CIS 563 Final Project Report

Crystal Lee

1 Introduction/Project Proposal

This semester, I implemented Professor Jiang's paper, "The Material Point Method for Simulating Continuum Materials." The paper describes a general approach to simulating various materials by calculating internal collisions using Lagrangian calculations on the particles and Eulerian calculations on the background grid of the simulation with relative computation simplicity. MPM allows us to simulate various different viscous solids and fluids, such as snow, toothpaste, and lava, and it has been used extensively in the graphics industry.

Additionally, I implemented a Poisson Sampler to randomly sample from a mesh and create the particles that will be used in the MPM algorithm.

I used Qt Creator for this project. I also used the Eigen library to allow for more efficient linear algebra processes and OpenGL for visualization purposes.

I worked alone on this project.

2 Introduction to The Material Point Method

The Material Point Method allows us to properly simulate large deformations in a much more accurate way. It involves a combination of Lagrangian calculations on the particles themselves and Eulerian calculations on the background grid of the simulation. This hybrid Eulerian/Lagrangian MPM allows us to integrate implicitly over a regular Cartesian grid to automate features such as self-collision and fracture. This results in physically accurate deformations with relative computational simplicity. The method has been verified in various papers for simulating various materials such as elastic objects, snow, and lava. It has also been integrated into the production pipeline at the Walt Disney Animation Studios and features in their movies *Frozen*, *Big Hero 6*, and *Zootopia*.

The Material Point Method is a generalization of the Particle in Cell (PIC) techniques, particularly the Fluid Implicit Particle Method (FLIP), to solid mechanics. FLIP has been known in the graphics community as a useful liquid simulation for a while. While these techniques have proven powerful, they also suffer from some well-known limitations. The original PIC method has stability, while FLIP suffers from noise and instability. However, the original PIC suffers from dissipation from a loss of information when transferring between grid and particle representations, which FLIP was designed to remove. MPM retains the stability of the original PIC while removing its dissipation using the Affine Particle-In-Cell (APIC) method.

This Affine Particle-In-Cell (APIC) method is an important part of transferring information between the particles and the grid. APIC controls noise by keeping the pure filter property of PIC but minimizes information loss by enriching each particle with a $3x3$ matrix giving locally affined, rather than locally constant, description of the flow. This not only removes the dissipation, but also allows for exact conservation of angular momentum across the transfers between particles and the grid. Furthermore, the method is applicable to both incompressible liquids and MPM simulations.

As a hybrid Lagrangian/Eulerian approach, MPM has several advantages compared to traditional Lagrandian methods (such as FEM solids) and Eulerian methods (such as grid-based fluids). As with Lagragian FEM, MPM can be derived from the weak form of conservation of momentum, allowing for physically accurate discretization of physical laws. Boundary conditions, solid wall collisions, and external forces can be easily applied on the grid and particles. Automatic self-collision is implicitly handled because particle movements are interpolated from undistortable nodal movement on a grid. MPM allows for automatic splitting and merging behaviors because of particle based material representation, which is very useful for fluids and granular materials. It also allows for easy automatic multi-material and multiphase coupling by fiving particles different material properties or constitutive models. Lastly, MPM can be used to simulate mesh-based Lagrangian forces without losing its advantages. This allows the coupling of point-based objects and mesh-based objects with a single solve on the grid that handles collisions automatically. Overall, MPM allows for the simulation of a much wider range of materials than a purely Langrangian or Eulerian method.

MPM also introduces a novel method for heat transport and melting/solidifying materials. This brings a wider range of material behaviors into reach of the already versatile MPM. Because these heat transport properties are not particularly relevant to what I want for my simulation, I will not be delving too much into them. But it should be noted that extending the material point method in this way requires several technical novelties, including a dilational/deviatoric splitting of the constitutive model. Additionally, showing an implicit treatment of the Eulerian evolution of the dilational part can be used to simulate arbitrarily incompressible materials.

3 Theory behind the MPM Algorithm

This is a summation of the important concepts that are helpful for understanding the MPM algorithm.

The crux of the MPM algorithm is transferring the attributes of particles to a grid, performing operations on that grid to manipulate the particles' attributes, and then transferring the updated information back to the particles. The particles used in MPM, despite the name, actually each represent a continuous piece of material, and together make the whole simulated material domain. For this reason, the material is referred to as a "continuum material".

As previously said, MPM is a hybrid of the Eulerian and Lagrangian approaches. In the Lagrangian view, velocity v and acceleration a are functions of the material configuration X and time t. The velocity is the first derivative of the $\phi(X, t)$ function, and acceleration is the second derivative. Physically, this means we are measuring these attributes on a fixed particle, which has its mass and occupies some volume since the beginning. Soon, we will encounter the Eulerian view, where we are sitting at a fixed position in space and measuring the velocity of whichever particle that is passing by that position. In this way, the particles utilize Lagrangian calculations, while the background grid of MPM utilizes Eulerian.

We can find the deformation of any material coordinate X at any time t through the Jacobian of the deformation map ϕ. This Jacobian is useful for many reasons, e.g. the physics of elasticity is naturally described in terms of this Jacobian. The Jacobian is also described as a coordinate's **deformation gradient** F.

$$F(X,t) = \frac{\partial \phi}{\partial X}(X,t) = \frac{\partial x}{\partial X}(X,t)$$

The deformation gradient F is discretely a small 2×2 or 3×3 matrix. Intuitively, the deformation gradient represent how deformed a material is locally. The determinant of F (commonly denoted with J) is also often useful because it characterizes infinitesimal volume change. It is commonly denoted with $J = \det(F)$. $J > 1$ denotes volume increase and $J < 1$ denotes volume decrease.

Another important component of the continuum material is its **stress**. Stress is related to the deformation gradient, and is discretely a small tensor (matrix) at each evaluated point. There are several constitutive models used to relate the deformation gradient to the stress, and each is used to govern the material's reponses under deformations.

For traditional solids, we can express the stress using first Piola-Kirchoff stress. The first Piola-Kirchioff stress P can be derived from a strain energy density function $\Psi(F)$, where $P = \frac{\partial \Psi}{\partial F}$. $\Psi(F)$ is an elastic energy density function designed to penalize non-rigid F. P has the same dimensions as F.

We can also relate P to the Cauchy stress σ.

$$\sigma = \frac{1}{J} P F^T = \frac{1}{\det(F)} \frac{\partial \psi}{\partial F} F^T$$

For hyperelastic materials, we can also use the Neo-Hookean model of stress. The energy density function for this model is

$$P = \mu(F - F^{-T}) + \mu \log(J) F^{-T}$$

where d denotes the dimension and is either 2 or 3, and μ and λ are related to Young's modulus E and Poisson ratio υ via

$$\mu = \frac{E}{2(1+\upsilon}, \quad \lambda = \frac{E\upsilon}{(1+\upsilon)(1-2\upsilon)}$$

A very valuable tool in relating the stress P to the deformation gradient F is the Singular Value Decomposition (SVD) calculation. $F = U\Sigma V^T$ is the "Polar SVD" form that is widely favored in the graphics/mechanics field. U and V are rolations, and the polar decomposition $F = RS$ can be reconstructed via $R = UV^T$ and $S = V\Sigma V^T$, where R is the closest rotation to F and S is symmetric.

The SVD is used in the fixed corrotated constitutive model of stress. Assuming the polar SVD $F = U\Sigma V^T$, we find that

$$P(F) = \frac{\partial \psi}{\partial F}(F) = 2\mu(F - R) + \lambda(J-1)JF^{-T}$$

The deformation gradient can be slightly refactored to represent both the elasticity and the plasticity deformation history of the material:

$$F = F_E F_P$$

The plastic part, F_P, represent the portion of the material's history that has been forgotten, i.e. a metal rod that has been bent into a coiled spring and has forgotten that it used to be straight. The elastic deformation, stored in F_E, represents deformations that are remembered, such as the spring being slightly compressed and exerting stress to restore itself to its coiled shape. Together, the F_E and the F_P components represent the full history of the metal rod.

As an implementation-specific aside - thinking of the consistency of yogurt, it has little to no elasticity but it does have a strong degree of plasticity; in other words, yogurt does not remember any deformations it goes through and does not try to restore itself to any previous state. Thus, yogurt is not a hyperelastic material and we want to include the plasticity component of the deformation gradient.

During the MPM algorithm, we will calculate the new deformation gradient F^{n+1} from the previous deformation F^n for the material, and from the new deformations we can find the elastic and plastic portions. We begin by assuming that all new deformation introduced from F^n to F^{n+1} was elastic. Thus, given a new F^{n+1}, we can decompose it as

$$F^{n+1} = \tilde{F}_E^{n+1} F_P^n$$

And we define $\tilde{F}_E^{n+1}$ as

$$\tilde{F}_E^{n+1} = F^{n+1}(F_P^n)^{-1}$$

Now we can use some small constraints θ_c and θ_s to manipulate how much of the new deformation was elastic or plastic. We enforce that the singular values of $\tilde{F}_E^{n+1}$ satisfy the constraint of lying in $[1-\theta_c, 1+\theta_2]$ and clamp their values to be in that range. Now, assuming that the SVD of $\tilde{F}_E^{n+1}$ is

$$\tilde{F}_E^{n+1} = U_E^{n+1} \tilde{\Sigma}_E (V_E^{n+1})^T$$

We can define F_E^{n+1} from the clamped singular values Σ_E^{n+1} as

$$F_E^{n+1} = U_E^{n+1} \Sigma_E (V_E^{n+1})^T$$

And now we can determine the new F_P^{n+1} such that $F^{n+1} = F_E^{n+1} F_E^{n+1}$ as

$$F_P^{n+1} = (F_E^{n+1})^{-1} F^{n+1}$$

Another important concept is the interpolating functions used to relate the Lagrangian particles to the Eulerian grid. In both the particle-to-grid and the grid-to-particle transfer steps of the MPM algorithm, interpolation functions are required. In our algorithm, interpolation functions are defined over grid nodes.

We denote the interpolation function at grid node i with $N_i(x)$, where i is the index for a grid node and can be in the form $i = (x, y)$ for 2D grids and $i = (x, y, z)$ for 3D. When $N_i(x)$ is evaluated at a particle location x_p, the particle is assigned a weight $N_i(x_p) = w_{ip}$. Intuitively, we are associating each particle p and grid node i a weight w_{ip} which determines how strongly the particle and node interact. IF the particle and grid node are close together, the weight should be large. If the particle and node are farther apart, the weight should be small.

Various types of interpolation functions can be used to calculate grid weights, including linear, cubic, and quadratic. In my implementation, quadratic interpolation is used. The piecewise quadratic interpolation function is defined as follows:

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

The interpolation along each axis is calculated using the following formulas:

$$a = N(\frac{1}{h}(x_p - x_i)), \quad b = N(\frac{1}{h}(y_p - y_i)), \quad c = N(\frac{1}{h}(z_p - z_i))$$

And the weight and the weight gradients are calculated as such:

$$w_{ip} = abc$$

$$\Delta w_{ip} = [(\frac{1}{h} * N'(x_p - x_i), b, c), (a, \frac{1}{h} * N'(y_p - y_i), c), (a, b, \frac{1}{h} * N'(z_p - z_i)]$$

Where i is the grid index, x_p is the evaluation position, h is the grid spacing, and x_i is the grid node position. Notation-wise, $w_{ip} = N_i(x)$ and $\nabla w_{ip} = \nabla N_i(x_p)$. $N'(x)$ is the derivative of $N(x)$.

The last piece of theory I will note is collision detection. We process collisions immediately after forces are applied to grid velocities. The collision treatments can be applied once more to particle velocities v_p^{n+1} just before updating positions to account for minor discrepancies between particle and grid velocities due to interpolation. All collisions calculated this way are inelastic.

The research papers outline a method of collision detection - we represent collision objects as level sets, which makes collision detection ($\phi \leq 0$) trivial. In case of a collision the local normal $n = \nabla\phi$ and object velocity v_{co} are computed. The particle/grid velocity v is transformed into the reference frame of the collision object, $v_{\text{rel}} = v - v_{\text{co}}$. If the bodies are separating ($v_n = v_{\text{rel}}\dot{n} \geq 0$), then no collision is applied. In this case, we let $v_t = v_{\text{rel}} - nv_n$ be the tangential portion of the relative velocity. If a sticking impulse is required ($||v_t|| \leq -\mu v_n$), then we simply let $v'_{\text{rel}} = 0$. Otherwise, we apply dynamic friction, and $v'_{\text{rel}} = v_t + \mu v_n v_t / ||v_t||$, where μ is the coefficient of friction. Finally, we transform the collided relative velocity back into world coordinates with $v' = v'_{\text{rel}} + v_{\text{co}}$.
For this method of detecting collisions, we can use two types of collision objects: rigid and deforming. In the rigid case, we store a stationary level set and a rigid transform, which we can use to compute ϕ, n, and v_{co} at any point. In the deforming case, we can load level set key frames and interpolate them.

In my implementation, I instead opted to implement collisions in a different way. All collisions in my implementation are considered to be objects outside the simulation, or "sticky" collisions, namely the walls and the ground. To accomplish this, I simply set $v_i^{n+1} = 0$ if x_i^i is in a collision object.

4 Step-by-step overview of the MPM Algorithm

The full algorithm of MPM assumes the particles to have already been initialized with a mass m_p, a volume V_p^0, an initial position x_p, an initial velocity v_p, an affine matrix B_p, and their material-related parameters.

1. **Particle to grid transfer (P2G)**
   The first step involves a particle-to-grid transfer. This step computes the grid masses and momentums using the APIC formula.
   $$m_i = \sum_p w_{ip} m_p$$
   $$m_i v_i = \sum_p w_{ip} m_p (v_p + B_p (D_p)^{-1} (x_i - x_p))$$
   where D_p is given by
   $$D_p = \sum_i w_{ip} (x_i - x_p)(x_i - x_p)^T$$
   and is derived by preserving affine motion during the transfers.

   Additionally, in the case of the quadratic or cubic interpolation stencils, the D_p matrix takes a surprisingly simple form. In the quadratic case, $D_p = \frac{1}{4}\Delta x^2 I$ and in the cubic case, $D_p = \frac{1}{3}\Delta x^2 I$.

2. **Compute grid velocities**
   The second step is computing the grid velocities, $v_i = \frac{m_i v_i}{m_i}$. For nodes with mass equal to 0, m_i and v_i are manually reset to 0.

3. **Identify grid degree of freedoms**
   Next, we identify the grid degree of freedoms. This step is mainly important for implementation efficiency. All grid nodes with nonzero masses are labeled with their degree of freedom. All other nodes will remain static and are not considered to be part of the solver unknowns.

4. **Compute explicit grid forces**
   Next, we compute the explicit grid forces f_i^n using the following equation.
   $$f_i^n = f_i(x_i^n) = -\sum_p V_p^0 (P_p (F_p^n)^T \nabla w_{ip}^n)$$
   This equation fully depends on the existing particle/grid weights and particle attributes. The force can also be written using the Cauchy stress if we don't have the energy density.
   $$f_i^n = f_i(x_i^n) = -\sum_p V_p^n \sigma_p^n \nabla w_{ip'}^n$$
   In this implementation, the fixed-corrotated model of stress is used, although the Neo-Hookean model has also been implemented and can be used as a substitute.

5. **Grid velocity update**
   We then update the grid velocity using the following equation.
   $$v_i^{n+1} = v_i^n + \Delta t f_i(x_i^n)/m_i$$
   This step takes the boundary conditions and collision objects into account. In the case of explicit integration, each nodal velocity can be independently set to the desired value.
   We use the "sticky" collision method to create walls and obstacles for the particles. If x_i^i is in a collision object, then $v_i^{n+1} = 0$.

6. **Update particle deformation gradient**
   Next, we update the particle deformation gradient F.

$$F_p^{n+1} = (I + \Delta t \sum_i v_i^{n+1} (\nabla w_{ip}^n)^T) F_p^n$$

   Notice that symplectic Euler time integration on the grid implies that

$$x_i^{n+1} = x_i^n + \Delta t v_i^{n+1}$$

   So our F update equation can also be written as

$$F_p(x) = (I + \sum_i (x_i - x_i^n)(\nabla w_{ip}^n)^T) F_p^n$$

   As outlined above, we can then calculate the elasticity and plasticity portions of this new deformation for each particle.

7. **Grid to particle transfer (G2P)**
   Next, we perform the grid to particle transfer. This step computes new particle velocities v_p^{n+1} and affine matrices B_p^{n+1}.

$$v_p = \sum_i w_{ip} v_i$$

$$B_p = \sum_i w_{ip} v_i (x_i - x_p)^T$$

   The D_p component is not required in this transfer because it takes on a surprisingly simple form in the cases of the interpolation stencils commonly used for MPM. For these stencils, multiplying by $(D_p)^{-1}$ results in a constant scaling factor.

8. **Particle advection**
   Finally, particles are advected with their new velocities:

$$x_p^{n+1} = x_p^n + \Delta t v_p^{n+1}$$

   Note that this is only true when APIC or PIC is used. In a FLIP or FLIP-PIC blending transfer scene, the following equation should be used instead.

$$x_p^{n+1} = \sum_i x_i^{n+1} w_{ip}$$

   Also in this step, we apply collision updates again to account for discrepancies.

This concludes the full algorithm of MPM.

5 Code Overview

The pipeline of my code is mostly based on three important classes, the `ParticleGrid`, the `Particle`, and the `PoissonSampler`.

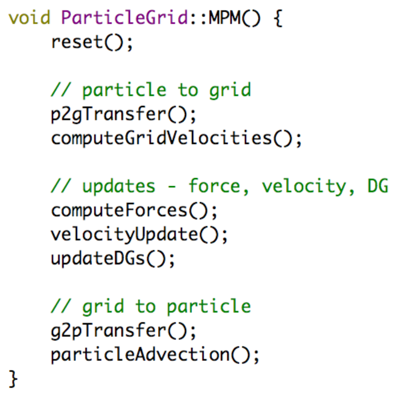
The `ParticleGrid` is the 3D grid used in the MPM algorithm and runs the MPM algorithm in its entirety. It contains the grid masses, momentums, forces, and velocities, and also contains functionality for calculating particle weights and weight gradients. The grid spans the entire scene, but there is a "buffer" of 2 grid cells on all sides which are expected to not hold any particles. This is because the various grid calculations are solved over a span of 4x4x4 grid cells around a particle particle. The buffer is required to prevent indexing out-of-bounds errors while performing our grid solves.

Each `Particle` contain its position in world space, mass, volume, velocity, affine matrix, stress, grid weight, grid weight gradient, and the elasticity and plasticity components of its deformation gradient.

These fields are updated in each time step of the MPM algorithm. The class also has functionality for updating its deformation gradient components given new deformation information, using the polar SVD method described previously.

The `PoissonSampler` class is, of course, meant to create a number of evenly spaced and randomly sampled points from a 3d mesh, using a background grid to calculate distances between samples. The amount of particles the sampler generates can be manipulated by changing the cell size of the background grid. As the particles are created when the Poisson Sampler runs its sampling algorithm, the sampler also contains the list of particles in our scene. The `ParticleGrid` is constructed using the sampler and has access to the particles through it. The sampler is also used to create the OpenGL buffer information of the particles and create them.

My program begins by loading a 3D cube mesh and sampling the mesh to find a large number of particles. These particles are then simulated using the particle grid, which contains all the functionality for the MPM algorithm, as seen in the following code screenshot:



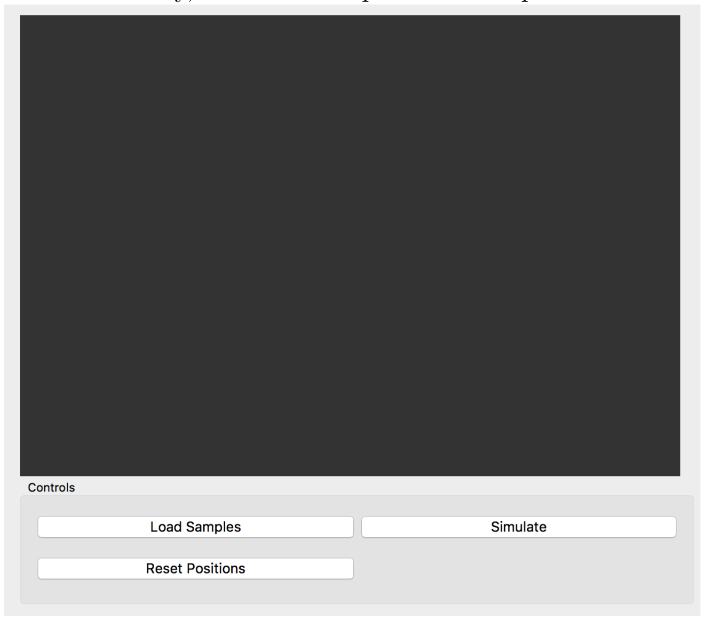
```
void ParticleGrid::MPM() {
    reset();

    // particle to grid
    p2gTransfer();
    computeGridVelocities();

    // updates - force, velocity, DG
    computeForces();
    velocityUpdate();
    updateDGs();

    // grid to particle
    g2pTransfer();
    particleAdvection();
}
```

Additionally, these various processes are performed through a simple GUI.



6 Challenges

As this was a solo project, I ran into many challenges over the course of the semester. A great many of these challenges were because I had difficulty wrapping my head around the high-level concepts presented in the papers I was referencing.

One very persistent challenge I encountered was simply how to handle my indexing as I transferred information from particle-to-grid and grid-to-particle. I did not implement the "2 grid cell buffer" until partway through my implementation, so for a long time I struggled with the grid solve operations due to not being able to properly access the grid cells I needed to. This led to errors such as my interpolation functions not returning reasonable values and my weights thus being wrong, which would throw every other operation off. This would lead to unpredictable behavior and the particles would fly up and down at high speeds or randomly stop and get stuck. After implementing the buffer for the grid, it became easier to debug grid indexing issues.

Another challenge came in calculating new particle deformation gradients and stresses. This was mainly due to the limitations of the libraries I was using. For most of the MPM algorithm, I am using the glm graphics library; unfortunately, the glm library does not contain any convenient ways to calculate SVD values. In order to use SVD, I decided to use the Eigen library, which performs slower than the glm library but does support SVD calculations. Using both libraries entailed switching from one to another and back within my deformation gradient calculation, and this required some tricky conversions. Additionally, when I did not do these conversions correctly, I would get NaN values for my deformation gradient components, and that would then cause NaN values in the rest of the algorithm. To debug these errors, I had to be extra careful in my deformation gradient and stress updates.

Another large challenge I had was memory and performance issues. For a long time, I was unable to render a lot of particles. I couldn't even render 2000 particles without my program crashing. I theorized that there was a memory leak somewhere, but I had a difficult time finding it. However, I eventually found that I was accidentally allocating twice as much memory for my particles. I fixed that, and just to be safe, I decided to store the particle data on the stack instead of the heap. After I made these changes, I could render many many more particles.

7 Conclusion

In conclusion, I set out to implement the MPM algorithm for simulating continuum materials, and I believe I have accomplished this goal. I ran into a lot of difficulty due to working solo and having a hard time grasping the papers, but I now feel that I understand the algorithm and I ended up with a good physically-based simulation.

I originally wanted to end up with a yogurt simulation just as an arbitrary material to simulate, but because I closely referenced papers that were for a snow sim, my simulation is instead closer to snow. If I continued to work on this project, I would like to refine the material to be yogurt as I initially wanted.

Another aspect where I want to improve my simulation would be making a more elegant collision detection system, a la the approach I outlined in the theory section. As it is, my implementation handles collision in a more simplistic way.

I would like to acknowledge everyone involved in this class: Professor Jiang and my TAs, Emily Vo, Josh Wolper, and Hannah Bollar. Thanks for a great semester :)

References

[1] Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, Andrew Selle, *The Material Point Method for Simulating Continuum Materials.* SIGGRAPH, Version 1, 2016.

[2] Chenfanfu Jiang, Craig Schroeder, Joseph Teran, *An Angular Momentum Conserving Affine-Particle-in-Cell Method.* ACM Trans Graph 34, 4, 2015.

[3] Chenfanfu Jiang, *The Material Point Method for the Physics-Based Simulation of Solids and Fluids.* PhD thesis, University of California, Los Angeles.

[4] Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, Andrew Selle, *The affine particle-in-cell method.* ACM Trans Graph, 34(4), 2015.

[5] Daniel Ram, Theodore Gast, Chenfanfu Jiang, Craig Schroeder, Alexey Stomakhin, Pirouz Kavehpour, *A Material Point Method for Viscoelastic Fluids, Foams and Sponges.* Proc ACM SIGGRAPH/Eurograph Symp Comp Anim, 2015.

[6] Alexey Stomakhin, Craig Schroeder, Chenfanfu Jiang, Lawrence Chai, Joseph Teran, Andrew Selle, *Augmented MPM for phase-change and varied materials.* ACM Transactions on Graphics (SIGGRAPH 2015 Proceedings), 2014.

[7] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, Andrew Selle, *A material point method for snow simulation.* ACM Trans Graph, 2013.

[8] Theodore Gast, Craig Schroeder, Alexey Stomakhin, Chenfanfu Jiang, Joseph Teran, *Optimization integrator for large time steps.* IEEE Trans Vis Comp Graph, 2015.

[9] Robert Bridson, *Fluid simulation for computer graphics.* Taylor and Francis, 2008.

[10] Robert Bridson, *Fast Poisson Disk Sampling in Arbitrary Dimensions.* ACM Transactions, 2013.