WAIT: Workshop on the Advances in Information Technology, Budapest, 2015

Simulating snow with the material point method

Tamás Umenhoffer[1]

[1] Department of Control Engineering and Informatics, Budapest University of Technology and Economics, Budapest, Hungary

Abstract

In this paper we describe a full pipeline for physically accurate dense snow simulation. Our system uses the material point method to simulate packs of snow as an elasto-plastic material. We focused on efficient implementation that fits into an animation and rendering pipeline typically used in motion picture production. We describe the material point method in details and list the tools that can help us to integrate the simulation into a rendering pipeline.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Animation

1. Introduction

Reproducing snow dynamics is a complex problem, as snow has various behaviour depending on whether we are talking about falling snowball, packing snow, footsteps in snow, or powder snow. We can notice that snow sometimes behaves as a fluid, but sometimes behave as an elastic solid. Unfortunately snow changes this behaviour continuously, so we can not choose only a fluid or a solid simulator for a given effect.

Figure 1: *Frames of an animation of a dropped snowball simulated with the material point method.*

Snow can be thought as a granular material, whose behaviour is mostly directed by inter-granular friction. Some graphics papers use simplified particle or rigid body systems [5, 2]. However it is hard to keep efficiency when increasing the detail, thus increasing the number of grains. This lead researchers to apply continuum models [11, 3, 6, 1]. One effective method is the fluid implicit particles (FLIP) [11], which was even used for human hair collision modelling[4]. FLIP is basically a fluid simulation method dealing with incompressible fluids. Fluid based methods model granular friction with viscosity. For some materials, and snow as well, compressibility should be modelled. The material point method (MPM)[9], was designed to extend FLIP to solid mechanics problems that require compressibility.

During the creation of the Disney movie titled Frozen, an MPM based method was introduced to simulate snow dynamics for computer animation[8]. This work inspired us to recreate the technique and integrate it to our preferred rendering pipeline. We did not change the original algorithm, but reimplement it and developed the components that could provide the necessary input and output to and from the animation software, the simulation software and the renderer. We used Autodesk Maya as the animation package as it is the most commonly used animation software in the movie industry.

The MPM simulator tracks material properties at particles, but uses an Eulerian background grid for computation, so we have to options to render out the simulation: using a volume renderer, or using a particle renderer. In contrast to the original paper we used a particle renderer, as increasing quality is more effective with increasing particle count than with increasing grid resolution. Increased 3D grid reso-

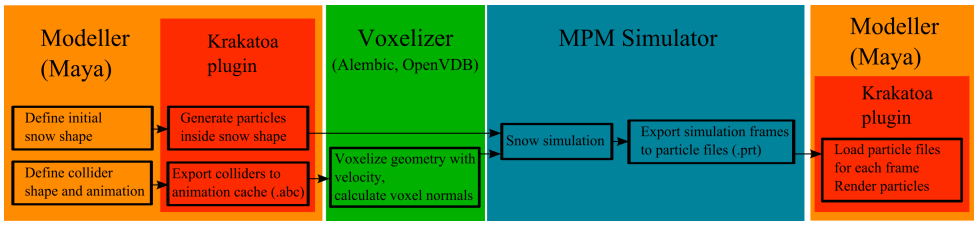

Figure 2: *Overview of our simulation pipeline.*

lutions have too high memory and rendering computational power needs. Maya also can not handle high particle counts effectively, but a renderer plugin exists, that is developed especially for this purpose. The Krakatoa renderer can handle millions of particles, can be used inside Maya (which is important for an instant visual feedback in Maya's 3D view), or as an external renderer.

This paper continues with the overview of our system, describing all the tools that are needed for a complete snow simulation and rendering. Then the main steps of the material point method are presented to demonstrate the operation of our simulator application. Finally our experiences are discussed.

2. System Overview

As we mentioned before our framework is based on Maya and the Krakatoa particle renderer. Figure 2 shows an overview of our simulation and rendering pipeline. For a complete simulation, taking into account the geometry of the virtual scene, the following problems should be solved.

First the initial snow particle positions should be given. For simple tests these positions can be filled procedurally, but for real scenes they are given by the artist. The easiest way is to model the volume of the snow and fill this volume with particles. The Krakatoa plugin for Maya has a feature to fill a closed polygonal geometry with particles using a user defined particle density. These particles can be exported into Krakatoa's particle file format and read by our simulator application.

The second task is to export the scene geometry with animations. This can be rather complicated as there are numerous different tools for animation in Maya. We can not prepare for all of them. However for such situations where only the final geometry is important and the concrete animation tools are not, we can cache the geometry in each frame and read it in the simulator. The computer animation community already created such a caching format and an open source library for reading and writing. It is called Alembic, and this format is becoming a standard in animation industry.

Our simulator performs collision detection not on a triangular mesh but on a volume grid, so we have to voxelize the exported geometry in each frame, and the simulator will read these voxel files instead of the geometry cache files. The problem with this voxelization is that we not only need to tell if a voxel is in the interior of a scene object, but we also need to tell its normal vector and velocity. If we have a binary volume storing empty and filled voxels the normals can be calculated relatively easily with central differences or higher order gradient methods [7].

Calculating voxel velocities is not straightforward. 3D optical flow could not be used efficiently here as the voxelized frames are binary, so most of the voxels will show no movement, and the rest will likely have aperture problem. However during reading the geometry cache and assuming that the topology does not change (which is true for most of the cases), we can pair the vertices of two adjacent frames and calculate their velocities with a simple subtraction. For each inner voxel we find the nearest triangle, calculate the projected barycentric coordinates of the voxel center and interpolate a the velocity from the vertex velocities.

Fortunately an open source library exists for voxelizing geometries and storing this sparse volumetric data in a hierarchical data structure. This library is called OpenVDB. OpenVDB also stores the nearest triangle index in each voxel, so we only have to implement the barycentric coordinate calculation and interpolation, and of course the normal vector calculation. Normals can also be calculated with interpolation, however if the geometry is complex compared to the grid resolution, significant noise can appear. Our normal calculation method is similar as in Thürmer et. all [10]. For each voxel we define its 26 neighbouring directions (d_k) and define the normal vector as:

$$N = \sum_k \sigma_k d_k$$

where σ_k is minus one if the neighbouring voxel is an inner voxel and zero otherwise. We separated the voxelizer from the simulator, thus the voxelizer stores the voxel grids in each frame in OpenVDB format, and the simulator takes these files as input.

The simulator calculates updated particle positions in each frame and stores these as Krakatoa particle files. These file sequences are loaded back into Maya as a particle animation and rendered with the Krakatoa renderer.

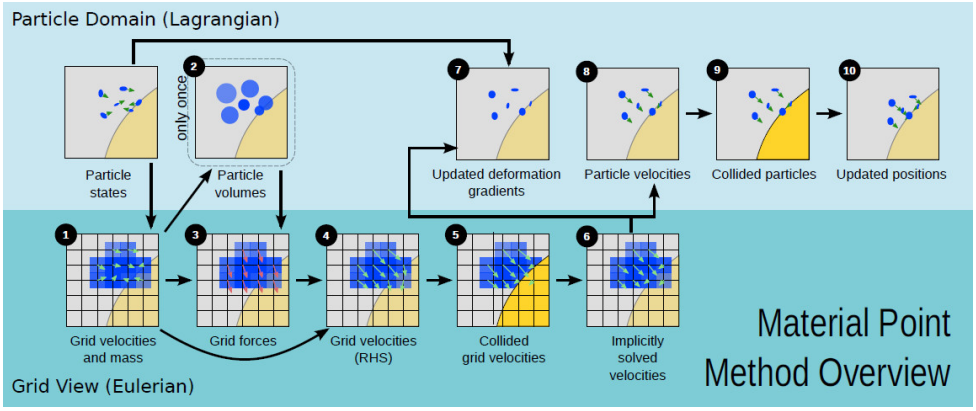

Figure 3: *Overview of the material point method (figure from [8]).*

3. MPM method

This section briefly describes the steps of the material point method used in our simulation. For a detailed description please refer to [8]. Figure 3 shows the main components of the simulation. The algorithm can be described briefly as follows. We track the material properties at particle positions. However some quantities are easier to compute on a grid, so our first step is to rasterize the particles onto a 3D grid. Next compute forces based on deformation gradients at each grid voxel, and update grid velocities. Using these velocities we perform collision detection on the grid. The final velocities are transferred back to the particles. Then calculate a new deformation gradient for each particle using refreshed voxel velocities. Table 1 list the notations of parameters used in our expressions and gives their typical values where available. This table also serves as a useful guide for implementing the MPM method, as all important particle and grid properties are listed. The next subsections describe the steps of the MPM simulation in more detail.

3.1. Rasterize particles

Each simulation frame starts with transferring particle velocities and mass to grid voxels, and ends with transferring updated velocities back to the particles. The grid can be thrown away at the end of each simulation step, so it can be redefined at the beginning of the simulation step, and can be fit to the actual particle positions. However for implementation reasons we used fixed grid position and resolution.

The connection between particles and the grid is achieved with an interpolation function. We used the same function as in [8], which use dyadic products of one-dimensional cubic B-splines. When transferring particle data, we compute the weights in a 5x5x5 voxel neighbourhood of each particle, and add the scaled particle quantity to the voxel value:

$$m_i^n = \sum_p m_p w_{ip}^n$$

Name	Notation	Typical values
Global parameters		
Young modulus	E_0	1.4×10^5
Poisson ratio	ν	0.2
Critical compression	θ_c	2.5×10^-2
Critical stretch	θ_s	7.5×10^-3
Hardening coefficient	ξ	10
Particle data		
Initial density	ρ_{p0}	400
Initial volume	V_{p0}	
Elastic force	F_{E_p}	
Plastic force	F_{P_p}	
Rotational force	R_{E_p}	
Elastic determinant	J_{E_p}	
Cell data		
Mass	m_i	
Velocity	v_i	
Force	f_i	
Collider flag	co_i	
Collider velocity	v_{co_i}	
Collider normal	n_{co_i}	

Table 1: *Notation of parameters used in our simulation framework. Default values are also listed where possible.*

$$v_i^n = \sum_p v_p^n m_p w_{ip}^n / m_i^n$$

When transferring voxel data back to particles, we again sample the 5x5x5 voxel neighbourhood of each particle, and sum their weighted average:

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

Because of the additivity, if we are interested in the derivative of one quantity we can simply transfer with the derivative of the weight function.

3.2. Particle volume and density

As an initial step particle density and volume should also be calculated. After rasterizing particle mass, density is given with the following expression:

$$\rho_p^0 = \sum_i m_i^0 w_{ip}^0 / h^3$$

Particle volume can be calculated from particle mass and density:

$$V_p^0 = m_p / \rho_p^0$$

These values are initial values, they are not going to change during simulation.

3.3. Compute forces

Calculating stress-based forces needs derivatives, which are easier to evaluate on the grid rather on the particles. Stress-based forces are defined by the deformation gradient, the final expression of these forces is:

$$f_i(x) = -\sum_p V_p^0 \cdot (2\mu T_{co-rot} + \lambda T_{contour}) \cdot (F_{E_p}^n)^T \cdot \nabla w_{ip}^n,$$

$$T_{co-rot} = F_{E_p}^n - R_{E_p}^n, T_{contour} = (J_{E_p}^n - 1) J_{E_p}^n F_{E_p}^{n\,-T},$$

where $\mu = \mu(F_P) = \mu_0 \cdot e^{\xi(1-J_{P_p})}$,

and $\lambda = \lambda(F_P) = \lambda_0 e^{\xi(1-J_{P_p})}$

Here P_p and E_p are the plastic and elastic deformations of a particle, J_{P_p} and J_{E_p} are their determinants. λ and μ are the Lamè coefficients and can be computed from the Poisson ratio and Young modulus in an initial step:

$$\lambda = \frac{E_0}{(1+\gamma)(1-2\gamma)} \text{ and } \mu = \frac{E_0}{2(1+\gamma)}$$

3.4. Update velocities

If the stress based forces are calculated, voxel velocities can be updated:

$$v_i^* = v_i^n + \triangle t / m_i \cdot f_i^n$$

Here we can also add the effect of any additional external forces like gravity.

3.5. Grid based collision

Collision handling is performed on the voxelized scene geometry after adding forces. An inelastic sliding collision is used. If the voxel is a collider cell and its velocity is not zero, the relative velocity is calculated: $v_{rel_i} = v_i - v_{co_i}$. If the relative velocity has opposing direction with the collider normal (thus the particle and the collider are not separating), only the tangential component of the velocity vector is kept. After this collision handling step we can finalize our voxel velocities v_i^{n+1}.

Here we should note that [8] used a semi-implicit integration scheme here, which had much better accuracy, so smaller time steps could be used. Due to its high implementation complexity and additional memory needs we did not implement it. This results about five times longer simulation times in our system.

3.6. Update deform gradient

From the updated velocities the deformation gradient of each particle should be calculated. This gradient is divided into a plastic and an elastic part F_{E_p} and F_{P_p}. First we assume that all changes are attributed to the elastic part:

$$\hat{F}_{E_P}^{n+1} = (I + \triangle t \nabla v_p^{n+1}) F_{E_P}^n,$$

$$\hat{F}_{P_p}^{n+1} = F_{P_p}^n,$$

where $\nabla v_p^{n+1} = \sum_i v_i^{n+1} (\nabla w_{ip}^n)^T$.

The next step is to extract the stretching part of this gradient, and identify the amount of deformation the material could not hold, thus it breaks. This is done with a singular value decomposition and clamping the singular values:

$$SVD(F_{E_p}^{n+1}) = U_p \hat{\Sigma}_p V_p^T,$$

$$\Sigma_p = clamp(\hat{\Sigma}_p, [1-\theta_c, 1+\theta_c])$$

From the clamped singular values we can recalculate the elastic and plastic deformation gradients:

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

$$F_{P_p}^{n+1} = F_{E_p}^{n+1\,-1} \hat{F}_{E_p}^{n+1} \hat{F}_{P_p}^{n+1} = V_p \Sigma_p^{-1} U_p^T \hat{F}_{E_p}^{n+1} F_{P_p}^n$$

These gradients will be used in the next simulation step to calculate stress-based forces.

3.7. Update particle velocities

Now that each voxel stores an updated velocity, these velocities should be written back to the particles. We use the same interpolation functions as for voxelizing particle data. Basically two methods can be used to update velocities: interpolate new velocities or interpolate the velocity change. The former is the classical particle in cell (PIC) method, the later is used in the fluid implicit particles (FLIP) method. For best results these two solutions should be mixed:

$$V_p^{n+1} = (1-\alpha)(\sum_i v_i^{n+1} w_{ip}^n) + \alpha(v_p^n + \sum_i (v_i^{n+1} - v_i^n) w_{ip}^n)$$

We used $\alpha = 0.9$.

3.8. Particle based collision

An additional collision handling step is needed as interpolation can bring back collision errors. We do the same calculations as in grid collision handling, but use particle velocities instead of voxel velocities, and address the grid cell the particle is in for collider information.

3.9. Update particle positions

Particles can be advected using their new velocities:

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

4. Conclusions

We introduced a reimplementation of the material point method for snow simulation. We also showed what tools can be used to efficiently integrate the simulation into an animation pipeline. The final framework is effective and easy to use. It supports any kind of animation on the scene objects.

Figure 1 shows frames from an animation when a snowball is dropped to the ground. The simulation used 600000 particles and 128x128x128 grid resolution. Figure 4 shows a snowball dropped onto a cube. Here we used 600 particles in a 40x40x40 grid. Figure 5 demonstrates animated scene geometry, where a flat layer of snow is pushed. This animation was simulated with 250000 particles in a 80x36x60 grid.

Our negative experience with the material point method was is that only high resolution grids and high particle count gives satisfying results. Low grid resolution causes a more elastic material. This sounds normal as the support of one particle is measured in voxels and not in world coordinates. Using lower grid resolution means that a particle will effect a larger surrounding, so elastic parameters should be retuned to achieve similar effects. This has a drawback that running draft simulations are circuitous. The other disadvantage is the high memory and computational cost. Simulation time steps should be kept very low, around 10^-5 if high velocities occur (which is often true in case of scene-snow interaction). The computational cost can be reduced if we use semi-implicit integration. Unfortunately this would increase memory needs even higher.

As a future work we plan to implement semi-implicit integration. Beside this we examine the parameter settings that should be used at different resolutions, to make draft simulations easier. In our implementation we have an option to duplicate a particle after simulation, thus we can increase particle count as a post processing step. Yet we place particles randomly around the original particle in a sphere. This is of course not equivalent to simulating with higher particle count, as fine detail movement is lost. We plan to extend this feature with taking into account the elastic deformation gradient calculated during simulation and deform random particle positions according to this tensor.

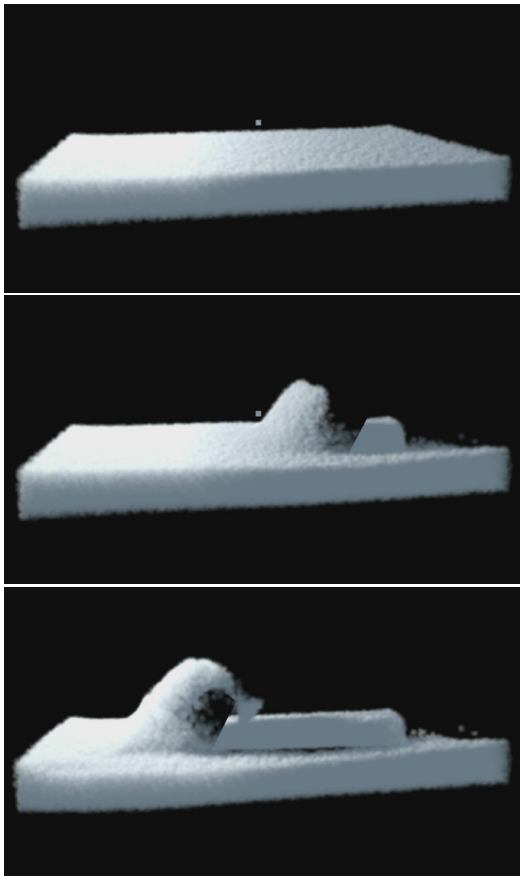

Figure 5: *A layer of snow pushed.*

Acknowledgements

This work has been supported by OTKA K-104476.

References

1. Iván Alduán and Miguel A. Otaduy. Sph granular flow with friction and cohesion. In Adam W. Bargteil and Michiel van de Panne, editors, *Symposium on Computer Animation*, pages 25–32. Eurographics Association, 2011.
2. Nathan Bell, Yizhou Yu, and Peter J. Mucha. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, pages 77–86, New York, NY, USA, 2005. ACM.
3. Toon Lenaerts and Philip Dutré. Mixing fluids and granular materials. *Comput. Graph. Forum*, 28(2):213–218, 2009.
4. Aleka McAdams, Andrew Selle, Kelly Ward, Eftychios Sifakis, and Joseph Teran. Detail preserving continuum

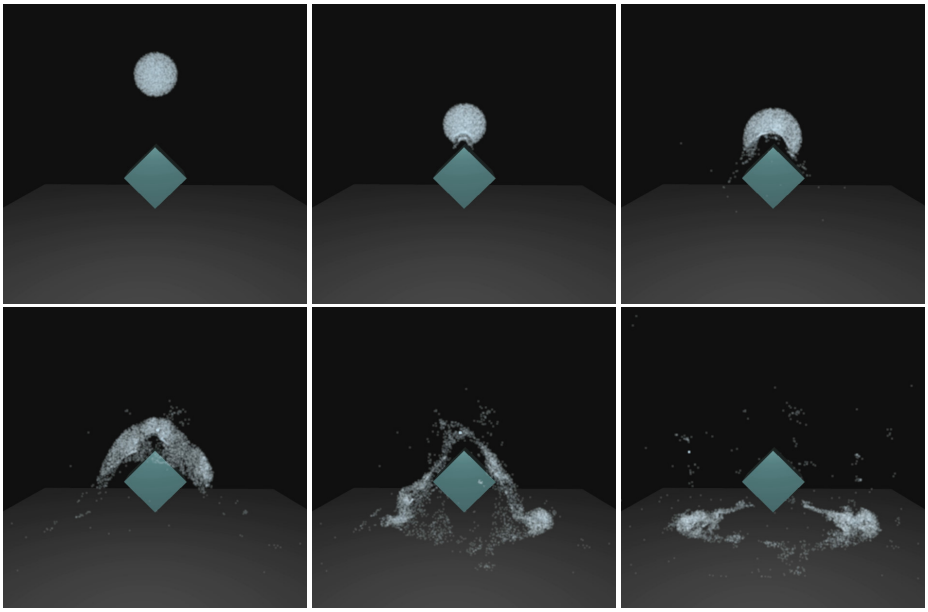

Figure 4: *A snowball dropped onto the edge of a cube.*

simulation of straight hair. *ACM Trans. Graph.*, 28(3), 2009.

5. Victor J. Milenkovic. Position-based physics: Simulating the motion of many highly interacting spheres and polyhedra. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 129–136, New York, NY, USA, 1996. ACM.

6. Rahul Narain, Abhinav Golas, and Ming C. Lin. Free-flowing granular materials with two-way solid coupling. *ACM Trans. Graph.*, 29(6):173, 2010.

7. László Neumann, Balázs Csébfalvi, Andreas König, and Eduard Gröller. Gradient estimation in volume data using 4d linear regression, 2000.

8. Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. *ACM Trans. Graph.*, 32(4):102:1–102:10, July 2013.

9. D. Sulsky, S.-J. Zhou, and H. L. Schreyer. Application of particle-in-cell method to solid mechanics. *Comp. Phys. Comm.*, 87:236–252, 1995.

10. G. Thürmer and C. A. Wüthrich. Normal computation for discrete surfaces in 3d space. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 97)*, pages 15–26, 1997.

11. Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Trans. Graph.*, 24(3):965–972, 2005.