Material Point Method for the Implementation of Solids and Fluids

TeaMPM

Austin Eng, Hannah Bollar, Joseph Klinger, Youssef Victor

CIS 700 : Physics Based Material Simulation
Professor Chenfanfu Jiang

25 December 2017

Abstract

With the advent of the recent hybridized Material Point Method, properly simulating large deformations has become much more accurate. This Eulerian/Lagrangian technique involves integrating implicitly over a simple Cartesian grid where computations such as collisions, fracture, and basic deformation occur. The method has been leveraged in several other papers on material simulation as well as at Disney Animation Studios for use in movies such as *Big Hero Six*. In the project discussed below, we discuss our implementation of this MPM technique alongside the mathematical manipulation of elasto-plastic materials. Ultimately we successfully and accurately replicated complex deformable objects such as snow with an MPM backing.

Introduction

As realistic graphics-based visual effects become more and more prevalent in the film industry, animation and live-action alike, there becomes an increasing demand for simulations to follow suit. The MPM APIC-based method we implemented is an improvement on previous work used in such simulations. The Material Point Method aspect of our simulation involves a combination of Lagrangian calculations (on the particles themselves) and Eulerian calculations (on the background grid of the simulation). Do to its set up, it automatically calculates internal collisions, which is advantageous over normal Lagrangian methods. A solely Lagrangian based system requires a meshing that when manipulated into an extensive tangle deformation, would unnecessarily add a large increase in the number of self-collision calculations. In MPM, the Eulerian grid backing is used for calculating the update derivatives for the deformation thus removing the need for a fully created Lagrangian Mesh in the first place, removing the need for these collision checks. This malleability allows MPM to be used more extensively than previous methods for simulating artistically a larger variation of solids and fluids alike with much more computational simplicity (example: twisting cloth now becomes more easily solvable). A minor downside of this is that hyperelasticity may be dampened or not as effectively simulated; however, overall it's still largely beneficial in terms of self-collisions and material surface changes. In terms of usefulness, MPM has begun being used extensively in the graphics industry, notably in the recent Pixar films, *Zootopia, Frozen,* and *Moana*. In this write up we explain the origins of MPM and work related to it, the mathematical background for how the simulation works, our implementation details, and current limitations of the algorithm leading to possible future work in this topic of interest.

Related Work

Our simulations use the Material Point Method (MPM), a method which was introduced and saw widespread usage recently. MPM is a development of older work and is a generalization of the Particle in Cell (PIC) and the Fluid Implicit Particle (FLIP) methods. [Sulsky et al., 1995] Most notably, our method combines both methods by using an Eulerian grid with Lagrangian material particles to produce our simulations. This means that this method does not require any inherent mesh connectivity, which means we do not need to do any remeshing along the simulation.

There are also other methods that could be used such as the Finite Element Method (FEM), which is used for elasto-plastic solids. This however requires special computations for different parts of the simulation such as fracturing, and self-collision which may even need special solvers per each scenario. The benefit of MPM is that its hybrid nature allows this to be handled for us automatically. Like in particle methods such as Smoothed Particle Hydrodynamics (SPH), our method does not need explicit particle connectivity, and as such allows for easy change of the topology.

Hybrid methods have been used in many papers to demonstrate a broad range of possible materials such as snow [Stomakhin, Schroeder, Chai, Teran & Selle, 2013] and for Affine Particle In Cell (APIC) simulations. [Jiang, Schroeder, Selle, Teran, & Stomakhin, 2015]

Mathematical Background

Particle to Grid: Attribute Transfer and weight computation

$m_i = \Sigma_p m_p w_{ip}$

$v_i = \Sigma_p m_p \underline{v}_p w_{ip}$

For each node i, if $(m_i \neq 0)$ then $v_i = \frac{v_i}{m_i}$ else $v_i = 0$

With APIC this becomes:

$m_i^n v_i^n = \Sigma_p w_{ip}^n m_p (v_p^n + B_p^n (D_p^n)^{-1}(x_i - x_p^n))$

With the D tensor being defined as:

$D_p^n = \frac{1}{4}\Delta x^2 I$

Because we’re using quadratic interpolation, the B tensor is defined in the grid to particle interpolation later in this iteration of the simulation. This means that for the first iteration,, we just set the tensor to the zero matrix since there’s no initial angular momentum to account for.

The piecewise quadratic interpolation function we used for the grid weights:

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

Interpolation along each axis:

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

$w_{ip} = abc$

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

In the simulation, the N and N’ values are stored individual along the -x, x_0, and +x locations in relation to the 1+baseNode.x location (same for the y, z grid directions). Then to calculate the wip and grad wip values, the simulation just multiplies together the appropriate already calculated values. Thus, only needing us to store 9 numbers for w_{ip} and Δw_{ip} individually for 3-dimensions instead of 27 for each w_{ip} and Δw_{ip}.

On Grid: Update Attributes
Force update: $f_i = -\Sigma_p V_p^{0} \underline{P}_p \underline{F}_p^{nT} \Delta w_{ip}$ using weight gradient, Cauchy stress and initial volume.

Velocity update based on current force: $v_i^{n+1} = v_i^{n} + \Delta t \frac{f_i^n}{m_i}$

Collision check with objects outside the simulation ("sticky"):
If x_i^n is in a collision object then $v_i^{n+1} = 0$.

Grid to Particle: Attribute transfer
Defining the B tensor for APIC: $B_p^{n+1} = \Sigma_i w_{ip}^n v_i^{n+1} (x_i - x_p^n)^T$
velocity transfer: $v_p^{n+1} = \Sigma_i v_i^{n+1} w_{ip}$
position transfer: $x_p^{n+1} = \Sigma_i x_i^{n+1} w_{ip}$
force transfer: $F_p^{n+1} = F_p^{n} + (\Sigma_i \Delta t v_i^{n+1} (\Delta w_{ip})^T)) F_p^{n}$

Snow:
Force computations:
Stress update involves clamping the elasticity force F_E and putting the remainder into the plasticity force component, F_P. In mathspeak, that is effectively:
Clamp Sigma_p, from the SVD of F_p, to the range: $[1 - \theta_{c,} \ 1 + \theta_s]$
To compute F_Plastic, we take the svd of F_E_p:

$$F_{E_p}^{n+1} = U_p \Sigma_p V_p^T$$

And multiply the original force component by the inverse to solve for the plasticity force component:

$$F_{P_p}^{n+1} = U_p \Sigma_p V_p^T F_p^{n+1}$$

Mu and lambda updates for snow computations:

$$\lambda(F_p) = \lambda_0 e^{\xi(1-J_P)}$$

$$\mu(F_p) = \mu_0 e^{\xi(1-J_P)}$$

Then the simulation iterates through again.

Discretization/Implementation Details

Implementation of a multigrid to store particle attributes is represented by a templated class which recursively contains an instance of the template class with one fewer dimension. This makes the grid implementation to any number of dimensions. The predominant class used in the implementation is `Multigrid` which represents the grid as a flattened 1-D array. The implementation internally stores the stride of each dimension so that a 1-D index can efficiently be computed from a k-D index by taking the dot product between the k-D index and the strides. The `Multigrid` implementation provides C++11 style iterators to iterate over the elements or indices which use a more optimized index computation. Dot products do not need to be taken every step of the iteration because the code can just iterate over the 1-D indices instead.

Particles are stored in a `ParticleSet` which which is just a tuple of a bunch of `std::vector<T>`. Likewise, multiple `Multigrid`s are stored in an `AttributeGrid` as a tuple. `ParticleSet` and `AttributeGrid` share the same interface for defining the particle attributes stored. A template class is defined which specifies an enum naming the attribute, its type, and a default value. This makes a consistent API for getting particle or grid attributes without polluting the code with a new member variable for every attribute: `auto& particleVelocities = particles.Get<SimulationAttribute::Velocity>();`. These wrapper classes also wrap resizing so that all grids and particle attribute lists are sized equally.

Particle to grid and grid to particle transfer is simplified with by a templated helper function `IterateParticleKernel` which iterates over particles and their neighboring grid indices. The function takes in a functor as a template parameter which is applied to every particle and grid index pair. The template functor pattern allows for simplified programming while maintaining efficient compiler optimization of the code. Furthermore, grid indices are generated from a static multigrid which provides index iterating functions so the code is agnostic to the dimension of the simulation.

Naive iteration over a particle's kernel requires that code using `IterateParticleKernel` to check if the grid cell index is in bounds. Branches in the code make it less performant because the compiler can no longer automatically vectorize operations over sequential values. To improve performance, the grid is padded by two additional cells (or whatever the size of the kernel is), eliminating the need to check if a grid cell is in bounds. Particle to grid or grid to particle transfers simply use extra padded grid cells which can be masked to zero.

Division of Responsibilities

Austin - Implemented the initial background grid and was responsible for heavy project framework and the main infrastructure. Also involved in debugging and optimizations for CPU overall speed for each iteration.

Joe - Involved in the implementation of snow elastic and plastic calculations, grid collision checks, and final base renders.

Hannah - Implemented the particle to grid and grid to particle math-based transfers and also involved in the math understanding for cauchy-stress calculations for the MPM implementation and other math aspects of the project.

Youssef - Implemented Affine Particle In Cell (APIC) method to allow for angular momentum conservation in the MPM Simulation visuals. Also added particle exporting into .geo format and rendering using Houdini.

Limitations and Future Work

Our model currently simulates a variety of different materials and has generated lots of interesting simulation visuals, but there are things that we wish we had time to do. Our model for snow currently fractures too uniformly, we would like to add noise to the original particle positions using a simple perlin noise function to add more randomness and realism to our snow fractures. We would also like to have a more user-friendly interface so that people are able to use this method to generate different types of materials without having to understand the underlying equations and code.
In the future, we would also like to add simulations with objects interacting with static solid objects in the scene and not just the wall. We would also like to add simulations with particles with an initial velocity. These are not that hard but we could not get around to adding them in time.

Our code also is heavily reliant on compiler optimizations. Even though we use the Eigen library which does take advantage of CPU SIMD operations, the code is still single-threaded on our CPU. We would like to expand our model to use GPGPU computations to accelerate the simulation time and maybe add on-the-fly rendering to our scene, since most simulations require minimal rendering setup as there is not much in the scene other than the background, a light and the particles.

Works Cited

Ming Gao, Andre Pradhana Tampubolon, Chenfanfu Jiang, and Eftychios Sifakis. 2017. An Adaptive Generalized Interpolation Material Point Method for Simulating Elastoplastic Materials. ACM Trans. Graph. 36, 6, Article 223 (November 2017), 12 pages. DOI: 10.1145/3130800.3130879

Jiang, Chenfanfu. *The Material Point Method for the Physics-Based Simulation of Solids and Fluids*. 2015. University of California Los Angeles, PhD Dissertation.

Jiang, C., Schroeder, C., Selle, A., Teran, J., & Stomakhin, A. (2015). The affine particle-in-cell method. ACM Transactions on Graphics (TOG), 34(4), 51.
Sulsky, D., Zhou, S., and Schreyer, H. (1995). Application of a particle-in-cell method to solid mechanics. Comp Phys Comm, 87(1):236–252.