAPH 2019 Courses (SIGGRAPH '19)
- **MPM in computer graphics** by Chenfanfu Jiang

The end

Questions are welcome!